Natural

Programming Guide

Version 6.3.13 for Windows

October 2012

Natural

This document applies to Natural Version 6.3.13 for Windows.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Copyright © 1992-2012 Software AG, Darmstadt, Germany and/or Software AG USA, Inc., Reston, VA, United States of America, and/or their licensors.

Detailed information on trademarks and patents owned by Software AG and/or its subsidiaries is located at http://documentation.softwareag.com/legal/.

Use of this software is subject to adherence to Software AG's licensing conditions and terms. These terms are part of the product documentation, located at http://documentation.softwareag.com/legal/ and/or in the root installation directory of the licensed product(s).

This software may include portions of third-party products. For third-party copyright notices and license terms, please refer to "License Texts, Copyright Notices and Disclaimers of Third-Party Products". This document is part of the product documentation, located at http://documentation.softwareag.com/legal/ and/or in the root installation directory of the licensed product(s).

Document ID: NATWIN-NNATPROGRAMMING-6313-20121005

Table of Contents

Preface xxi
I Natural Programming Modes 1
1 Natural Programming Modes 3
Purpose of Programming Modes 4
Setting/Changing the Programming Mode 5
Functional Differences 5
II Object Types 11
2 Using Natural Programming Objects 13
Types of Programming Objects 14
Creating and Maintaining Programming Objects 14
3 Data Areas 17
Use of Data Areas 18
Local Data Area 18
Global Data Area 19
Parameter Data Area 28
4 Programs, Functions, Subprograms and Subroutines 33
A Modular Application Structure 34
Multiple Levels of Invoked Objects 34
Program 36
Function 39
Subroutine 41
Subprogram 46
Processing Flow when Invoking a Routine 48
5 Processing a Rich GUI Page - Adapter 51
6 Maps 53
Benefits of Using Maps 54
Types of Maps 54
Creating Maps 55
Starting/Stopping Map Processing 55
7 Helproutines 57
Invoking Help 58
Specifying Helproutines 58
Programming Considerations for Helproutines 59
Passing Parameters to Helproutines 59
Equal Sign Option 60
Array Indices 61
Help as a Window 61
8 Multiple Use of Source Code - Copycode 63
Use of Copycode 64
Processing of Copycode 64
9 Documenting Natural Objects - Text 65
Use of Text Objects 66
Writing Text 66

10 Creating Event Driven Applications - Dialog 67
11 Creating Component Based Applications - Class 69
12 Using Non-Natural Files - Resource 71
Use of Resources 72
Shared Resources 72
Private Resources 73
API for Processing Resources 73
III Defining Fields 75
13 Use and Structure of DEFINE DATA Statement 77
Field Definitions in DEFINE DATA Statement 78
Defining Fields within a DEFINE DATA Statement 78
Defining Fields in a Separate Data Area 79
Structuring a DEFINE DATA Statement Using Level Numbers 79
14 User-Defined Variables 83
Definition of Variables 84
Referencing of Database Fields Using (r) Notation 85
Renumbering of Source-Code Line Number References 86
Format and Length of User-Defined Variables 87
Special Formats 88
Index Notation 91
Referencing a Database Array 93
Referencing the Internal Count for a Database Array (C* Notation) 101
Qualifying Data Structures 104
Examples of User-Defined Variables 105
15 Function Call 107
Calling User-Defined Functions 108
Function Result 109
Evaluation Sequence 109
Restrictions 109
Syntax Description 110
Example 114
16 Introduction to Dynamic Variables and Fields 119
Purpose of Dynamic Variables 120
Definition of Dynamic Variables 120
Value Space Currently Used for a Dynamic Variable 121
Size Limitation Check 121
Allocating/Freeing Memory Space for a Dynamic Variable 122
17 Using Dynamic and Large Variables 125
General Remarks 126
Assignments with Dynamic Variables 127
Initialization of Dynamic Variables 129
String Manipulation with Dynamic Alphanumeric Variables 129
Logical Condition Criterion (LCC) with Dynamic Variables 130
AT/IF-BREAK of Dynamic Control Fields 132
Parameter Transfer with Dynamic Variables 132

Work File Access with Large and Dynamic Variables 135
DDM Generation and Editing for Varying Length Columns 136
Accessing Large Database Objects 138
Performance Aspects with Dynamic Variables 139
Outputting Dynamic Variables 140
Dynamic X-Arrays 141
18 User-Defined Constants 143
Numeric Constants 144
Alphanumeric Constants 145
Unicode Constants 146
Date and Time Constants 149
Hexadecimal Constants 150
Logical Constants 152
Floating Point Constants 152
Attribute Constants 153
Handle Constants 154
Defining Named Constants 154
19 Initial Values (and the RESET Statement) 157
Default Initial Value of a User-Defined Variable/Array 158
Assigning an Initial Value to a User-Defined Variable/Array 158
Resetting a User-Defined Variable to its Initial Value 160
20 Redefining Fields 163
Using the REDEFINE Option of DEFINE DATA 164
Example Program Illustrating the Use of a Redefinition 165
21 Arrays 167
Defining Arrays 168
Initial Values for Arrays 169
Assigning Initial Values to One-Dimensional Arrays 169
Assigning Initial Values to Two-Dimensional Arrays 170
A Three-Dimensional Array 174
Arrays as Part of a Larger Data Structure 176
Database Arrays 177
Using Arithmetic Expressions in Index Notation 177
Arithmetic Support for Arrays 178
22 X-Arrays 181
Definition 182
Storage Management of X-Arrays 183
Storage Management of X-Group Arrays 183
Referencing an X-Array 185
Parameter Transfer with X-Arrays 186
Parameter Transfer with X-Group Arrays 187
X-Array of Dynamic Variables 188
Lower and Upper Bound of an Array 189
IV User-Defined Functions 191
23 User-Defined Functions 193

Introduction to User-Defined Functions 194
Restrictions 195
Function Call versus Subprogram Call 195
Function Definition (DEFINE FUNCTION) 198
Symbolic and Variable Function Call 198
Function Result and Parameters 198
Explicit Prototype Definition (DEFINE PROTOTYPE) 199
Implicit (Automatic) Prototype Definition 199
Prototype Cast (PT Clause) 200
Intermediate Result Definition (IR Clause) 200
Combinations of Possible Prototype Definitions 200
Evaluation Sequence of Functions in Statements 202
Using a Function as a Statement 204
V Accessing Data in a Database 207
24 Natural and Database Access 209
Database Management Systems Supported by Natural 210
Profile Parameters Influencing Database Access 211
Access through Data Definition Modules 211
Natural's Data Manipulation Language 212
Natural's Special SQL Statements 213
25 Accessing Data in an Adabas Database 215
Adabas Database Management Interfaces ADA and ADA2 216
Data Definition Modules - DDMs 216
Database Arrays 218
Defining a Database View 224
Statements for Database Access 227
Multi-Fetch Clause 239
Database Processing Loops 240
Database Update - Transaction Processing 246
Selecting Records Using ACCEPT/REJECT 253
AT START/END OF DATA Statements 257
Unicode Data 259
26 Accessing Data in an SQL Database 261
Generating Natural DDMs 262
Setting Natural Profile Parameters 262
Natural DML Statements 263
Natural SQL Statements 269
Flexible SQL 277
RDBMS-Specific Requirements and Restrictions 278
Data-Type Conversion 281
Date/Time Conversion 281
Obtaining Diagnostic Information about Database Errors 283
SQL Authorization 283
27 Accessing Data in a Tamino Database 285
Prerequisite 286

DDM and View Definitions with Natural for Tamino 286
Natural Statements for Tamino Database Access 290
Natural for Tamino Restrictions 294
VI Controlling Data Output 297
28 Report Specification - (rep) Notation 299
Use of Report Specifications 300
Statements Concerned 300
Examples of Report Specification 300
29 Layout of an Output Page 301
Statements Influencing a Report Layout 302
General Layout Example 302
30 Statements DISPLAY and WRITE 305
DISPLAY Statement 306
WRITE Statement 307
Example of DISPLAY Statement 308
Example of WRITE Statement 308
Column Spacing - SF Parameter and nX Notation 309
Tab Setting - nT Notation 310
Line Advance - Slash Notation 311
Further Examples of DISPLAY and WRITE Statements 314
31 Index Notation for Multiple-Value Fields and Periodic Groups 315
Use of Index Notation 316
Example of Index Notation in DISPLAY Statement 316
Example of Index Notation in WRITE Statement 317
32 Page Titles, Page Breaks, Blank Lines 319
Default Page Title 320
Suppress Page Title - NOTITLE Option 320
Define Your Own Page Title - WRITE TITLE Statement 321
Logical Page and Physical Page 324
Page Size - PS Parameter 326
Page Advance 326
New Page with Title 329
Page Trailer - WRITE TRAILER Statement 330
Generating Blank Lines - SKIP Statement 332
AT TOP OF PAGE Statement 333
AT END OF PAGE Statement 334
Further Example 336
33 Column Headers 337
Default Column Headers 338
Suppress Default Column Headers - NOHDR Option 338
Define Your Own Column Headers 339
Combining NOTITLE and NOHDR 340
Centering of Column Headers - HC Parameter 340
Width of Column Headers - HW Parameter 340
Filler Characters for Headers - Parameters FC and GC 341

Underlining Character for Titles and Headers - UC Parameter 342
Suppressing Column Headers - Slash Notation 343
Further Examples of Column Headers 344
34 Parameters to Influence the Output of Fields 345
Overview of Field-Output-Relevant Parameters 346
Leading Characters - LC Parameter 346
Unicode Leading Characters - LCU Parameter 347
Insertion Characters - IC Parameter 347
Unicode Insertion Characters - ICU Parameter 348
Trailing Characters - TC Parameter 348
Unicode Trailing Characters - TCU Parameter 348
Output Length - AL and NL Parameters 349
Display Length for Output - DL Parameter 349
Sign Position - SG Parameter 351
Identical Suppress - IS Parameter 353
Zero Printing - ZP Parameter 355
Empty Line Suppression - ES Parameter 355
Further Examples of Field-Output-Relevant Parameters 357
35 Code Page Edit Masks - EM Parameter 359
Use of EM Parameter 360
Edit Masks for Numeric Fields 360
Edit Masks for Alphanumeric Fields 361
Length of Fields 361
Edit Masks for Date and Time Fields 362
Customizing Separator Character Displays 362
Examples of Edit Masks 364
Further Examples of Edit Masks 366
36 Unicode Edit Masks - EMU Parameter 367
37 Vertical Displays 369
Creating Vertical Displays 370
Combining DISPLAY and WRITE 370
Tab Notation - T*field 371
Positioning Notation x/y 372
DISPLAY VERT Statement 373
Further Example of DISPLAY VERT with WRITE Statement 379
VII Further Programming Aspects 381
38 End of Statement, Program or Application 383
End of Statement 384
End of Program 384
End of Application 384
39 Processing of Application Errors 387
Natural's Default Error Processing 388
Application Specific Error Processing 388
Using an ON ERROR Statement Block 389
Using an Error Transaction Program 390

Error Processing Related Features ... 393
40 Conditional Processing - IF Statement ... 397
Structure of IF Statement ... 398
Nested IF Statements ... 400
41 Loop Processing ... 403
Use of Processing Loops ... 404
Limiting Database Loops ... 404
Limiting Non-Database Loops - REPEAT Statement ... 406
Example of REPEAT Statement ... 407
Terminating a Processing Loop - ESCAPE Statement ... 408
Loops Within Loops ... 408
Example of Nested FIND Statements ... 408
Referencing Statements within a Program ... 409
Example of Referencing with Line Numbers ... 411
Example with Statement Reference Labels ... 412
42 Control Breaks ... 415
Use of Control Breaks ... 416
AT BREAK Statement ... 416
Automatic Break Processing ... 421
Example of System Functions with AT BREAK Statement ... 422
Further Example of AT BREAK Statement ... 424
BEFORE BREAK PROCESSING Statement ... 424
Example of BEFORE BREAK PROCESSING Statement ... 424
User-Initiated Break Processing - PERFORM BREAK PROCESSING Statement ... 425
Example of PERFORM BREAK PROCESSING Statement ... 427
43 Data Computation ... 429
COMPUTE Statement ... 430
Statements MOVE and COMPUTE ... 431
Statements ADD, SUBTRACT, MULTIPLY and DIVIDE ... 432
Example of MOVE, SUBTRACT and COMPUTE Statements ... 432
COMPRESS Statement ... 433
Example of COMPRESS and MOVE Statements ... 434
Example of COMPRESS Statement ... 435
Mathematical Functions ... 436
Further Examples of COMPUTE, MOVE and COMPRESS Statements ... 437
44 System Variables and System Functions ... 439
System Variables ... 440
System Functions ... 441
Example of System Variables and System Functions ... 442
Further Examples of System Variables ... 443
Further Examples of System Functions ... 444
45 Stack ... 445
Use of Natural Stack ... 446
Stack Processing ... 446

Placing Data on the Stack ... 447
Clearing the Stack ... 448
46 Processing of Date Information ... 449
Edit Masks for Date Fields and Date System Variables ... 450
Default Edit Mask for Date - DTFORM Parameter ... 450
Date Format for Alphanumeric Representation - DF Parameter ... 451
Date Format for Output - DFOUT Parameter ... 453
Date Format for Stack - DFSTACK Parameter ... 454
Year Sliding Window - YSLW Parameter ... 455
Combinations of DFSTACK and YSLW ... 457
Year Fixed Window ... 459
Date Format for Default Page Title - DFTITLE Parameter ... 459
47 Text Notation ... 461
Defining a Text to Be Used with a Statement - the 'text' Notation ... 462
Defining a Character to Be Displayed n Times before a Field Value - the 'c'(n) Notation ... 463
48 User Comments ... 465
Using an Entire Source Code Line for Comments ... 466
Using the Latter Part of a Source Code Line for Comments ... 467
49 Logical Condition Criteria ... 469
Introduction ... 470
Relational Expression ... 471
Extended Relational Expression ... 475
Evaluation of a Logical Variable ... 476
Fields Used within Logical Condition Criteria ... 477
Logical Operators in Complex Logical Expressions ... 479
BREAK Option - Compare Current Value with Value of Previous Loop Pass ... 480
IS Option - Check whether Content of Alphanumeric or Unicode Field can be Converted ... 482
MASK Option - Check Selected Positions of a Field for Specific Content ... 484
MASK Option Compared with IS Option ... 491
MODIFIED Option - Check whether Field Content has been Modified ... 493
SCAN Option - Scan for a Value within a Field ... 494
SPECIFIED Option - Check whether a Value is Passed for an Optional Parameter ... 496
50 Rules for Arithmetic Assignment ... 499
Field Initialization ... 500
Data Transfer ... 500
Field Truncation and Field Rounding ... 503
Result Format and Length in Arithmetic Operations ... 503
Arithmetic Operations with Floating-Point Numbers ... 504
Arithmetic Operations with Date and Time ... 506
Performance Considerations for Mixed Format Expressions ... 510
Precision of Results of Arithmetic Operations ... 510

Error Conditions in Arithmetic Operations 511
Processing of Arrays 512
51 Invoking Natural Subprograms from 3GL Programs 519
Passing Parameters from the 3GL Program to the Subprogram 520
Example of Invoking a Natural Subprogram from a 3GL Program 521
52 Issuing Operating System Commands from within a Natural Program 523
Syntax 524
Parameters 524
Parameter Options 524
Return Codes 525
Examples 525
53 Statements for Internet and XML Access 527
Statements Available 528
Further References 529
VIII Portable Natural Generated Programs 531
54 Portable Natural Generated Programs 533
Compatibility 534
Endian Mode Considerations 534
ENDIAN Parameter 535
Transferring Natural Generated Programs 535
Portable FILEDIR.SAG and Error Message Files 537
IX Introduction to Event-Driven Programming 539
55 What is an Event-Driven Application? 541
Introduction 542
Program-Driven Applications 543
Event Driven Applications 544
What is Happening Here? 545
Writing Event-Driven Code 545
Components of an Event Driven Application 546
56 GUI Development Environments 549
57 GUI Design Tips 551
Introduction 552
Do Your Research 552
Screen Design 553
Menu Design 554
Color Usage 555
Consistency Check 555
58 Tasks Involved in Creating an Application 557
59 Tutorial 559
Creating a Dialog 560
Assigning Attributes to the Dialog 561
Creating Dialog Elements Inside the Dialog 563
Assigning Attributes to the Dialog Elements 565
Creating the Application's Local Data Area 566
Attaching Event Handler Code to the Dialog Element 567

Checking, Stowing and Running the Application ... 568
60 Basic Terminology ... 569
Attribute ... 570
Base Dialog ... 570
Control ... 571
Dialog ... 571
Dialog Box ... 571
Dialog Editor ... 571
Dialog Element ... 571
Event ... 572
Event Handler ... 572
Handle ... 572
Item ... 572
MDI - Multiple Document Interface ... 572
MDI Child Window ... 573
MDI Frame Window ... 573
Modal Window ... 573
SDI - Single Document Interface ... 573
Popup ... 573
Window ... 573
X Event-Driven Programming Techniques ... 575
61 Introduction ... 577
62 How To Open and Close Dialogs ... 579
Opening a Dialog ... 580
Operands ... 580
Passing Parameters to the Dialog ... 581
Permanence in Creating, Passing and Checking Data ... 582
Processing Steps When Opening a Dialog ... 583
Closing Dialogs ... 584
Initializing Attribute Values ... 584
63 How To Edit a Dialog's Enhanced Source Code ... 587
What Is The Enhanced Source Code Format? ... 588
Avoiding Incompatibilities Between Dialog Editor And Program Editor ... 589
How To Use The Enhanced Source Code Format ... 590
64 How Dialogs, Controls and Items Are Related Hierarchically ... 591
65 How To Define Dialog Elements ... 593
Introduction ... 594
HANDLE OF GUI ... 595
NULL-HANDLE ... 595
66 How To Manipulate Dialog Elements ... 597
Introduction ... 598
Querying, Setting and Modifying Attribute Values ... 598
Querying and Modifying Unicode Attribute Values ... 599
Restrictions ... 600
Numeric/Alphanumeric Assignment ... 600

67 How To Create and Delete Dialog Elements Dynamically 603
Introduction 604
Global Attribute List 604
Creating Dialog Elements Statically and Dynamically 604
How to Handle Events of Dynamically Created Dialog Elements 606
68 How To Enable and Disable Dialog Elements 609
69 Defining and Using Context Menus 611
Introduction 612
Construction 612
Association 613
Invocation 614
Manual Invocation 617
Sharing of Context Menus 618
70 Using the Clipboard and Drag and Drop 621
Introduction 622
Clipboard Specifics 624
Drag and Drop Specifics 625
Drag and Drop Insertion Marks 627
Drag-Drop Checklist 628
71 System Variables 631
72 Generated Variables 633
#DLG$PARENT 634
#DLG$WINDOW 634
73 Using the TERMINATE or STOP Statements within Dialog-based Applications 635
Introduction 636
Solution 636
Example 636
74 Message Files and Variables as Sources of Attribute Values 639
75 Triggering User-Defined Events 641
Introduction 642
Passing Parameters to the Dialog 643
76 Suppressing Events 645
77 Menu Structures, Toolbars and the MDI 647
Creating a Menu Structure 648
Parent-Child Hierarchy in Menu Structures 650
Creating a Toolbar 650
Sharing Menu Structures, Toolbars and DILs (MDI Application) 651
78 Executing Standardized Procedures 653
Introduction 654
PROCESS GUI Statement 654
79 Linking Dialog Elements to Natural Variables 655
80 Validating Input in a Dialog Element 657
81 Storing and Retrieving Client Data for a Dialog Element 659
Introduction 660

Integer Data 660
Handle Data 661
Keyed Alphanumeric Client Data 661
Keyed Client Data in Native Format 664
Key Enumeration 667
82 Creating Dialog Elements on a Canvas Control 669
83 Label Editing in Tree View and List View Controls 673
Introduction 674
Label Editing 674
Changing an Item's Label Programmatically 676
84 Working with ActiveX Controls 677
Terminology 678
How To Define an ActiveX Control 678
How To Create an ActiveX Control 678
Accessing Simple Properties 679
Colors 680
Pictures 681
Fonts 681
Variants 683
Arrays 684
Using the PROCESS GUI Statement 684
85 Working with Arrays of Dialog Elements 691
86 Working with Control Boxes 693
Introduction 694
Purpose of Exclusive Control Boxes 694
Examples of Use of Exclusive Control Boxes 695
Creation of the Wizard Pages 696
87 Working with Date and Time Picker (DTP) Controls 701
Introduction 702
Date and Time Formats 702
Inputting Dates and Times 703
Null Values 704
Calendar Colors and Font 704
88 Working with Dialog Bar Controls 705
Introduction 706
Creating a Dialog Bar Control 706
Types of Dialog Bar Control 706
UI Transparency 709
Client-Size Event 709
Close Button 710
Sample Code 710
89 Working with Error Events 715
90 Working with a Group of Radio Button Controls 717
91 Working with Image List Controls 719
Introduction 720

Creating the Image List Control 720
Adding Images 720
Composite Images 721
Scaling and Transparency 721
Bitmaps vs. Icons 722
Using an Image List 723
Referencing Images from the Image List 723
Overlay Images 724
Modifying Images 725
Deleting Images 726
Deleting the Image List Control 726
92 Working with List Box Controls and Selection Box Controls 729
93 Working with List View Controls 733
Introduction 734
View Modes 734
Setting Item Images 736
Item Placement 736
Item Selection 738
Item Activation 739
List View Columns and Sub-items 740
Sorting 743
Label Editing 745
Multiple Context Menus 747
Drag and Drop 748
94 Working with Nested Controls 753
Introduction 754
Which Control Types can be Containers? 755
Creating a Nested Control 755
Multiple Selection, Control Sequence and Clipboard Operations 756
95 Working with a Dynamic Information Line 759
96 Working with Spin Controls 761
Introduction 762
Up-Down Control 762
Buddy Control 762
Date and Time Formats 763
Inputting Dates and Times 764
Null Values 765
Calendar Colors and Font 765
97 Working with a Status Bar 767
98 Working with Status Bar Controls 769
Introduction 770
Creating a Status Bar Control 770
Using Status Bar Controls without Panes 770
Outputting Text to a Status Bar Control 771
Sharing a Status Bar in an MDI Application 772

Pane-specific Context Menus 773
99 Working with Tab Controls 775
Creating a Tab Control 776
Assigning Controls to Tabs 776
Use of Control Boxes as Tab Control Pages 777
Switching Between Controls Belonging To Different Tabs 778
Mixing Tab-dependent and Tab-independent Controls 779
Keyboard Navigation 779
Tab Switching Events 780
100 Working with Tree View Controls 781
Introduction 782
Setting Item Images 782
Item Selection 783
Item Activation 783
Item Data 784
Sorting 784
Label Editing 785
Multiple Context Menus 786
Dynamic Item Creation 786
Drag and Drop 788
101 Working with Dynamic Information Line and Status Bar 791
102 Adding a Maximize/Minimize/System Button 793
103 Defining Color 795
104 Adding Text in a Certain Font 797
105 Adding Online Help 799
106 Defining Mnemonic and Accelerator Keys 803
Introduction 804
Defining a Mnemonic Key 804
Defining an Accelerator Key 805
Displaying Accelerator Keys in Menus 805
107 Dynamic Data Exchange - DDE 807
Concepts 808
Developing a DDE Server Application 809
Developing a DDE Client Application 810
Return Codes 811
108 Object Linking and Embedding - OLE 815
What is OLE in the Natural Context? 816
OLE Documents Support 816
Embedding and Linking 816
Visual Editing - In-place Activation 817
ActiveX Controls Support 818
OLE Container Control 818
Attributes, Events and PROCESS GUI Statement Actions 821
XI Results Interface 823
109 Results Interface 825

Purpose of the Results Interface 826
Results Window Control Bar Access 826
Tab Handling 827
Image Handling 827
Context Menu Handling 828
Command Handling 828
Column Handling 829
Row Handling 829
Data Handling 830
Selection Handling 830
XII Designing Character-Based User Interfaces for Your Application 831
110 Screen Design 833
Control of the Message Line - Terminal Command %M 834
Assigning Colors to Fields - Terminal Command %= 834
Infoline - Terminal Command %X 835
Windows 836
Standard/Dynamic Layout Maps 842
Multilingual User Interfaces 842
Skill-Sensitive User Interfaces 847
111 Dialog Design 849
Field-Sensitive Processing 850
Simplifying Programming 852
Line-Sensitive Processing 853
Column-Sensitive Processing 854
Processing Based on Function Keys 854
Processing Based on Function-Key Names 855
Processing Data Outside an Active Window 856
Copying Data from a Screen 859
Statements REINPUT/REINPUT FULL 862
Object-Oriented Processing - The Natural Command Processor 863
XIII Natural Native Interface 865
112 Introduction 867
113 Interface Library and Location 869
114 Interface Versioning 871
115 Interface Access 873
116 Interface Instances and Natural Sessions 875
117 Interface Functions 877
nni_get_interface 879
nni_free_interface 880
nni_initialize 880
nni_is_initialized 882
nni_uninitialize 882
nni_enter 883
nni_try_enter 883
nni_leave 884

nni_logon 885
nni_logoff 885
nni_callnat 886
nni_create_object 887
nni_send_method 888
nni_get_property 890
nni_set_property 891
nni_delete_object 893
nni_create_parm 894
nni_create_module_parm 895
nni_create_method_parm 896
nni_create_prop_parm 897
nni_parm_count 898
nni_init_parm_s 898
nni_init_parm_sa 899
nni_init_parm_d 901
nni_init_parm_da 901
nni_get_parm_info 903
nni_get_parm 903
nni_get_parm_array 905
nni_get_parm_array_length 906
nni_put_parm 907
nni_put_parm_array 908
nni_resize_parm_array 909
nni_delete_parm 910
nni_from_string 911
nni_to_string 912
118 Parameter Description Structure 915
119 Natural Data Types 917
120 Flags 919
121 Return Codes 921
122 Natural Exception Structure 923
123 Interface Usage 925
124 Threading Issues 927
XIV NaturalX 929
125 Introduction to NaturalX 931
Why NaturalX? 932
Programming Techniques 933
126 Developing NaturalX Applications 937
Development Environments 938
Defining Classes 938
Using Classes and Objects 943
127 Distributing NaturalX Applications 947
General 948
Globally Unique Identifiers - GUIDs 950

128 ActiveX Component SoftwareAG.NaturalX.Utilities 951
Purpose 952
Interfaces 954
129 Interface INaturalXUtilities 955
Purpose 956
Methods 956
130 Interface IRunningObjects 959
Purpose 960
Methods 962
131 ActiveX Component SoftwareAG.NaturalX.Enumerator 963
Purpose 964
Interface 965
132 Interface IEnumerator 967
Purpose 968
Methods 968
XV 971
133 Natural Reserved Keywords 973
Alphabetical List of Natural Reserved Keywords 974
Performing a Check for Natural Reserved Keywords 989
134 Referenced Example Programs 991
READ Statement 992
FIND Statement 993
Nested READ and FIND Statements 997
ACCEPT and REJECT Statements 999
AT START OF DATA and AT END OF DATA Statements 1001
DISPLAY and WRITE Statements 1004
DISPLAY Statement 1008
Column Headers 1009
Field-Output-Relevant Parameters 1011
Edit Masks 1017
DISPLAY VERT with WRITE Statement 1020
AT BREAK Statement 1021
COMPUTE, MOVE and COMPRESS Statements 1022
System Variables 1025
System Functions 1028

Preface

This guide is complemental to the Natural reference documentation in that it provides basic information and some longer, in-depth articles on various aspects of programming with Natural. You should be familiar with this information before you start to write Natural applications. See also *First Steps*. This tutorial contains a series of sessions which introduce you to some of the basics of Natural programming.

Natural Programming Modes	Describes the differences between the two Natural programming modes: Reporting Mode and Structured Mode. Generally, it is recommended to use structured mode exclusively, because it provides for more clearly structured applications. Therefore, all explanations and examples in this documentation refer to structured mode. Any peculiarities of reporting mode will not be taken into consideration.
Object Types	Within an application, you can use several types of programming objects to achieve an efficient application structure. This document discusses the various types of Natural programming objects, such as data areas, programs, subprograms, subroutines, helproutines, maps.
Defining Fields	Describes how you define the fields you wish to use in a program.
User-Defined Functions	Explains the benefits of using the Natural programming object "function", shows the difference between using function calls and subprogram calls and describes the methods available for defining and calling a function.
Accessing Data in a Database	Describes various aspects of using Natural to access data in an Adabas database and in various non-Adabas databases supported by Natural. On principle, the features and examples described for Adabas also apply to other database management systems. Differences, if any, are described in the relevant interface documentation and in the *Statements* documentation or *Parameter Reference*.
Controlling Data Output	Discusses various aspects of how you can control the format of an output report created with Natural, that is, the way in which the data are displayed.
Further Programming Aspects	Discusses various other aspects of programming with Natural.
Portable Natural Generated Programs	As of Natural 5, generated programs are portable across the platforms UNIX, OpenVMS and Windows.
Introduction to Event-Driven Programming	Provides fundamental information on event-driven programming.
Event-Driven Programming Techniques	Addresses the more experienced GUI programmer and describes essential programming techniques.
Results Interface	The Results Interface enables programmers to display data within the results window of Natural Studio.

Designing Character-Based User Interfaces for Your Application	Provides information on components of Natural which you can use to design character-based user interfaces for your applications.
Natural Native Interface	Describes the Natural Native Interface which enables a non-Natural application to execute Natural code using C function calls.
NaturalX	Describes how to develop and distribute object-based applications.
Natural Reserved Keywords	Contains a list of all keywords and words that are reserved in the Natural programming language.
Referenced Example Programs	The preceding sections of the *Programming Guide* contain several examples of Natural programs. In addition, links are provided there to further example programs (mainly for reporting mode) which are contained in this separate section. **Note:** 1. All example programs shown in the *Programming Guide* are also provided in source-code form in the Natural library SYSEXPG. The example programs use data from the files EMPLOYEES and VEHICLES, which are supplied by Software AG for demonstration purposes. The Natural library SYSEXPG also includes example programs for *Natural Functions*. 2. Further example programs of using Natural statements are provided in the Natural library SYSEXSYN and are documented in the section *Referenced Example Programs* in the *Statements* documentation. 3. Please ask your Natural administrator about the availability of the libraries SYSEXPG and SYSEXSYN at your site. 4. To use any Natural example program to access an Adabas database, the Adabas nucleus parameter OPTIONS must be set to TRUNCATION.

Notation *vrs* or *vr*

When used in this documentation, the notation *vrs* or *vr* represents the relevant product version (see also *Version* in the *Glossary*).

Note: For information on Natural Application Programming Interfaces (APIs), see: *SYSEXT - Natural Application Programming Interfaces* and *SYSAPI - APIs of Natural Add-On Products* in the *Utilities* documentation.

I Natural Programming Modes

1 Natural Programming Modes

- Purpose of Programming Modes 4
- Setting/Changing the Programming Mode 5
- Functional Differences 5

Purpose of Programming Modes

Natural offers two ways of programming:

- Reporting Mode
- Structured Mode

Note: Generally, it is recommended to use structured mode exclusively, because it provides for more clearly structured applications.

Reporting Mode

Reporting mode is only useful for the creation of adhoc reports and small programs which do not involve complex data and/or programming constructs. (If you decide to write a program in reporting mode, be aware that small programs may easily become larger and more complex.)

Please note that certain Natural statements are available only in reporting mode, whereas others have a specific structure when used in reporting mode. For an overview of the statements that can be used in reporting mode, see *Reporting Mode Statements* in the *Statements* documentation.

Structured Mode

Structured mode is intended for the implementation of complex applications with a clear and well-defined program structure. The major benefits of structured mode are:

- The programs have to be written in a more structured way and are therefore easier to read and consequently easier to maintain.
- As all fields to be used in a program have to be defined in one central location (instead of being scattered all over the program, as is possible in reporting mode), overall control of the data used is much easier.

With structured mode, you also have to make more detail planning before the actual programs can be coded, thereby avoiding many programming errors and inefficiencies.

For an overview of the statements that can be used in structured mode, see *Statements Grouped by Functions* in the *Statements* documentation.

Setting/Changing the Programming Mode

The default programming mode is set by the Natural administrator with the profile parameter SM.

You can change the mode by using the Natural system command GLOBALS and the session parameter SM:

Mode	System Command
Structured	GLOBALS SM=ON
Reporting	GLOBALS SM=OFF

For further information on the Natural profile and session parameter SM, see *SM - Programming in Structured Mode* in the *Parameter Reference*.

For information on how to change the programming mode, see *SM - Programming in Structured Mode* in the *Parameter Reference*.

Functional Differences

The following major functional differences exist between reporting mode and structured mode:

- Syntax Related to Closing Loops and Functional Blocks
- Closing a Processing Loop in Reporting Mode
- Closing a Processing Loop in Structured Mode
- Location of Data Elements in a Program
- Database Reference

Note: For detailed information on functional differences that exist between the two modes, see the *Statements* documentation. It provides separate syntax diagrams and syntax element descriptions for each mode-sensitive statement. For a functional overview of the statements that can be used in reporting mode, see *Reporting Mode Statements* in the *Statements* documentation.

Syntax Related to Closing Loops and Functional Blocks

Reporting Mode:	`(CLOSE) LOOP` and `DO ... DOEND` statements are used for this purpose. `END-...` statements (except `END-DEFINE`, `END-DECIDE` and `END-SUBROUTINE`) cannot be used.
Structured Mode:	Every loop or logical construct must be explicitly closed with a corresponding `END-...` statement. Thus, it becomes immediately clear, which loop/logical constructs ends where. `LOOP` and `DO/DOEND` statements cannot be used.

The two examples below illustrate the differences between the two modes in constructing processing loops and logical conditions.

Reporting Mode Example:

The reporting mode example uses the statements `DO` and `DOEND` to mark the beginning and end of the statement block that is based on the `AT END OF DATA` condition. The `END` statement closes all active processing loops.

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

Structured Mode Example:

The structured mode example uses an `END-ENDDATA` statement to close the `AT END OF DATA` condition, and an `END-READ` statement to close the `READ` loop. The result is a more clearly structured program in which you can see immediately where each construct begins and ends:

```
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
```

Closing a Processing Loop in Reporting Mode

The statements `END`, `LOOP` (or `CLOSE LOOP`) or `SORT` may be used to close a processing loop.

The `LOOP` statement can be used to close more than one loop, and the `END` statement can be used to close all active loops. These possibilities of closing several loops with a single statement constitute a basic difference to structured mode.

A `SORT` statement closes all processing loops and initiates another processing loop.

Example 1 - LOOP:

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

Example 2 - END:

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

Example 3 - SORT:

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

Closing a Processing Loop in Structured Mode

Structured mode uses a specific loop-closing statement for each processing loop. Also, the `END` statement does not close any processing loop. The `SORT` statement must be preceded by an `END-ALL` statement, and the `SORT` loop must be closed with an `END-SORT` statement.

Example 1 - FIND:

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

Example 2 - READ:

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

Example 3 - SORT:

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

Location of Data Elements in a Program

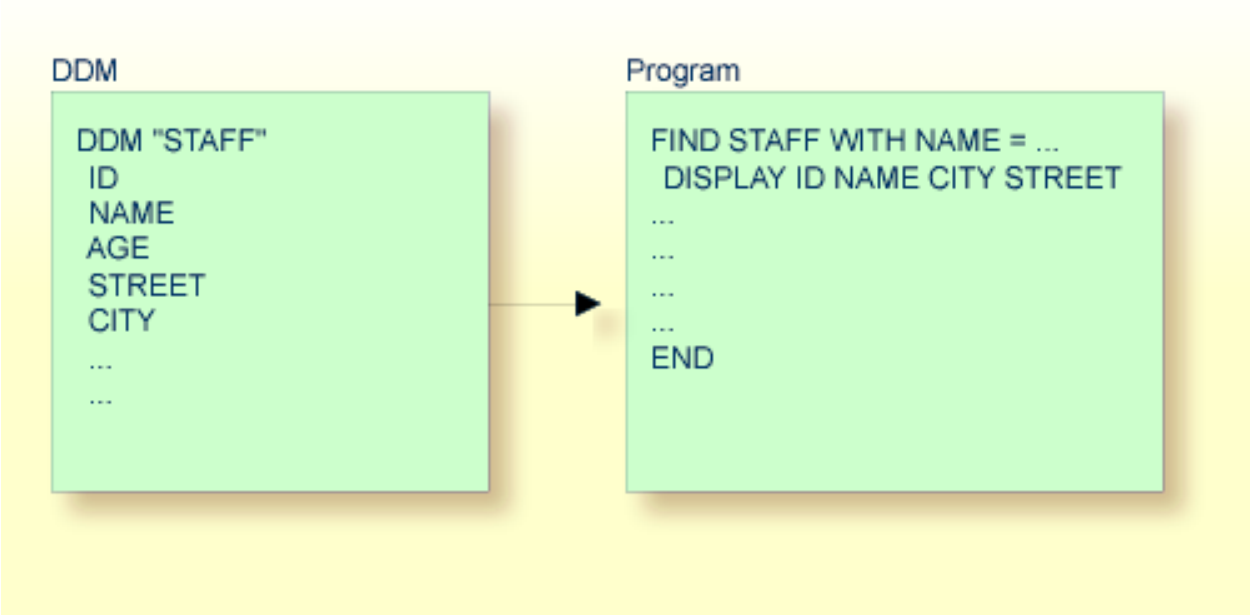
In reporting mode, you can use database fields without having to define them in a `DEFINE DATA` statement; also, you can define user-defined variables anywhere in a program, which means that they can be scattered all over the program.

In structured mode, *all* data elements to be used have to be defined in one central location (either in the `DEFINE DATA` statement at the beginning of the program, or in a data area outside the program).

Database Reference

Reporting Mode:

In reporting mode, database fields and data definition modules (DDMs) may be referenced without having been defined in a **data area**.



Structured Mode:

In structured mode, each database field to be used must be specified in a `DEFINE DATA` statement as described in *Defining Fields* and *Accessing Data in an Adabas Database*.

DDM
DDM "STAFF"
ID
NAME
AGE
STREET
CITY
...
...
Program
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

II Object Types

This part describes the various types of Natural programming objects that can be used to achieve an efficient application structure. All Natural objects are stored in Natural libraries. Natural libraries are contained in Natural system files.

Using Natural Programming Objects

Data Areas

Programs, Functions, Subprograms and Subroutines

Processing a Rich GUI Page - Adapter

Maps

Helproutines

Multiple Use of Source Code - Copycode

Documenting Natural Objects - Text

Creating Event Driven Applications - Dialog

Creating Component Based Applications - Class

Using Non-Natural Files - Resource

2 Using Natural Programming Objects

- Types of Programming Objects 14
- Creating and Maintaining Programming Objects 14

Types of Programming Objects

Within a Natural application, you can use the following types of programming objects:

- **Program**
- **Class**
- **Subprogram**
- **Function**
- **Adapter**
- **Subroutine**
- **Copycode**
- **Helproutine**
- **Text**
- **Dialog**
- **Map**
- **Local Data Area**
- **Global Data Area**
- **Parameter Data Area**
- **Resource**

Creating and Maintaining Programming Objects

To create and maintain the programming objects, you use the Natural editors.

- Local data areas, global data areas and parameter data areas are created and maintained with the data area editor.
- Maps are created and maintained with the map editor.
- Dialogs are created and maintained with the dialog editor.
- Classes are created and maintained with the Class Builder.
- All other types of objects listed above are created and maintained with the program editor.

For information about the naming conventions that apply to Natural objects, see *Object Naming Conventions*.

For detailed information on using these objects, see *Creating, Maintaining and Executing Natural Objects* in *Using Natural Studio*.

3 Data Areas

- Use of Data Areas 18
- Local Data Area 18
- Global Data Area 19
- Parameter Data Area 28

Use of Data Areas

As explained in *Defining Fields*, all fields that are to be used in a program have to be defined in a `DEFINE DATA` statement.

The fields can be defined within the `DEFINE DATA` statement itself; or they can be defined outside the program in a separate data area, with the `DEFINE DATA` statement referencing that data area.

A separate data area is a Natural object that can be used by multiple Natural programs, subprograms, subroutines, helproutines, dialogs or classes. A data area contains data element definitions, such as user-defined variables, constants and database fields from a data definition module (DDM).

All data areas are created and edited with the data area editor.

Natural supports three types of data area:

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

Local Data Area

Variables defined as local are used only within a single Natural programming object. There are two options for defining local data:

- Define local data within a program.
- Define local data outside a program in a separate Natural programming object, a local data area (LDA).

  Such a local data area is initialized when a program, subprogram or external subroutine that uses this local data area starts to execute.

For a clear application structure and for easier maintainability, it is usually better to define fields in data areas outside the programs.

Example 1 - Fields Defined Directly within a DEFINE DATA Statement:

In the following example, the fields are defined directly within the `DEFINE DATA` statement of the program.

```
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
...
```

Example 2 - Fields Defined in a Separate Data Area:

In the following example, the same fields are not defined in the `DEFINE DATA` statement of the program, but in an LDA, named `LDA39`, and the `DEFINE DATA` statement in the program contains only a reference to that data area.

Program:

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

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                          ↩
```

Global Data Area

The following topics are covered below:

- Creating and Referencing a GDA
- Creating and Deleting GDA Instances

- Data Blocks

Creating and Referencing a GDA

GDAs are created and modified with the data area editor. For further information, refer to *Data Area Editor* in the *Editors* documentation.

A GDA that is referenced by a Natural programming object must be stored in the same Natural library (or a steplib defined for this library) where the object that references this GDA is stored.

Note: Using a GDA named `COMMON` for startup:
If a GDA named `COMMON` exists in a library, the program named `ACOMMON` is invoked automatically when you `LOGON` to that library.

Important: When you build an application where multiple Natural programming objects reference a GDA, remember that modifications to the data element definitions in the GDA affect all Natural programming objects that reference that data area. Therefore these objects must be recompiled by using the `CATALOG` or `STOW` command after the GDA has been modified.

To use a GDA, a Natural programming object must reference it with the `GLOBAL` clause of the `DEFINE DATA` statement. Each Natural programming object can reference only one GDA; that is, a `DEFINE DATA` statement must not contain more than one `GLOBAL` clause.

Creating and Deleting GDA Instances

The first instance of a GDA is created and initialized at runtime when the first Natural programming object that references it starts to execute.

Once a GDA instance has been created, the data values it contains can be shared by all Natural programming objects that reference this GDA (`DEFINE DATA GLOBAL` statement) and that are invoked by a `PERFORM`, `INPUT` or `FETCH` statement. All objects that share a GDA instance are operating on the same data elements.

A new GDA instance is created if the following applies:

- A subprogram that references a GDA (*any* GDA) is invoked with a `CALLNAT` statement.
- A subprogram that does *not* reference a GDA invokes a programming object that references a GDA (*any* GDA).

If a new instance of a GDA is created, the current GDA instance is suspended and the data values it contains are stacked. The subprogram then references the data values in the newly created GDA instance. The data values in the suspended GDA instance or instances is inaccessible. A programming object only refers to one GDA instance and cannot access any previous GDA instances. A GDA data element can only be passed to a subprogram by defining the element as a parameter in the `CALLNAT` statement.

When the subprogram returns to the invoking programming object, the GDA instance it references is deleted and the GDA instance suspended previously is resumed with its data values.

A GDA instance and its contents is deleted if any of the following applies:

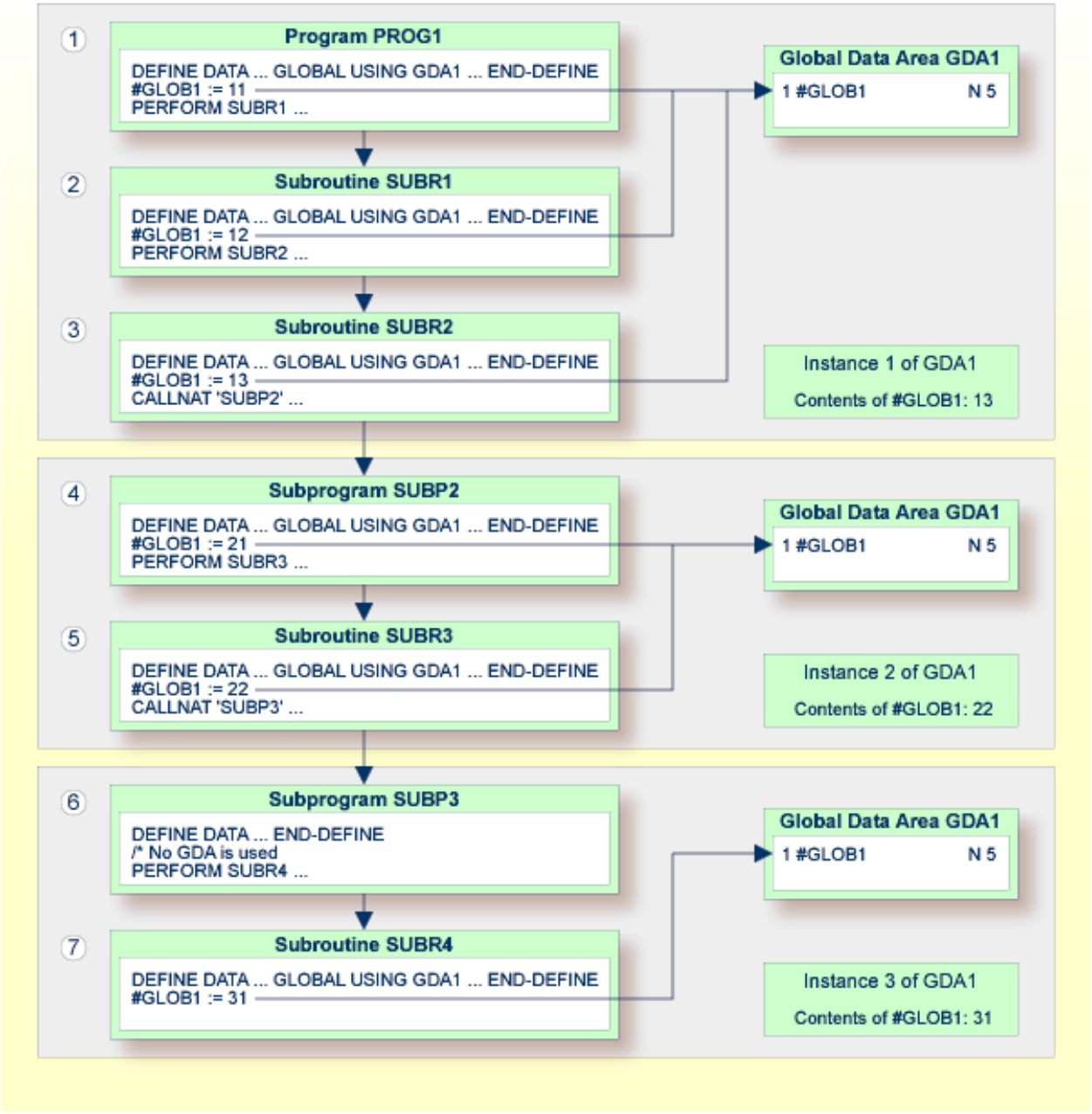
- The next `LOGON` is performed.
- Another GDA is referenced on the same level (levels are described later in this section).
- A `RELEASE VARIABLES` statement is executed. In this case, the data values in a GDA instance are reset either when a program at the level 1 finishes executing, or if the program invokes another program via a `FETCH` or `RUN` statement.

  The following graphics illustrate how programming objects reference GDAs and share data elements in GDA instances.

Sharing GDA Instances

The graphic below illustrates that a subprogram referencing a GDA cannot share the data values in a GDA instance referenced by the invoking program. A subprogram that references the same GDA as the invoking program creates a new instance of this GDA. The data elements defined in a GDA that is referenced by a subprogram can, however, be shared by a subroutine or a helproutine invoked by the subprogram.

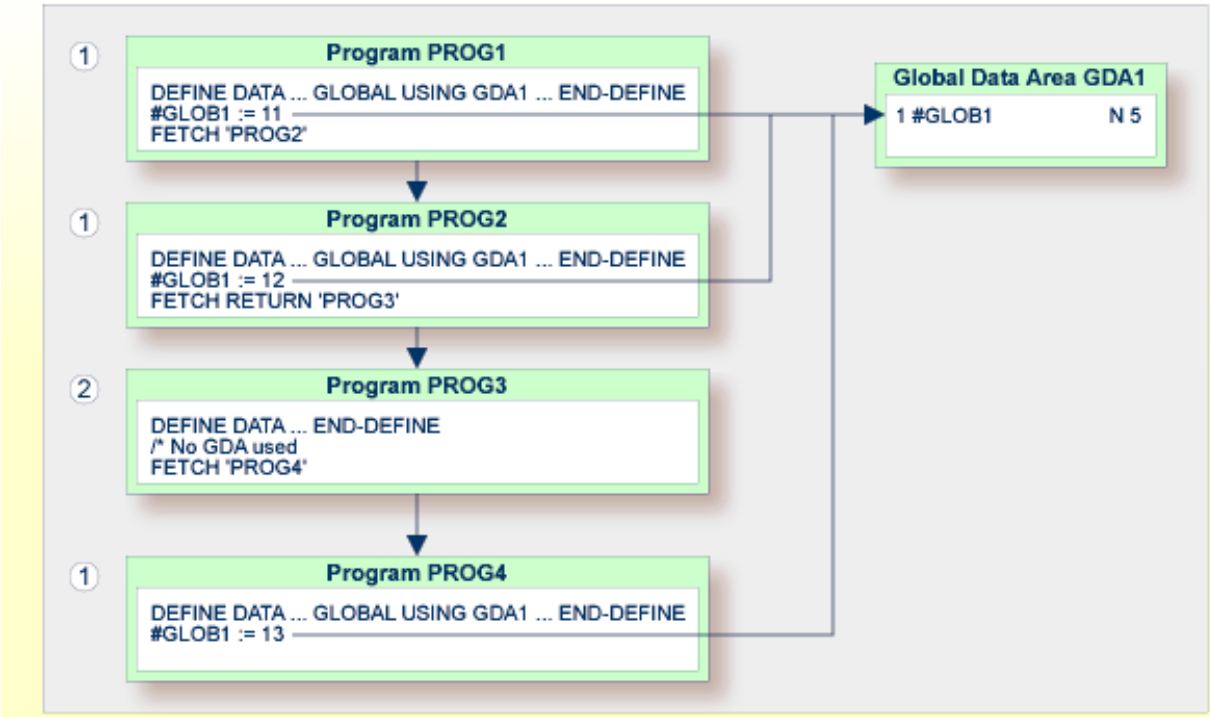
The graphic below shows three GDA instances of `GDA1` and the final values each GDA instance is assigned by the data element `#GLOB1`. The numbers (1) to (7) indicate the hierarchical levels of the programming objects.



Using FETCH or FETCH RETURN

The graphic below illustrates that programs referencing the same GDA and invoking one another with the `FETCH` or `FETCH RETURN` statement share the data elements defined in this GDA. If any of these programs does not reference a GDA, the instance of the GDA referenced previously remains active and the values of the data elements are retained.

The numbers (1) and (2) indicate the hierarchical levels of the programming objects.



Using FETCH with different GDAs

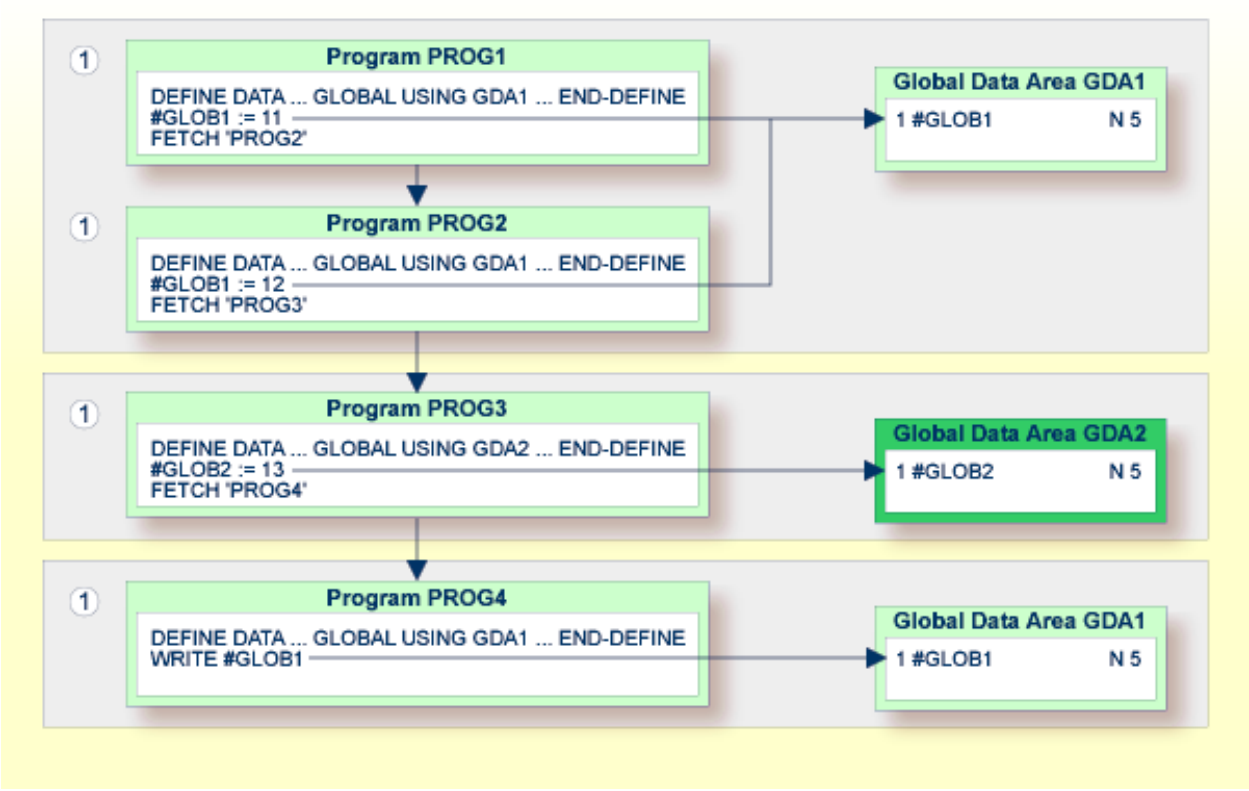
The graphic below illustrates that if a program uses the `FETCH` statement to invoke another program that references a different GDA, the current instance of the GDA (here: `GDA1`) referenced by the invoking program is deleted. If this GDA is then referenced again by another program, a new instance of this GDA is created where all data elements have their initial values.

You cannot use the `FETCH RETURN` statement to invoke another program that references a different GDA.

The number (1) indicates the hierarchical level of the programming objects.

The invoking programs `PROG3` and `PROG4` affect the GDA instances as follows:

- The statement `GLOBAL USING GDA2` in `PROG3` creates an instance of `GDA2` and deletes the current instance of `GDA1`.
- The statement `GLOBAL USING GDA1` in `PROG4` deletes the current instance of `GDA2` and creates a new instance of `GDA1`. As a result, the `WRITE` statement displays the value zero (0).



Data Blocks

To save data storage space, you can create a GDA with data blocks.

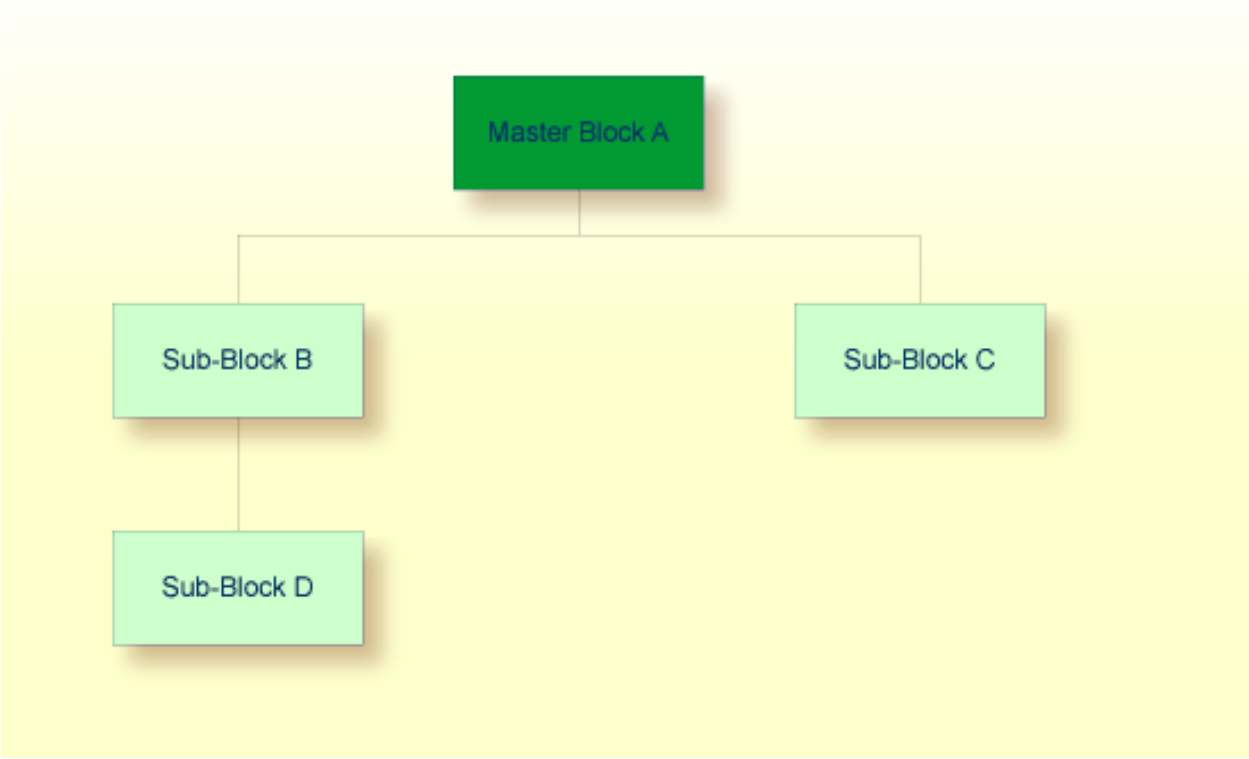
The following topics are covered below:

- Example of Data Block Usage
- Defining Data Blocks
- Block Hierarchies

Example of Data Block Usage

Data blocks can overlay each other during program execution, thereby saving storage space.

For example, given the following hierarchy, Blocks B and C would be assigned the same storage area. Thus it would not be possible for Blocks B and C to be in use at the same time. Modifying Block B would result in destroying the contents of Block C.



Defining Data Blocks

You define data blocks in the data area editor. You establish the block hierarchy by specifying which block is subordinate to which: you do this by entering the name of the "parent" block in the comment field of the block definition.

In the following example, `SUB-BLOCKB` and `SUB-BLOCKC` are subordinate to `MASTER-BLOCKA`; `SUB-BLOCKD` is subordinate to `SUB-BLOCKB`.

The maximum number of block levels is 8 (including the master block).

Example:

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
```

To make the specific blocks available to a program, you use the following syntax in the `DEFINE DATA` statement:

Program 1:

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

Program 2:

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

Program 3:

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

Program 4:

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

With this structure, Program 1 can share the data in `MASTER-BLOCKA` with Program 2, Program 3 or Program 4. However, Programs 2 and 3 cannot share the data areas of `SUB-BLOCKB` and `SUB-BLOCKC` because these data blocks are defined at the same level of the structure and thus occupy the same storage area.

Block Hierarchies

Care needs to be taken when using data block hierarchies. Let us assume the following scenario with three programs using a data block hierarchy:

Program 1:

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

Program 2:

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

Program 3:

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

Explanation:

- Program 1 uses the global data area `G-BLOCK` with `MASTER-BLOCKA` and `SUB-BLOCKB`. The program modifies a field in `SUB-BLOCKB` and fetches Program 2 which specifies only `MASTER-BLOCKA` in its data definition.
- Program 2 resets (deletes the contents of) `SUB-BLOCKB`. The reason is that a program on Level 1 (for example, a program called with a `FETCH` statement) resets any data blocks that are subordinate to the blocks it defines in its own data definition.
- Program 2 now fetches Program 3 which is to display the field modified in Program 1, but it returns an empty screen.

For details on program levels, see *Multiple Levels of Invoked Objects*.

Parameter Data Area

A subprogram is invoked with a `CALLNAT` statement. With the `CALLNAT` statement, parameters can be passed from the invoking object to the subprogram.

These parameters must be defined with a `DEFINE DATA PARAMETER` statement in the subprogram:

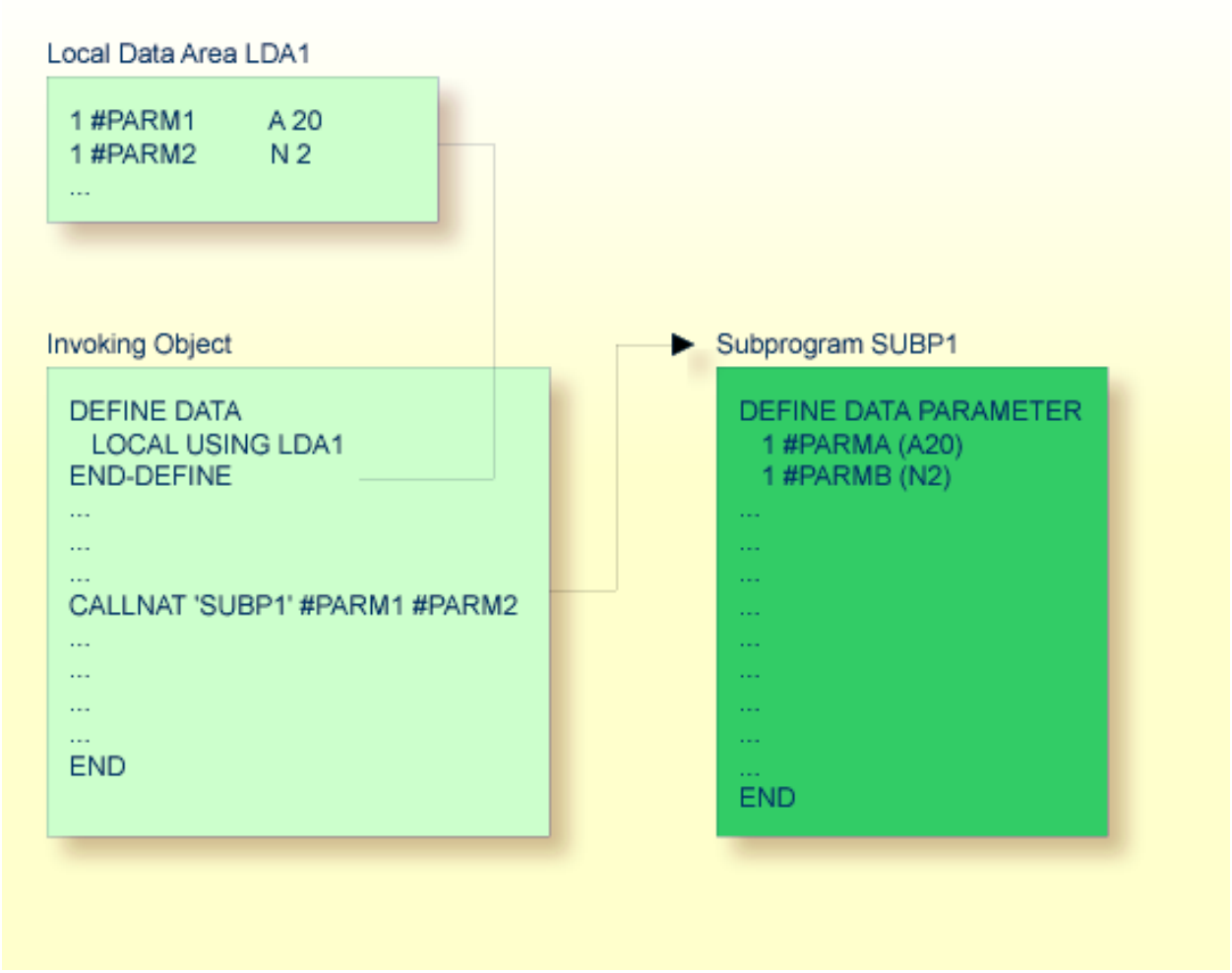
- they can be defined in the `PARAMETER` clause of the `DEFINE DATA` statement itself; or
- they can be defined in a separate parameter data area, with the `DEFINE DATA PARAMETER` statement referencing that PDA.

The following topics are covered below:

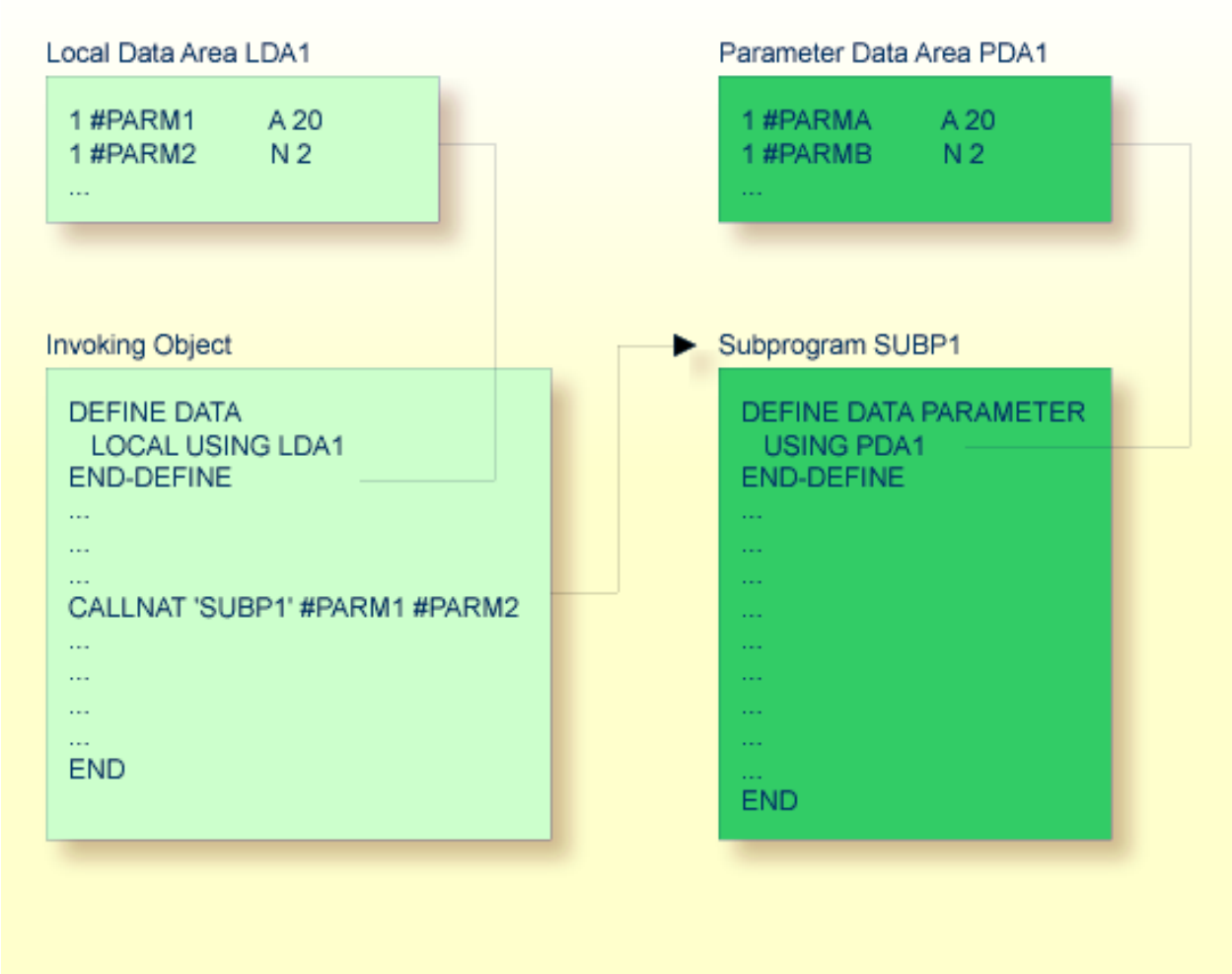
- Parameters Defined within DEFINE DATA PARAMETER Statement

- Parameters Defined in Parameter Data Area

Parameters Defined within DEFINE DATA PARAMETER Statement



Parameters Defined in Parameter Data Area



In the same way, parameters that are passed to an external subroutine via a `PERFORM` statement must be defined with a `DEFINE DATA PARAMETER` statement in the external subroutine.

In the invoking object, the parameter variables passed to the subprogram/subroutine need not be defined in a PDA; in the illustrations above, they are defined in the LDA used by the invoking object (but they could also be defined in a GDA).

The sequence, format and length of the parameters specified with the `CALLNAT`/`PERFORM` statement in the invoking object must exactly match the sequence, **format** and length of the fields specified in the `DEFINE DATA PARAMETER` statement of the invoked subprogram/subroutine. However, the names of the variables in the invoking object and the invoked subprogram/subroutine need not be the same (as the parameter data are transferred by address, not by name).

To guarantee that the data element definitions used in the invoking program are identical to the data element definitions used in the subprogram or external subroutine, you can specify a PDA

in a `DEFINE DATA LOCAL USING` statement. By using a PDA as an LDA you can avoid the extra effort of creating an LDA that has the same structure as the PDA.

4 Programs, Functions, Subprograms and Subroutines

- A Modular Application Structure 34
- Multiple Levels of Invoked Objects 34
- Program 36
- Function 39
- Subroutine 41
- Subprogram 46
- Processing Flow when Invoking a Routine 48

This document discusses those object types which can be invoked as routines; that is, as subordinate programs.

Helproutines and maps, although they are also invoked from other objects, are strictly speaking not routines as such, and are therefore discussed in separate documents; see ***Helproutines*** and ***Maps***.

A Modular Application Structure

Typically, a Natural application does not consist of a single huge program, but is split into several modules. Each of these modules will be a functional unit of manageable size, and each module is connected to the other modules of the application in a clearly defined way. This provides for a well structured application, which makes its development and subsequent maintenance a lot easier and faster.

During the execution of a main program, other programs, subprograms, subroutines, helproutines and maps can be invoked. These objects can in turn invoke other objects (for example, a subroutine can itself invoke another subroutine). Thus, the modular structure of an application can become quite complex and extend over several levels.

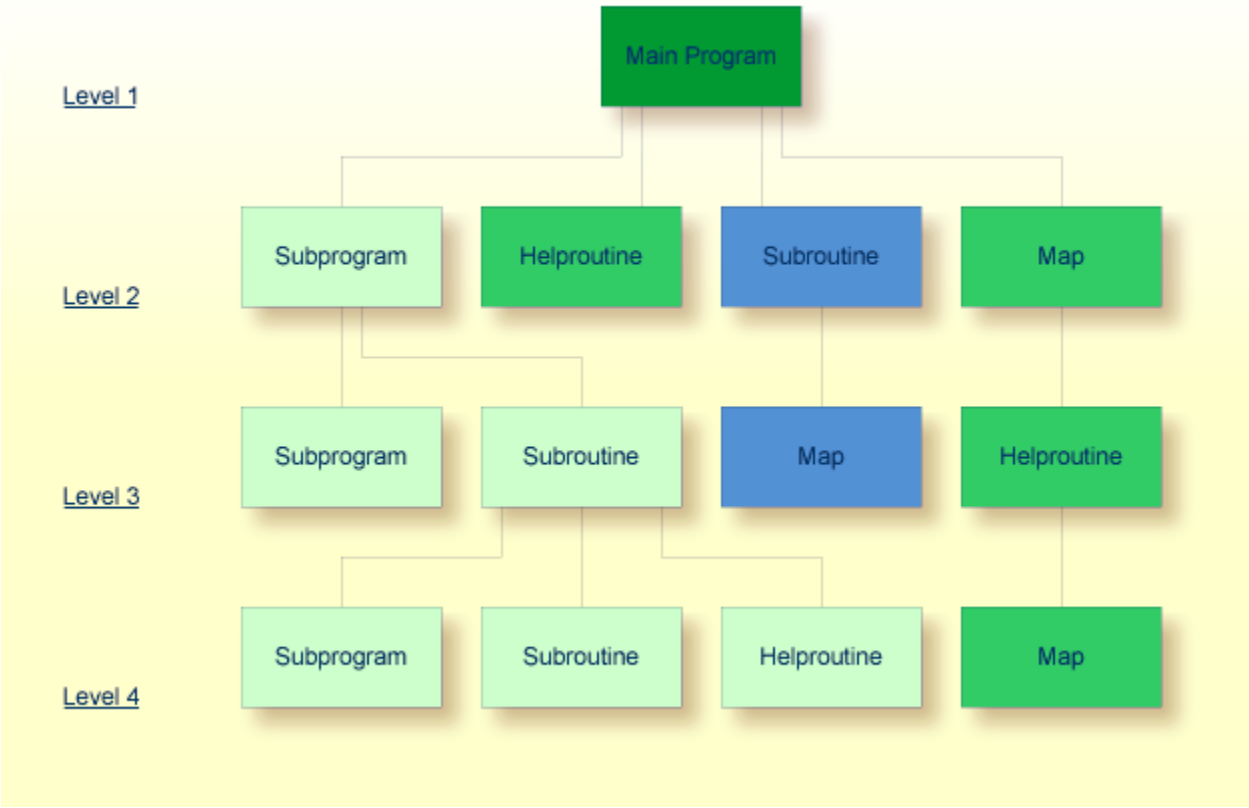
Multiple Levels of Invoked Objects

Each invoked object is one level below the level of the object from which it was invoked; that is, with each invocation of a subordinate object, the level number is incremented by 1.

Any program that is directly executed is at Level 1; any subprogram, subroutine, map or helproutine directly invoked by the main program is at Level 2; when such a subroutine in turn invokes another subroutine, the latter is at Level 3.

A program invoked with a `FETCH` statement from within another object is classified as a main program, operating from Level 1. A program that is invoked with `FETCH RETURN`, however, is classified as a subordinate program and is assigned a level one below that of the invoking object.

The following illustration is an example of multiple levels of invoked objects and also shows how these levels are counted:



If you wish to ascertain the level number of the object that is currently being executed, you can use the system variable `*LEVEL` (which is described in the *System Variables* documentation).

This document discusses the following Natural object types, which can be invoked as routines (that is, subordinate programs):

- program
- function
- subroutine
- subprogram

Helproutines and maps, although they are also invoked from other objects, are strictly speaking not routines as such, and are therefore discussed in separate documents; see ***Helproutines*** and ***Maps***.

Basically, programs, subprograms and subroutines differ from one another in the way data can be passed between them and in their possibilities of sharing each other's data areas. Therefore the decision which object type to use for which purpose depends very much on the data structure of your application.

Program

A program can be executed - and thus tested - by itself.

- To compile and execute a source program, you use the system command `RUN`.
- To execute a program that already exists in compiled form, you use the system command `EXECUTE`.

A program can also be invoked from another object with a `FETCH` or `FETCH RETURN` statement. The invoking object can be another program, a **subprogram**, **function**, **subroutine** or **helproutine**.

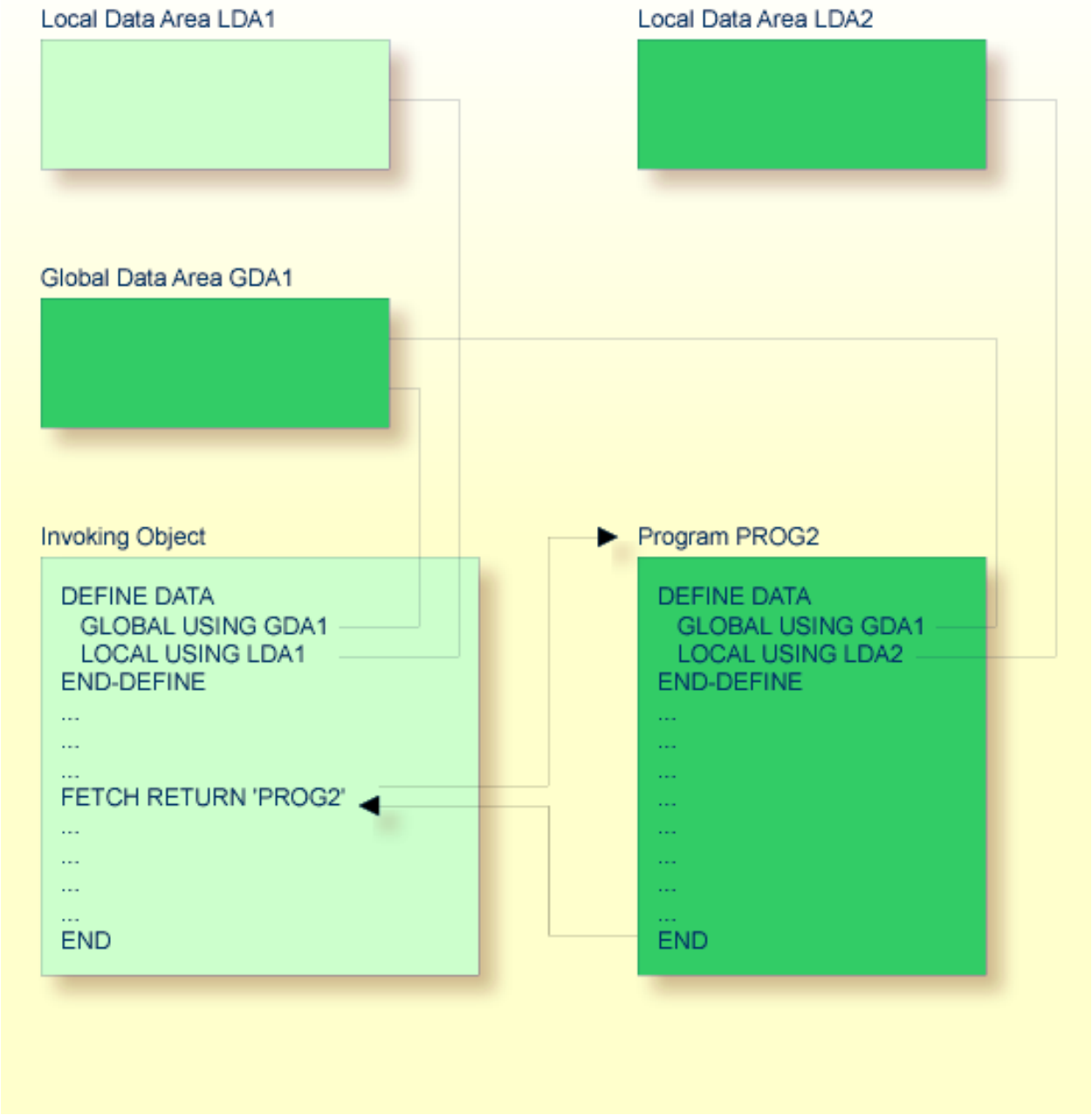
- When a program is invoked with `FETCH RETURN`, the execution of the invoking object will be suspended - not terminated - and the fetched program will be activated as a *subordinate program*. When the execution of the `FETCH`ed program is terminated, the invoking object will be re-activated and its execution continued with the statement following the `FETCH RETURN` statement.
- When a program is invoked with `FETCH`, the execution of the invoking object will be terminated and the `FETCH`ed program will be activated as a *main program*. The invoking object will not be re-activated upon termination of the fetched program.

The following topics are covered below:

- Program Invoked with FETCH RETURN

- Program Invoked with FETCH

Program Invoked with FETCH RETURN

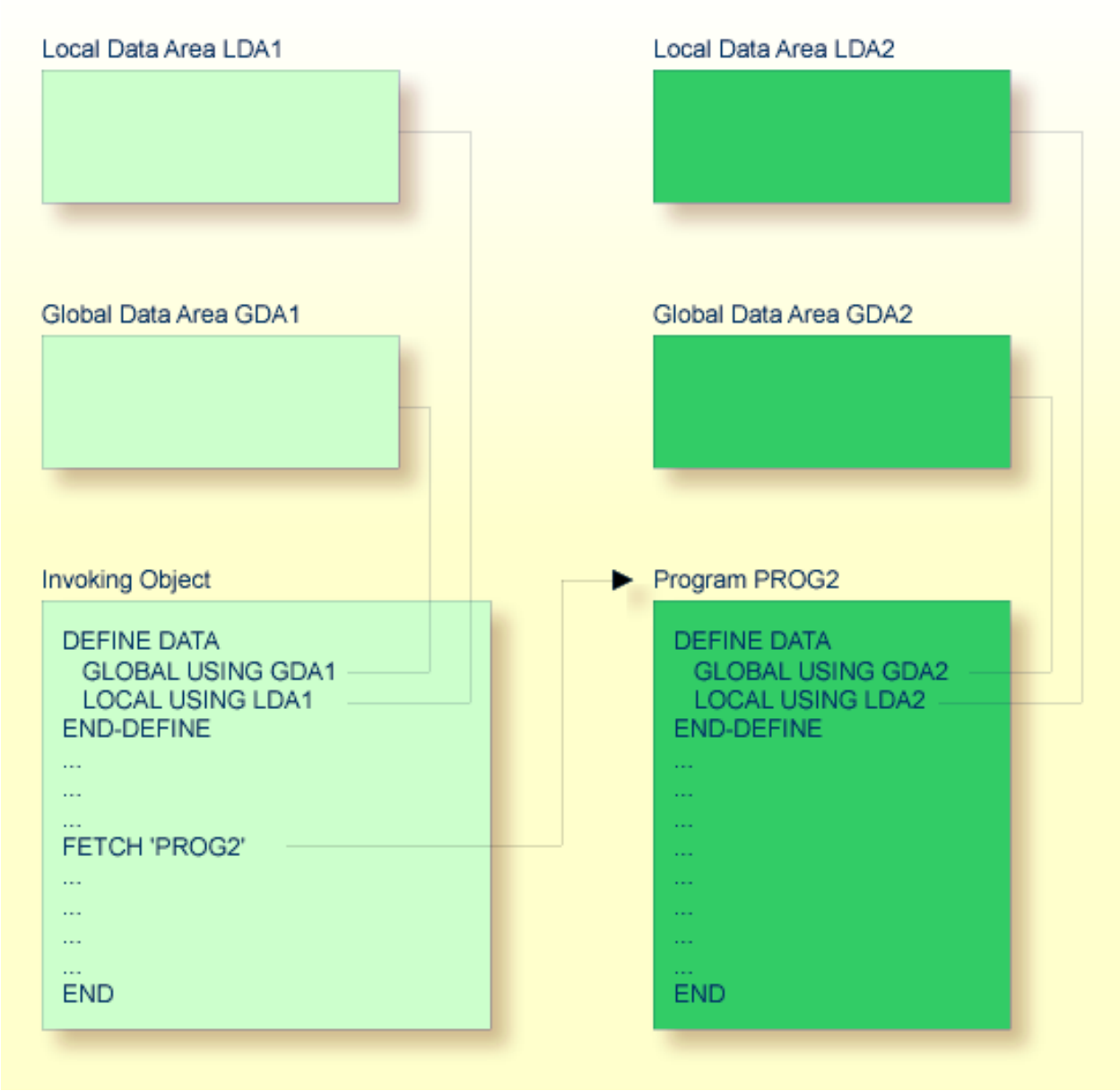


A program invoked with `FETCH RETURN` can access the global data area used by the invoking object.

In addition, every program can have its own local data area, in which the fields that are to be used only within the program are defined.

However, a program invoked with FETCH RETURN cannot have its own global data area.

Program Invoked with FETCH



A program invoked with FETCH as a main program usually establishes its own global data area (as shown in the illustration above). However, it could also use the same global data area as established by the invoking object.

Note: A source program can also be invoked with a RUN statement; see the RUN statement in the *Statements* documentation.

Function

The Natural object of type function contains one `DEFINE FUNCTION` statement for the definition of a single function and the `END` statement. The function itself can be invoked using the **function call** syntax.

The basic structure of the `DEFINE FUNCTION` statement is like this:

```
DEFINE FUNCTION function-name
   RETURNS ...
   DEFINE DATA
   ...
   END-DEFINE

   statements ...

END-FUNCTION
```

And the function call has the structure:

```
function-name (< ... >)
```

The `DEFINE FUNCTION` statement offers the keyword `RETURNS` for the result value of a function. With the `DEFINE DATA` statement, the parameters for the function call as well as local and independent variables for the function logic can be defined. A global data area (for example, `GDA1`) cannot be referenced inside the function definition.

The block of statements after the `RETURNS` keyword and the `DEFINE DATA` statement must contain all those statements which are to be executed when the function is called.

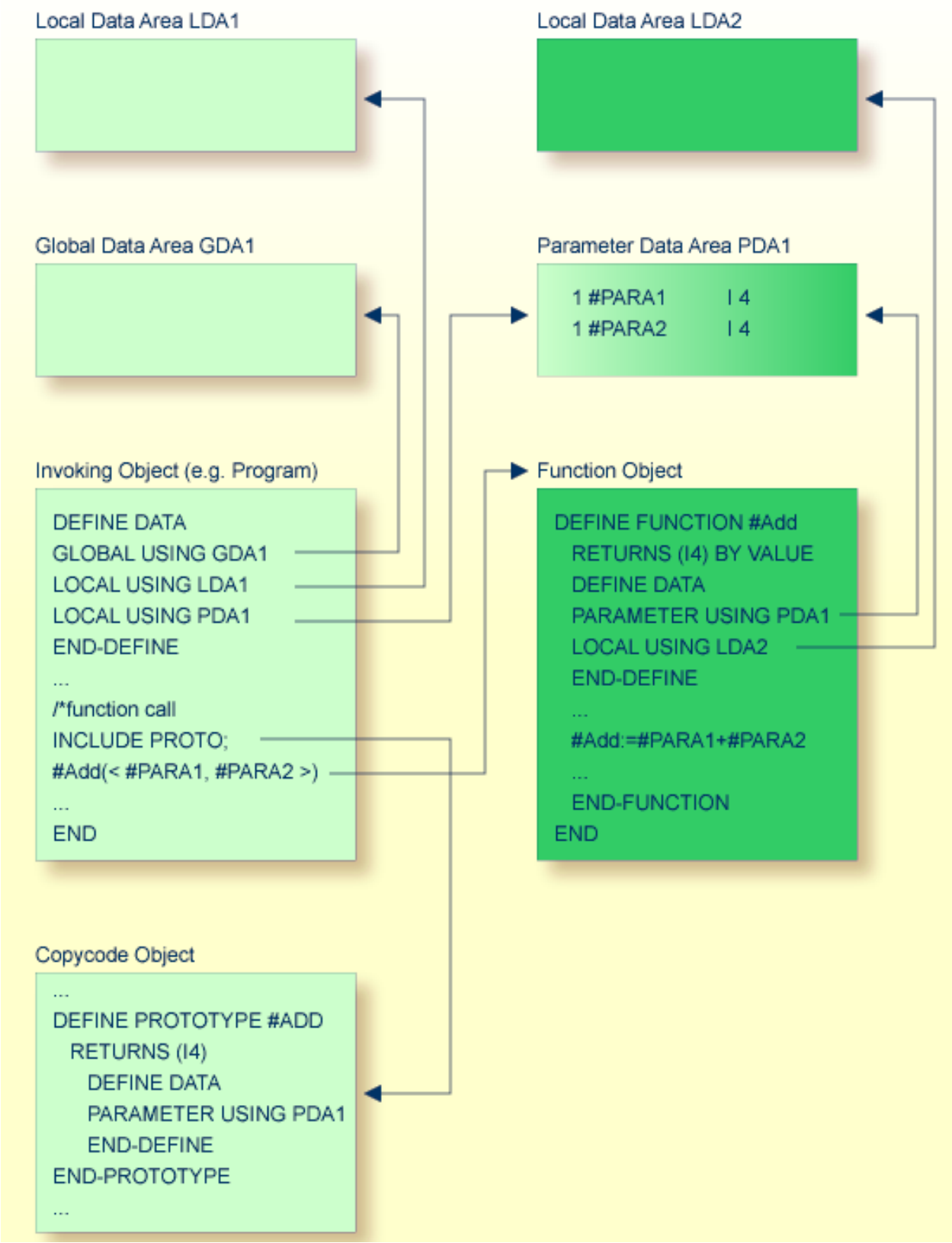
Parameter data areas (for example, `PDA1`) may be used to access parameters for function calls and function definitions in order to minimize the maintainance effort when changing parameters.

The function call can be used either as an operand within a Natural statement or stand-alone in place of a Natural statement. The arguments of the function call must be specified using a special bracket notation: `(<...>)`.

The `DEFINE PROTOTYPE` statement may be used to define the result and parameter layouts for a certain function. This may be considered if the function is not yet available or if a variable function call is to be used. At compilation time, the type of the result variable and the parameters will be checked.

The `DEFINE PROTOTYPE` statement may be included in a **copycode** object if the function call, and therefore the prototype definition can be used in several programming objects.

For further information, see the section ***User-Defined Functions***.


Local Data Area LDA1
Local Data Area LDA2
Global Data Area GDA1
Parameter Data Area PDA1
1 #PARA1 I 4
1 #PARA2 I 4
Invoking Object (e.g. Program)
DEFINE DATA
GLOBAL USING GDA1
LOCAL USING LDA1
LOCAL USING PDA1
END-DEFINE
...
/*function call
INCLUDE PROTO;
#Add(< #PARA1, #PARA2 >)
...
END
Function Object
DEFINE FUNCTION #Add
RETURNS (I4) BY VALUE
DEFINE DATA
PARAMETER USING PDA1
LOCAL USING LDA2
END-DEFINE
...
#Add:=#PARA1+#PARA2
...
END-FUNCTION
END
Copycode Object
...
DEFINE PROTOTYPE #ADD
RETURNS (I4)
DEFINE DATA
PARAMETER USING PDA1
END-DEFINE
END-PROTOTYPE
...

Subroutine

The statements that make up a subroutine must be defined within a `DEFINE SUBROUTINE ... END-SUBROUTINE` statement block.

A subroutine is invoked with a `PERFORM` statement.

A subroutine may be an *inline subroutine* or an *external subroutine*:

- **Inline Subroutine**
  An inline subroutine is defined within the object which contains the `PERFORM` statement that invokes it.
- **External Subroutine**
  An external subroutine is defined in a separate object - of type subroutine - outside the object which invokes it.

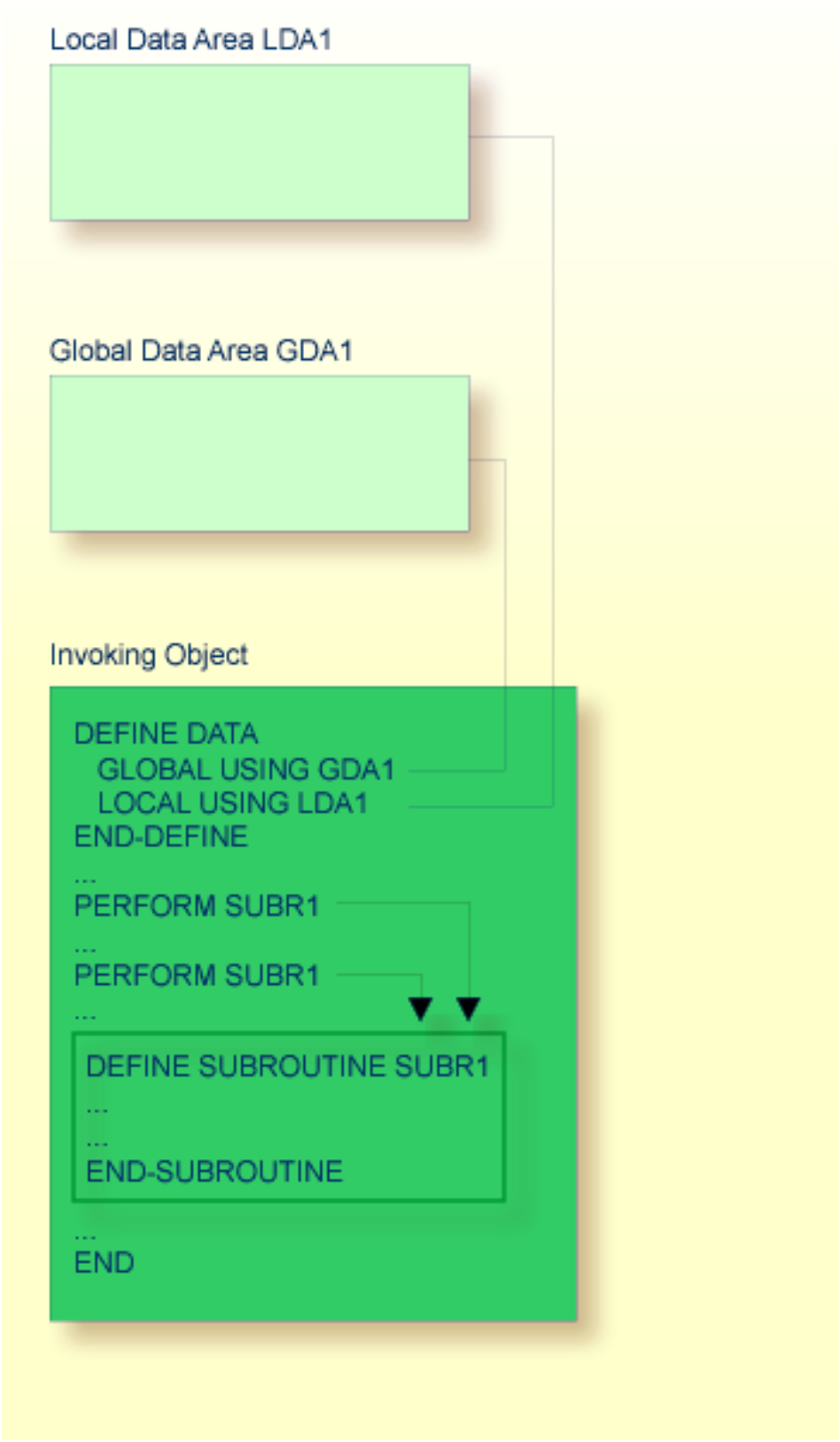
If you have a block of code which is to be executed several times within an object, it is useful to use an inline subroutine. You then only have to code this block once within a `DEFINE SUBROUTINE` statement block and invoke it with several `PERFORM` statements.

The following topics are covered below:

- Inline Subroutine
- Data Available to an Inline Subroutine
- External Subroutine

- Data Available to an External Subroutine

Inline Subroutine


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

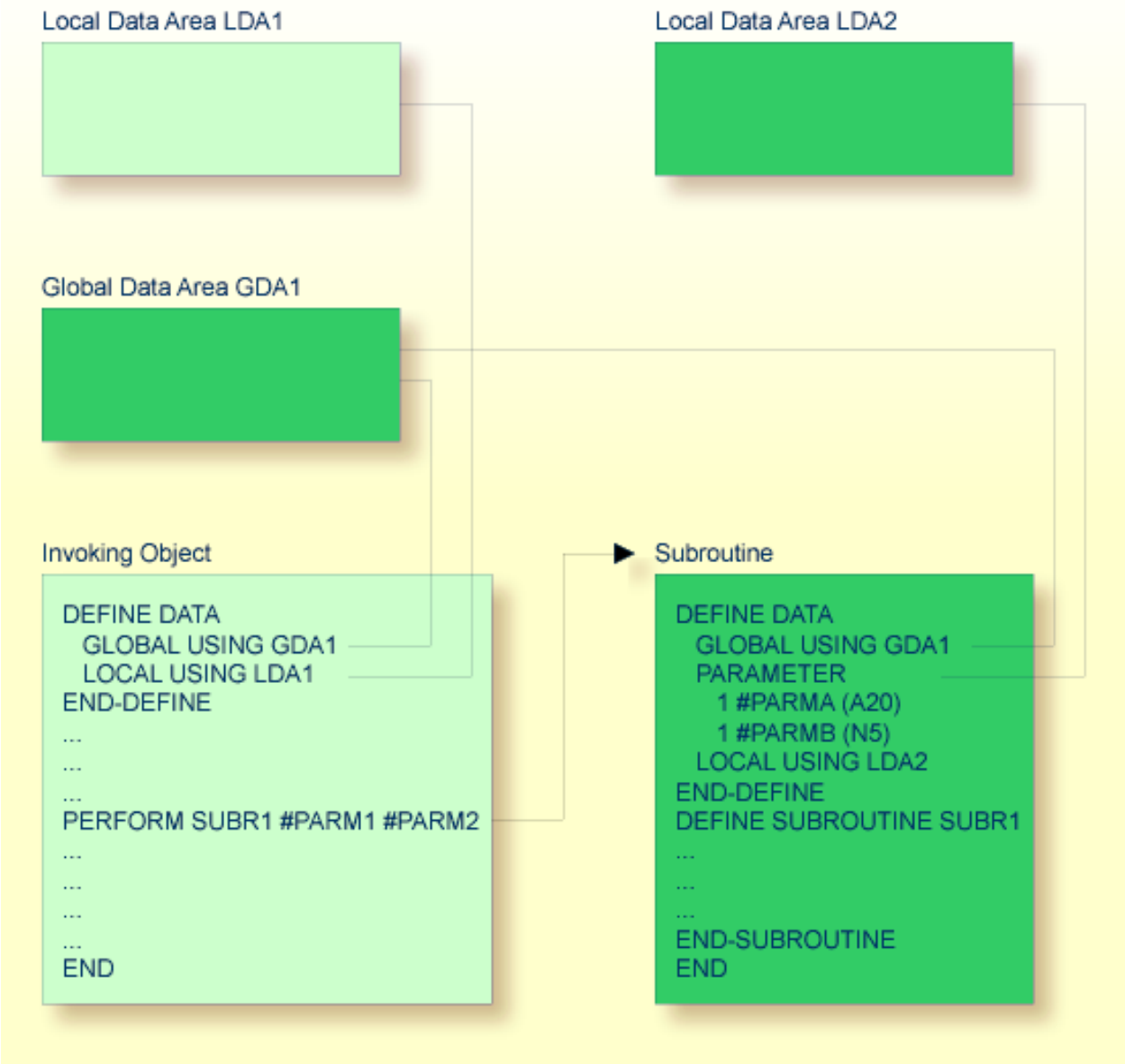
An inline subroutine can be contained within a programming object of type program, **function**, subprogram, subroutine or helproutine.

If an inline subroutine is so large that it impairs the readability of the object in which it is contained, you may consider putting it into an external subroutine, so as to enhance the readability of your application.

Data Available to an Inline Subroutine

An inline subroutine has access to the local data area and the global data area used by the object in which it is contained.

External Subroutine



An external subroutine - that is, an object of type subroutine - cannot be executed by itself. It must be invoked from another object. The invoking object can be a program, **function**, subprogram, subroutine or helproutine.

Data Available to an External Subroutine

An external subroutine can access the global data area used by the invoking object.

Moreover, parameters can be passed with the `PERFORM` statement from the invoking object to the external subroutine. These parameters must be defined either in the `DEFINE DATA PARAMETER` statement of the subroutine, or in a **parameter data area** used by the subroutine.

In addition, an external subroutine can have its **local data area**, in which the fields that are to be used only within the subroutine are defined.

However, an external subroutine cannot have its own **global data area**.

Subprogram

Typically, a subprogram would contain a generally available standard function that is used by various objects in an application.

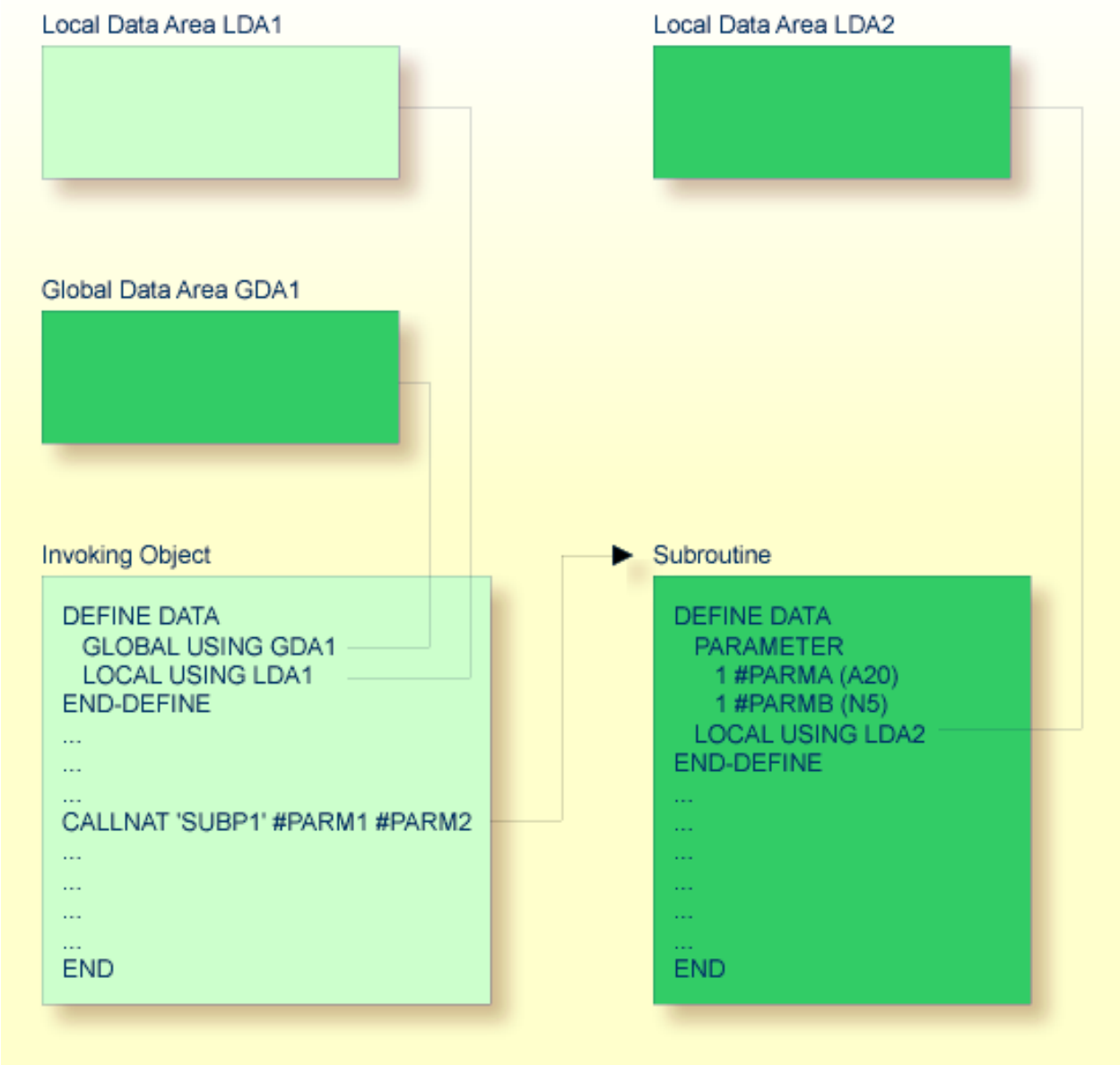
A subprogram cannot be executed by itself. It must be invoked from another object. The invoking object can be a program, **function**, subprogram, subroutine or helproutine.

A subprogram is invoked with a `CALLNAT` statement.

When the `CALLNAT` statement is executed, the execution of the invoking object will be suspended and the subprogram executed. After the subprogram has been executed, the execution of the invoking object will be continued with the statement following the `CALLNAT` statement.

Data Available to a Subprogram

With the `CALLNAT` statement, parameters can be passed from the invoking object to the subprogram. These parameters are the only data available to the subprogram from the invoking object. They must be defined either in the `DEFINE DATA PARAMETER` statement of the subprogram, or in a **parameter data area** used by the subprogram.



In addition, a subprogram can have its own local data area, in which the fields to be used within the subprogram are defined.

If a subprogram in turn invokes a subroutine or helproutine, it can also establish its own global data area to be shared with the subroutine/helproutine.

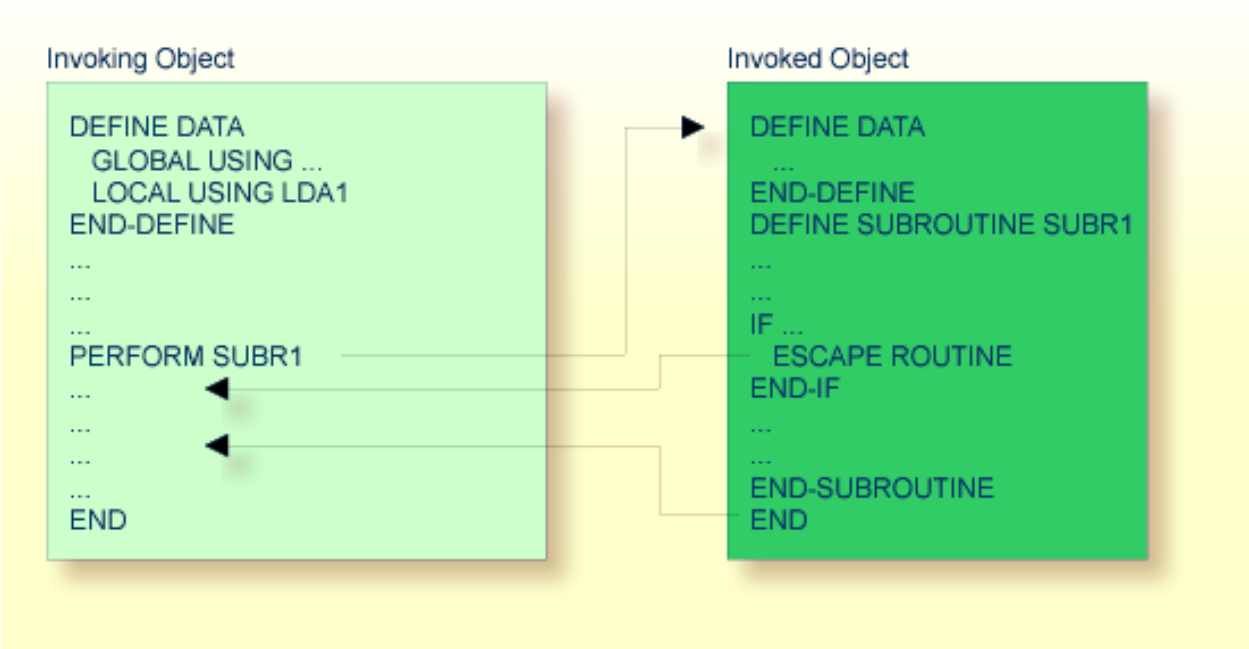
Processing Flow when Invoking a Routine

When the CALLNAT, PERFORM or FETCH RETURN statement that invokes a routine - a subprogram, an external subroutine, or a program respectively - is executed, the execution of the invoking object is suspended and the execution of the routine begins.

The execution of the routine continues until either its END statement is reached or processing of the routine is stopped by an ESCAPE ROUTINE statement being executed.

In either case, processing of the invoking object will then continue with the statement following the CALLNAT, PERFORM or FETCH RETURN statement used to invoke the routine.

Example:


Invoking Object
DEFINE DATA
GLOBAL USING ...
LOCAL USING LDA1
END-DEFINE
...
...
...
PERFORM SUBR1
...
...
...
...
END
Invoked Object
DEFINE DATA
...
END-DEFINE
DEFINE SUBROUTINE SUBR1
...
...
IF ...
ESCAPE ROUTINE
END-IF
...
...
END-SUBROUTINE
END

5 Processing a Rich GUI Page - Adapter

The Natural object of type "adapter" is used to represent a rich GUI page in a Natural application. This object type plays a similar role for the processing of a rich GUI page as the object type map plays for terminal I/O processing. But it is different from a map in that it does not contain layout information.

An object of type adapter is generated from an external page layout. It serves as an interface that enables a Natural application to send data to an external I/O system for presentation and modification, using an externally defined and stored page layout. The adapter contains the Natural code necessary to perform this task.

An application program refers to an adapter in the `PROCESS PAGE USING` statement.

For information on the object type "adapter", see the *Natural for Ajax* documentation.

6 Maps

- Benefits of Using Maps 54
- Types of Maps 54
- Creating Maps 55
- Starting/Stopping Map Processing 55

As an alternative to specifying screen layouts dynamically, the `INPUT` statement offers the possibility to use predefined map layouts which makes use of the Natural object type “map”.

Benefits of Using Maps

Using predefined map layouts rather than dynamic screen-layout specifications offers various advantages such as:

- Clearly structured applications as a result of a consequent separation of program logic and display logic.
- Map layout modifications possible without making changes to the main programs.
- The language of an applications's user interface can be easily adapted for internationalization or localization.

The benefit of using programming objects such as maps will become obvious when it comes to maintaining existing Natural applications.

Types of Maps

Maps (screen layouts) are those parts of an application which the users see on their screens.

The following types of maps exist:

- **Input Map**
  The dialog with the user is carried out via input maps.
- **Output Map**
  If an application produces any output report, this report can be displayed on the screen by using an output map.
- **Help Map**
  Help maps are, in principle, like any other maps, but when they are assigned as help, additional checks are performed to ensure their usability for help purpose.

The object type “map” comprises

- the map body which defines the screen layout and
- an associated **parameter data area** (PDA) which, as a sort of interface, contains data definitions such as name, **format**, length of each field presented on a specific map.

Related Topics:

- For information on selection boxes that can be attached to input fields, see *SB - Selection Box* in the `INPUT` statement documentation and *SB - Selection Box* in the *Parameter Reference*.
- For information on split screen maps where the upper portion may be used as an output map and the lower portion as an input map, see *Split-Screen Feature* in the `INPUT` statement documentation.

Creating Maps

Maps and help map layouts are created and edited in the map editor.

The appropriate local data area (LDA) is created and maintained in the data area editor.

Depending on the platform on which Natural is installed, these editors have either a character user interface or a graphical user interface.

Related Topics:

- For information on using the data area editor, see *Data Area Editor* in the platform-specific *Editors* documentation.
- For information on using the map editor, see *Map Editor* in the platform-specific *Editors* documentation.
- For information on input processing using screen layouts specified dynamically, see *Syntax 1 - Dynamic Screen Layout Specification* in the `INPUT` statement documentation.
- For information on input processing using a map layout created with the map editor, see *Syntax 2 - Using Predefined Map Layout* in the `INPUT` statement documentation.

Starting/Stopping Map Processing

An *input map* is invoked with an `INPUT USING MAP` statement.

An *output map* is invoked with a `WRITE USING MAP` statement.

Processing of a map can be stopped with an `ESCAPE ROUTINE` statement in a processing rule.

7 Helproutines

- Invoking Help 58
- Specifying Helproutines 58
- Programming Considerations for Helproutines 59
- Passing Parameters to Helproutines 59
- Equal Sign Option 60
- Array Indices 61
- Help as a Window 61

Helproutines have specific characteristics to facilitate the processing of help requests. They may be used to implement complex and interactive help systems. They are created with the program editor.

Invoking Help

A Natural user can invoke a Natural helproutine either by entering the help character in a field, or by pressing the help key (usually PF1). The default help character is a question mark (?).

- The help character must be entered only once.
- The help character must be the only character modified in the input string.
- The help character must be the first character in the input string.

If a helproutine is specified for a numeric field, Natural will allow a question mark to be entered for the purpose of invoking the helproutine for that field. Natural will still check that valid numeric data are provided as field input.

If not already specified, the help key may be specified with the `SET KEY` statement:

```
SET KEY PF1=HELP
```

A helproutine can only be invoked by a user if it has been specified in the **program** or **map** from which it is to be invoked.

Specifying Helproutines

A helproutine may be specified:

- in a program: at statement level and at field level;
- in a map: at map level and at field level.

If a user requests help for a field for which no help has been specified, or if a user requests help without a field being referenced, the helproutine specified at the statement or map level is invoked.

A helproutine may also be invoked by using a `REINPUT USING HELP` statement (either in the program itself or in a processing rule). If the `REINPUT USING HELP` statement contains a `MARK` option, the helproutine assigned to the marked field is invoked. If no field-specific helproutine is assigned, the map helproutine is invoked.

A `REINPUT` statement in a helproutine may only apply to `INPUT` statements within the same helproutine.

The name of a helproutine may be specified either with the session parameter HE of an INPUT statement:

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

or by using the extended field editing facility of the map editor (see *Creating Maps* and the *Editors* documentation).

The name of a helproutine may be specified as an alphanumeric constant or as an alphanumeric variable containing the name. If it is a constant, the name of the helproutine must be specified within apostrophes.

Programming Considerations for Helproutines

Processing of a helproutine can be stopped with an ESCAPE ROUTINE statement.

Be careful when using END OF TRANSACTION or BACKOUT TRANSACTION statements in a helproutine, because this will affect the transaction logic of the main program.

Passing Parameters to Helproutines

A helproutine can access the currently active **global data area** (but it cannot have its own global data area). In addition, it can have its own **local data area**.

Data may also be passed from/to a helproutine via parameters. A helproutine may have up to 20 explicit parameters and one implicit parameter. The explicit parameters are specified with the HE operand after the helproutine name:

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

The implicit parameter is the field for which the helproutine was invoked:

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

where 001 is an explicit parameter and #A is the implicit parameter/the field.

This is specified within the DEFINE DATA PARAMETER statement of the helproutine as:

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

Please note that the implicit parameter (#PARM2 in the above example) may be omitted. The implicit parameter is used to access the field for which help was requested, and to return data from the helproutine to the field. For example, you might implement a calculator program as a helproutine and have the result of the calculations returned to the field.

When help is called, the helproutine is called before the data are passed from the screen to the program data areas. This means that helproutines cannot access data entered within the same screen transaction.

Once help processing is complete, the screen data will be refreshed: any fields which have been modified by the helproutine will be updated - excluding fields which had been modified by the user before the helproutine was invoked, but including the field for which help was requested. Exception: If the field for which help was requested is split into several parts by dynamic attributes (DY session parameter), and the part in which the question mark is entered is *after* a part modified by the user, the field content will not be modified by the helproutine.

Attribute control variables are not evaluated again after the processing of the helproutine, even if they have been modified within the helproutine.

Equal Sign Option

The equal sign (=) may be specified as an explicit parameter:

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

This parameter is processed as an internal field (**format**/length A65) which contains the field name (or map name if specified at map level). The corresponding helproutine starts with:

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

This option may be used to access one common helproutine which reads the field name and provides field-specific help by accessing the application online documentation or the Predict data dictionary.

Array Indices

If the field selected by the help character or the help key is an **array** element, its indices are supplied as implicit parameters (1 - 3 depending on rank, regardless of the explicit parameters).

The **format**/length of these parameters is I2.

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

The corresponding helproutine starts with:

```
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
...
```

Help as a Window

The size of a help to be displayed may be smaller than the screen size. In this case, the help appears on the screen as a window, enclosed by a frame, for example:

```
********************************************************************************
                          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.         !
```

Within a helproutine, the size of the window may be specified as follows:

- by a `FORMAT` statement (for example, to specify the page size and line size: `FORMAT PS=15 LS=30`);
- by an `INPUT USING MAP` statement; in this case, the size defined for the map (in its map settings) is used;
- by a `DEFINE WINDOW` statement; this statement allows you to either explicitly define a window size or leave it to Natural to automatically determine the size of the window depending on its contents.

The position of a help window is computed automatically from the position of the field for which help was requested. Natural places the window as close as possible to the corresponding field without overlaying the field. With the `DEFINE WINDOW` statement, you may bypass the automatic positioning and determine the window position yourself.

For further information on window processing, please refer to the `DEFINE WINDOW` statement in the *Statements* documentation and the terminal command `%W` in the *Terminal Commands* documentation.

8 Multiple Use of Source Code - Copycode

- Use of Copycode 64
- Processing of Copycode 64

This chapter describes the advantages and the use of copycode.

Use of Copycode

Copycode is a portion of source code which can be included in another object via an `INCLUDE` statement.

So, if you have a statement block which is to appear in identical form in several objects, you may use copycode instead of coding the statement block several times. This reduces the coding effort and also ensures that the blocks are really identical.

Processing of Copycode

The copycode is included at compilation; that is, the source-code lines from the copycode are not physically inserted into the object that contains the `INCLUDE` statement, but they will be included in the compilation process and are thus part of the resulting object module.

Consequently, when you modify the source code of copycode, you also have to catalog all objects which use that copycode using the `CATALOG` system command.

Attention:

- Copycode cannot be executed on its own. It cannot be stowed with a `STOW` system command, but only saved using the `SAVE` system command.
- An `END` statement must not be placed within a copycode.

For further information, refer to the description of the `INCLUDE` statement (in the *Statements* documentation).

9 Documenting Natural Objects - Text

- Use of Text Objects 66
- Writing Text 66

The Natural object type “text” is used to write text rather than programs.

Use of Text Objects

You can use this type of object to document Natural objects in more detail than you can, for example, within the source code of a program.

Text objects may also be useful at sites where Predict is not available for program documentation purposes.

Writing Text

You write the text using the program editor.

The only difference in handling as opposed to writing programs is that there is no lower to upper case translation, that is, the text you write stays as it is.

You can write any text you wish (there is no syntax check).

Text objects can only be saved with the system command `SAVE`, they cannot be stowed with the system command `STOW`. They cannot be executed using the system command `RUN`, but only displayed in the editor.

10 Creating Event Driven Applications - Dialog

Dialogs are used in conjunction with event-driven programming when creating Natural applications for graphical user interfaces (GUIs).

For information on dialogs and event-driven programming, refer to *Event-Driven Programming*.

11 Creating Component Based Applications - Class

Classes are used to create and distribute component based applications in a client/server environment.

For details, refer to the ***NaturalX*** section of the *Programming Guide* and to the *Class Builder* section of the *Editors* documentation.

12 Using Non-Natural Files - Resource

- Use of Resources 72
- Shared Resources 72
- Private Resources 73
- API for Processing Resources 73

This section describes the Natural object of type resource.

Use of Resources

Natural distinguishes two kinds of resources:

- **Shared Resources**
  A **shared resource** is any non-Natural file that is used in a Natural application and is maintained in the Natural library system.
- **Private Resources**
  A **private resource** is a file that is assigned to one and only one Natural object and is considered to be part of that object. An object can have at most one private resource file. At the moment, only **Natural dialogs** have private resources.

Both shared and private resources belonging to a Natural library are maintained in a subdirectory named *..\RES* in the directory that represents the Natural library in the file system.

Shared Resources

A shared resource is any non-Natural file that is used in a Natural application and is maintained in the Natural library system. A non-Natural file that is to be used as a shared resource must be contained in the subdirectory named *..\RES* of a Natural library.

Example of Using a Shared Resource

The bitmap *MYPICTURE.BMP* is to be displayed in a bitmap control in a dialog `MYDLG`, contained in a library `MYLIB`. First the bitmap is put into the Natural library `MYLIB` by moving it into the directory *..\MYLIB\RES*. The following code snippet from the dialog `MYDLG` shows how it is then assigned to the bitmap control:

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

The advantages of using the bitmap as a shared resource are:

- The file name can be specified in the Natural dialog without a path name.
- The file can be kept in a Natural library together with the Natural object that uses it.

Note: In previous Natural versions non-Natural files were usually kept in a directory that was defined with the environment variable NATGUI_BMP. Existing applications that use this approach will work in the same way as before, because Natural always searches for a shared resource file in this directory, if it was not found in the current library.

Private Resources

Private resources are used internally by Natural to store binary data that is part of Natural objects. These files are recognized by the file name extension NR*, where * is a character that depends on the type of the Natural object. Natural maintains private resource files and their contents automatically. A Natural object can have a maximum of one private resource file.

Currently, only **Natural dialogs** have a private resource file. This file is used to store the configuration of ActiveX controls that are defined in a dialog and are configured with their own property pages.

For information on how to configure an ActiveX control, see *Attributes Windows for Dialogs and Dialog Elements*, *ActiveX Control Property Pages*.

Example of Private Resources

The name of the private resource file of the dialog MYDLG is *MYDLG.NR3*.

Natural creates, modifies and deletes this file automatically as needed, when the dialog is created, modified, deleted, etc.

The private resource file is used to store binary data related to the dialog MYDLG.

API for Processing Resources

In the library SYSEXT, the following application programming interface (API) exists, which gives user applications access to resources' unique user exit routines:

API	Purpose
USR4208N	Write, read, delete a resource by using short or long name.

III Defining Fields

This part describes how you define the fields you wish to use in a program. These fields can be database fields and user-defined fields.

Use and Structure of `DEFINE DATA` Statement
User-Defined Variables
Function Call
Introduction to Dynamic Variables and Fields
Using Dynamic and Large Variables
User-Defined Constants
Initial Values (and the `RESET` Statement)
Redefining Fields
Arrays
X-Arrays

Please note that only the major options of the `DEFINE DATA` statement are discussed here. Further options are described in the *Statements* documentation.

The particulars of database fields are described in ***Accessing Data in an Adabas Database***. On principle, the features and examples described there for Adabas also apply to other database management systems. Differences, if any, are described in the relevant database interface documentation and in the *Statements* documentation or *Parameter Reference*.

13 Use and Structure of DEFINE DATA Statement

- Field Definitions in DEFINE DATA Statement 78
- Defining Fields within a DEFINE DATA Statement 78
- Defining Fields in a Separate Data Area 79
- Structuring a DEFINE DATA Statement Using Level Numbers 79

The first statement in a Natural program written in **structured mode** must always be a `DEFINE DATA` statement which is used to define fields for use in a program.

For information on structural indentation of a source program, see the Natural system command `STRUCT`.

Field Definitions in DEFINE DATA Statement

In the `DEFINE DATA` statement, you define all the fields - database fields as well as user-defined variables - that are to be used in the program.

There are two ways to define the fields:

- The fields can be defined within the `DEFINE DATA` statement itself (see **below**).
- The fields can be defined outside the program in a **local or global data area**, with the `DEFINE DATA` statement referencing that data area (see **below**).

If fields are used by multiple programs/routines, they should be defined in a data area outside the programs.

For a clear application structure, it is usually better to define fields in data areas outside the programs.

Data areas are created and maintained with the data area editor.

In the **first example** below, the fields are defined within the `DEFINE DATA` statement of the program. In the **second example**, the same fields are defined in a **local data area** (LDA), and the `DEFINE DATA` statement only contains a reference to that data area.

Defining Fields within a DEFINE DATA Statement

The following example illustrates how fields can be defined within the `DEFINE DATA` statement itself:

```
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
...
```

Defining Fields in a Separate Data Area

The following example illustrates how fields can be defined in a **local data area** (LDA):

Program:

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

Local Data Area `LDA39`:

```
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
 ↩
```

Structuring a DEFINE DATA Statement Using Level Numbers

The following topics are covered:

- Structuring and Grouping Your Definitions
- Level Numbers in View Definitions
- Level Numbers in Field Groups
- Level Numbers in Redefinitions

Structuring and Grouping Your Definitions

Level numbers are used within the `DEFINE DATA` statement to indicate the structure and grouping of the definitions. This is relevant with:

- **view definitions**
- **field groups**
- **redefinitions**

Level numbers are 1- or 2-digit numbers in the range from 01 to 99 (the leading zero is optional).

Generally, variable definitions are on Level 1.

The level numbering in view definitions, redefinitions and groups must be sequential; no level numbers may be skipped.

Level Numbers in View Definitions

If you define a view, the specification of the view name must be on Level 1, and the fields the view is comprised of must be on Level 2. (For details on view definitions, see *Database Access*.)

Example of Level Numbers in View Definition:

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

Level Numbers in Field Groups

The definition of groups provides a convenient way of referencing a series of consecutive fields. If you define several fields under a common group name, you can reference the fields later in the program by specifying only the group name instead of the names of the individual fields.

The group name must be specified on Level 1, and the fields contained in the group must be one level lower.

For group names, the same naming conventions apply as for user-defined variables.

Example of Level Numbers in Group:

```
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 this example, the fields `#FIELDC`, `#FIELDD` and `#FIELDE` are defined under the common group name `#GROUPA`. The other three fields are not part of the group. Note that `#GROUPA` only serves as a group name and is not a field in its own right (and therefore does not have a format/length definition).

Level Numbers in Redefinitions

If you redefine a field, the `REDEFINE` option must be on the same level as the original field, and the fields resulting from the redefinition must be one level lower. For details on redefinitions, see *Redefining Fields*.

Example of Level Numbers 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 this example, the database field `BIRTH` is redefined as three user-defined variables, and the user-defined variable `#FIELDA` is redefined as three other user-defined variables.

14 User-Defined Variables

- Definition of Variables 84
- Referencing of Database Fields Using (r) Notation 85
- Renumbering of Source-Code Line Number References 86
- Format and Length of User-Defined Variables 87
- Special Formats 88
- Index Notation 91
- Referencing a Database Array 93
- Referencing the Internal Count for a Database Array (C* Notation) 101
- Qualifying Data Structures 104
- Examples of User-Defined Variables 105

User-defined variables are fields which you define yourself in a program. They are used to store values or intermediate results obtained at some point in program processing for additional processing or display.

See also *Naming Conventions for User-Defined Variables* in *Using Natural Studio*.

Definition of Variables

You define a user-defined variable by specifying its name and its format/length in the `DEFINE DATA` statement.

You define the characteristics of a variable with the following notation:

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

This notation follows the variable name, optionally separated by one or more blanks.

No blanks are allowed between the individual elements of the notation.

The individual elements may be specified selectively as required, but when used together, they must be separated by the characters as indicated above.

Example:

In this example, a user-defined variable of alphanumeric format and a length of 10 positions is defined with the name `#FIELD1`.

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

Notes:

1. If operating in structured mode or if a program contains a `DEFINE DATA LOCAL` clause, variables cannot be defined dynamically in a statement.
2. This does not apply to application-independent variables (AIVs); see also *Defining Application-Independent Variables*

Referencing of Database Fields Using (r) Notation

A statement label or the source-code line number can be used to refer to a previous Natural statement. This can be used to override Natural's default referencing (as described for each statement, where applicable), or for documentation purposes. See also ***Loop Processing***, ***Referencing Statements within a Program***.

The following topics are covered below:

- Default Referencing of Database Fields
- Referencing with Statement Labels
- Referencing with Source-Code Line Numbers

Default Referencing of Database Fields

Generally, the following applies if you specify no statement reference notation:

- By default, the innermost active database loop (`FIND`, `READ` or `HISTOGRAM`) in which the database field in question has been read is referenced.
- If the field is not read in any active database loop, the last previous `GET` statement (in reporting mode also `FIND FIRST` or `FIND UNIQUE` statement) is referenced which is not contained in an already closed loop and which has read the field.

Referencing with Statement Labels

Any Natural statement which causes a processing loop to be initiated and/or causes data elements to be accessed in the database may be marked with a symbolic label for subsequent referencing.

A label may be specified either in the form *`label.`* before the referencing object or in parentheses *`(label.)`* after the referencing object (but not both simultaneously).

The naming conventions for labels are identical to those for variables. The period after the label name serves to identify the entry as a label.

Example:

```
...
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
...
```

Referencing with Source-Code Line Numbers

A statement may also be referenced by using the number of the source-code line in which the statement is located.

All four digits of the line number must be specified (leading zeros must not be omitted).

Example:

```
...
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
...
```

Renumbering of Source-Code Line Number References

Numeric four-digit source-code line numbers that reference a statement (see *Referencing of Database Fields Using (r) Notation* and also *Referencing Statements within a Program*) are also renumbered if the Natural source program is renumbered. For the user's convenience and to aid in readability and debugging, all source code line number references that occur in a statement, an alphanumeric constant or a comment are renumbered. The position of the source code line number reference in the statement or alphanumeric constant (start, middle, end) does not matter.

The following patterns are recognized as being a valid source code line number reference and are renumbered (*nnnn* is a four-digit number):

Pattern	Sample Statement
(*nnnn*)	ESCAPE BOTTOM (0150)
(*nnnn*/	DISPLAY ADDRESS-LINE(0010/1:5)
(*nnnn*,	DISPLAY NAME(0010,A10/1:5)

If the left parenthesis or the four-digit number *nnnn* is followed by a blank, or the four-digit number *nnnn* is followed by a period, the pattern is not considered to be a valid source code line number reference.

To avoid that a four-digit number that is contained in an alphanumeric constant is unintentionally renumbered, the constant should be split up and the different parts should be concatenated to form a single value by use of a hyphen.

Example:

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

should be replaced by

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

Format and Length of User-Defined Variables

Format and length of a user-defined variable are specified in parentheses after the variable name.

Fixed-length variables can be defined with the following formats and corresponding lengths.

For the definition of Format and Length in dynamic variables, see *Definition of Dynamic Variables*.

Format	Explanation	Definable Length	Internal Length (in Bytes)
A	Alphanumeric	1 - 1073741824 (1GB)	1 - 1073741824
B	Binary	1 - 1073741824 (1GB)	1 - 1073741824
C	Attribute Control	-	2
D	Date	-	4
F	Floating Point	4 or 8	4 or 8
I	Integer	1 , 2 or 4	1, 2 or 4
L	Logical	-	1
N	Numeric (unpacked)	1 - 29	1 - 29
P	Packed numeric	1 - 29	1 - 15
T	Time	-	7
U	Unicode (UTF-16)	1 - 536870912 (0.5 GB)	2 - 1073741824

Length can only be specified if format is specified. With some formats, the length need not be explicitly specified (as shown in the table above).

For fields defined with format N or P, you can use decimal position notation in the form *nn*`.`*m*, where *nn* represents the number of positions before the decimal point, and *m* represents the number of positions after the decimal point. The sum of the values of *nn* and *m* must not exceed `29`, and the value of *m* must not exceed `7`.

Notes:

1. When a user-defined variable of format P is output with a `DISPLAY`, `WRITE`, or `INPUT` statement, Natural internally converts the format to N for the output.

2. In reporting mode, if format and length are not specified for a user-defined variable, the default format/length N7 will be used, unless this default assignment has been disabled by the profile/session parameter FS.

For a database field, the format/length as defined for the field in the data definition module (DDM) apply. (In reporting mode, it is also possible to define in a program a different format/length for a database field.)

In structured mode, format and length may only be specified in a data area definition or with a DEFINE DATA statement.

Example of Format/Length 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 = ...
...
```

In reporting mode, format/length may be defined within the body of the program, if no DEFINE DATA statement is used.

Example of Format/Length Definition - Reporting Mode:

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

Special Formats

In addition to the standard alphanumeric (A) and numeric (B, F, I, N, P) formats, Natural supports the following special formats:

- Format C - Attribute Control
- Formats D - Date, and T - Time
- Format L - Logical

- Format: Handle

Format C - Attribute Control

A variable defined with format C may be used to assign attributes dynamically to a field used in a DISPLAY, INPUT, PRINT, PROCESS PAGE or WRITE statement.

For a variable of format C, no length can be specified. The variable is always assigned a length of 2 bytes by Natural.

Example:

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

For further information, see the session parameter CV.

Formats D - Date, and T - Time

Variables defined with formats D and T can be used for date and time arithmetic and display. Format D can contain date information only. Format T can contain date and time information; in other words, date information is a subset of time information. Time is counted in tenths of seconds.

For variables of formats D and T, no length can be specified. A variable with format D is always assigned a length of 4 bytes (P6) and a variable of format T is always assigned a length of 7 bytes (P12) by Natural. If the profile parameter MAXYEAR is set to 9999, a variable with format D is always assigned a length of 4 bytes (P7) and a variable of format T is always assigned a length of 7 bytes (P13) by Natural.

Example:

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

For further information, see the session parameter EM and the system variables *DATX and *TIMX.

The value in a date field must be in the range from 1st January 1582 to 31st December 2699.

Format L - Logical

A variable defined of format L may be used as a logical condition criterion. It can take the value `TRUE` or `FALSE`.

For a variable of format L, no length can be specified. A variable of format L is always assigned a length of 1 byte by Natural.

Example:

```
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
```

For further information on logical value presentation, see the session parameter `EM`.

Format: Handle

A variable defined as `HANDLE OF OBJECT` can be used as an object handle.

For further information on object handles, see the section ***NaturalX***.

A variable defined as `HANDLE OF` *`dialog-element-type`* can be used as a GUI handle.

For further information on GUI handles, see ***HANDLE OF GUI*** (in *Event-Driven Programming Technics*).

Index Notation

An index notation is used for fields that represent an array.

An integer numeric constant or user-defined variable may be used in index notations. A user-defined variable can be specified using one of the following formats: N (numeric), P (packed), I (integer) or B (binary), where format B may be used only with a length of less than or equal to 4.

A system variable, system function or qualified variable cannot be used in index notations.

Array Definition - Examples:

1. #ARRAY (3)
   Defines a one-dimensional array with three occurrences.
2. FIELD (*label*.,A20/5) **or***label*.FIELD(A20/5)
   Defines an array from a database field referencing the statement marked by *label*. with format alphanumeric, length 20 and 5 occurrences.
3. #ARRAY (N7.2/1:5,10:12,1:4)
   Defines an array with format/length N7.2 and three array dimensions with 5 occurrences in the first, 3 occurrences in the second and 4 occurrences in the third dimension.
4. FIELD (*label*./i:i + 5) **or***label*.FIELD(i:i + 5)
   Defines an array from a database field referencing the statement marked by *label*..

   FIELD represents a multiple-value field or a field from a periodic group where *i* specifies the offset index within the database occurrence. The size of the array within the program is defined as 6 occurrences (i:i + 5). The database offset index is specified as a variable to allow for the positioning of the program array within the occurrences of the multiple-value field or periodic group. For any repositioning of *i*, a new access must be made to the database using a GET or GET SAME statement.

Natural allows for the definition of arrays where the index does not have to begin with 1. At runtime, Natural checks that index values specified in the reference do not exceed the maximum size of dimensions as specified in the definition.

Notes:

1. For compatibility with earlier Natural versions, an array range may be specified using a hyphen (-) instead of a colon (:).
2. A mix of both notations, however, is *not* permitted.
3. The hyphen notation is only allowed in reporting mode (but *not* in a DEFINE DATA statement).

The maximum index value is 1,073,741,824. The maximum size of a data area per programming object is 1,073,741,824 bytes (1 GB).

Simple arithmetic expressions using the plus (+) and minus (-) operators may be used in index references. When arithmetic expressions are used as indices, these operators must be preceded and followed by a blank.

Arrays in group structures are resolved by Natural field by field, not group occurrence by group occurrence.

Example of Group Array Resolution:

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

If the group defined above were output in a `WRITE` statement:

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

the occurrences would be output in the following order:

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

and *not*:

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

Array Referencing - Examples:

1. `#ARRAY (1)`
   References the first occurrence of a one-dimensional array.
2. `#ARRAY (7:12)`
   References the seventh to twelfth occurrence of a one-dimensional array.
3. `#ARRAY (i + 5)`
   References the i+fifth occurrence of a one-dimensional array.
4. `#ARRAY (5,3:7,1:4)`
   Reference is made within a three dimensional array to occurrence 5 in the first dimension, occurrences 3 to 7 (5 occurrences) in the second dimension and 1 to 4 (4 occurrences) in the third dimension.
5. An asterisk may be used to reference all occurrences within a dimension:

```
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,*)
... ↩
```

Using a Slash before an Array Occurrence

If a variable name is followed by a 4-digit number enclosed in parentheses, Natural interprets this number as a line-number reference to a statement. Therefore a 4-digit array occurrence must be preceded by a slash (/) to indicate that it is an array occurrence; for example:

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

not:

```
#ARRAY(1000)
```

because the latter would be interpreted as a reference to source code line 1000.

If an index variable name could be misinterpreted as a format/length specification, a slash (/) must be used to indicate that an index is being specified. If, for example, the occurrence of an array is defined by the value of the variable N7, the occurrence must be specified as:

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

not:

```
#ARRAY (N7)
```

because the latter would be misinterpreted as the definition of a 7-byte numeric field.

Referencing a Database Array

The following topics are covered below:

- Referencing Multiple-Value Fields and Periodic-Group Fields
- Referencing Arrays Defined with Constants
- Referencing Arrays Defined with Variables
- Referencing Multiple-Defined Arrays

Note: Before executing the following example programs, please run the program `INDEXTST` in the library `SYSEXPG` to create an example record that uses 10 different language codes.

Referencing Multiple-Value Fields and Periodic-Group Fields

A multiple-value field or periodic-group field within a view/DDM may be defined and referenced using various index notations.

For example, the first to tenth values and the Ith to Ith+10 values of the same multiple-value field/periodic-group field of a database record:

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

or:

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

Notes:

1. The same lower bound index may only be used once per array (this applies to constant indexes as well as variable indexes).
2. For an array definition using a variable index, the lower bound must be specified using the variable by itself, and the upper bound must be specified using the same variable plus a constant.

Referencing Arrays Defined with Constants

An array defined with constants may be referenced using either constants or variables. The upper bound of the array cannot be exceeded. The upper bound will be checked by Natural at compilation time if a constant is used.

Reporting Mode Example:

```
** 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
```

Structured Mode Example:

```
** 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
```

If a multiple-value field or periodic-group field is defined several times using constants and is to be referenced using variables, the following syntax is used.

Reporting Mode Example:

```
** 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
```

Structured Mode Example:

```
** 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
```

Referencing Arrays Defined with Variables

Multiple-value fields or periodic-group fields in arrays defined with variables must be referenced using the same variable.

Reporting Mode Example:

```
** 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
```

Structured Mode Example:

```
** 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
```

If a different index is to be used, an unambiguous reference to the first encountered definition of the array with variable index must be made. This is done by qualifying the index expression as shown below.

Reporting Mode Example:

```
** 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
```

Structured Mode Example:

```
** 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
```

The expression `I.` is used to create an unambiguous reference to the array definition and "positions" to the first value within the read array range (`LANG(I.1:5)`).

The current content of `I` at the time of the database access determines the starting occurrence of the database array.

Referencing Multiple-Defined Arrays

For multiple-defined arrays, a reference with qualification of the index expression is usually necessary to ensure an unambiguous reference to the desired array range.

Reporting Mode Example:

```
** 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
```

Structured Mode Example:

```
** 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
```

A similar syntax is also used if multiple-value fields or periodic-group fields are defined using index variables.

Reporting Mode Example:

```
** 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
```

Structured Mode Example:

```
** 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
```

Referencing the Internal Count for a Database Array (C* Notation)

It is sometimes necessary to reference a multiple-value field and/or a periodic group without knowing how many values/occurrences exist in a given record. Adabas maintains an internal count of the number of values of each multiple-value field and the number of occurrences of each periodic group. This count may be referenced by specifying `C*` immediately before the field name.

Note concerning databases other than Adabas:

Tamino	With XML databases, the `C*` notation cannot be used.
SQL	With SQL databases, the `C*` notation cannot be used.

The explicit format and length permitted to declare a `C*` field is either

- integer (I) with a length of 2 bytes (I2) or 4 bytes (I4),
- numeric (N) or packed (P) with only integer (but no precision) digits; for example (N3).

If no explicit format and length is supplied, format/length (N3) is assumed as default.

Examples:

`C*LANG`	Returns the count of the number of values for the multiple-value field `LANG`.
`C*INCOME`	Returns the count of the number of occurrences for the periodic group `INCOME`.
`C*BONUS(1)`	Returns the count of the number of values for the multiple-value field `BONUS` in periodic group occurrence 1 (assuming that `BONUS` is a multiple-value field within a periodic group.)

Example Program Using the C* Variable:

```
** 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
```

Output of Program `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* for Multiple-Value Fields Within Periodic Groups

For a multiple-value field within a periodic group, you can also define a `C*` variable with an index range specification.

The following examples use the multiple-value field `BONUS`, which is part of the periodic group `INCOME`. All three examples yield the same result.

Example 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
```

Example 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
```

Example 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
```

Caution: As the Adabas format buffer does not permit ranges for count fields, they are generated as individual fields; therefore a `C*` index range for a large array may cause an Adabas format buffer overflow.

Qualifying Data Structures

To identify a field when referencing it, you may qualify the field; that is, before the field name, you specify the name of the level-1 data element in which the field is located and a period.

If a field cannot be identified uniquely by its name (for example, if the same field name is used in multiple groups/views), you must qualify the field when you reference it.

The combination of level-1 data element and field name must be unique.

Example:

```
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
...
```

The qualifier must be a level-1 data element.

Example:

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

Qualifying a Database Field:

If you use the same name for a user-defined variable and a database field (which you should not do anyway), you must qualify the database field when you want to reference it

Caution: If you do not qualify the database field when you want to reference it, the user-defined variable will be referenced instead.

Examples of User-Defined Variables

```
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 ↩
```

15 Function Call

- Calling User-Defined Functions 108
- Function Result 109
- Evaluation Sequence 109
- Restrictions 109
- Syntax Description 110
- Example 114

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

Related topics:

- Object type **Function**
- **User-defined Functions**
- `DEFINE FUNCTION` statement
- `DEFINE PROTOTYPE` statement

Calling User-Defined Functions

A function call performs an invocation of a **user-defined function**, a special kind of a subroutine which is implemented in a separate programming object of type **function**.

Usually, a function call is provided with parameters and returns a result. It may be used within a Natural statement instead of a read-only operand. In this case, the function has to return a result, which is then processed by the statement like a field containing the same value.

It is also possible to use a function call stand-alone in place of a Natural statement. Then the function call need not return a result value.

There are different ways to call a function:

- Symbolic Function Call
- Variable Function Call

Symbolic Function Call

When the specified *call-name* represents the name of the function to be executed, the function call is denoted as a *symbolic* function call. Be aware, the function name is defined in the `DEFINE FUNCTION` statement and does not necessarily have to match the name of the module in which the function is defined. The function interface result and parameter layouts used to resolve a function call are either specified by a `DEFINE PROTOTYPE` statement with the same name or loaded automatically if the cataloged version of the called function exists.

Variable Function Call

A function can also be called in an indirect form, denoted as a *variable* function call. In this case, the specified *call-name* is an alphanumeric variable, which contains the name of the called function at execution time. This variable has to be referenced in a prototype definition (`DEFINE PROTOTYPE` statement) in conjunction with the keyword `VARIABLE`. If this prototype does not contain the right result layout or parameter layout, another prototype can be assigned with the `(PT=)` option.

Function Result

According to the function definition, a function call may return a single result field. This can be a scalar value or an array field, which is processed like a temporary field in the statement where the function call is embedded. If the result is an array, the function call must be immediately followed by an *array-index-expression* addressing the required occurrences.

For example, to access the first occurrence of the array returned:

```
#FCT(<#A,#B>)(1)
```

Evaluation Sequence

If a single or multiple function calls are used within a statement, the evaluation of all functions is performed in a separate step before the statement execution will be started. The function calls are executed in the same order in which they appear in the statement.

Restrictions

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

- in positions where the operand value is changed by the Natural statement (for example, `MOVE 1 TO #FCT(<..>)`);
- in a `DEFINE DATA` statement;
- in a database access statement (`READ`, `FIND`, `SELECT`, `UPDATE`, `STORE`, etc.);
- in an `AT BREAK` or `IF BREAK` statement;
- as an argument of Natural system functions (`AVER`, `SUM`, `*TRIM`, etc.);
- as an array index notation;
- as a parameter of a function call.

If a function call is used in an `INPUT` statement, the return value will be treated like a constant value. This leads to an automatic assignment of attribute (`AD=O`) to make this field write-protected (for output only).

Syntax Description

A function call may consist of the following syntax elements:

- call-name
- prototype-cast
- intermediate-result-definition
- Parameter(s)
- array-index-expression

call-name

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

Operand Definition Table:

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

Syntax Element Description:

Syntax Element	Description
function-name	**Function Name:** *function-name* is the name of the function to be called. Be aware, the function name is defined in the `DEFINE FUNCTION` statement and does not necessarily have to match the name of the module in which the function is defined. If a prototype with the same name has already been defined before, this prototype is used to pick up the function interface; that is, the result layout definition and the required parameters.
variable-name	**Variable Function Name:** *variable-name* is the name of the variable containing the real name of the function to be called at runtime. In order to declare the call name as a variable name, a prototype with the same name, which is classified with keyword `VARIABLE`, must be defined before.

prototype-cast

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

For every function call, Natural tries to get information on the function result and the calling parameters. This leads to a search for a prototype with the same name as the call name. If such a prototype is not available or if this prototype does not contain the right result layout or parameter layout, another prototype can be linked to the function call with a `(PT=)` clause. In this case, the referenced prototype steps in place and is used to define the function result layout and the parameter layout. The calling mode (***symbolic*** or ***variable***) declared in the referenced prototype is ignored.

Syntax Element Description:

Syntax Element	Description
prototype-name	**Prototype Name:** *prototype-name* is the identifier of the prototype whose result and parameters layouts are to be used.
prototype-variable-name	**Prototype Variable Name:** *prototype-variable-name* is the name of an alphanumeric field used as function name in a function call. At execution time it has to contain the name of the function to be called. The name has to follow the same rules which apply for a variable reference, including field qualification, but without array index references.

intermediate-result-definition

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

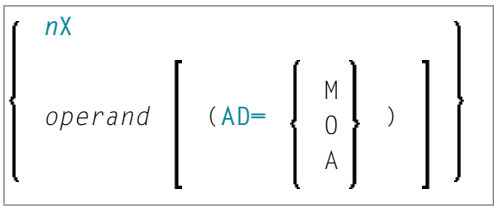
This clause can be used to specify the *format-length/array-definition* of the return value for a function call without using an explicit or implicit prototype definition, that is, it enables the explicit specification of the function result layout. If a prototype is available for this function call or if the cataloged version of the called function exists, the result format given in the `(IR=)` clause is checked for move compatibility.

Syntax Element Description:

Syntax Element	Description
format-length	**Format/Length Definition:** 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***.
array-definition	**Array Dimension 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.
HANDLE OF OBJECT	**Handle of Object:** Used in conjunction with NaturalX. For further information, see *NaturalX* in the *Programming Guide*.
A, B or U	**Data Format:** Possible formats are alphanumeric, binary or Unicode for dynamic variables.
DYNAMIC	**Dynamic Variable:** A field may be defined as DYNAMIC. For further information on processing dynamic variables, see ***Introduction to Dynamic Variables and Fields***.

Parameter(s)

Parameters are the data values passed to the function. They can be provided as a constant value or a variable, depending on what is defined in the DEFINE DATA PARAMETER section of the function definition. The semantic and syntactic rules which apply to the function parameters are the same as described in the parameters section of subprograms; see *Parameters* in the description of the CALLNAT statement.



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	no

Syntax Element Description:

Syntax Element	Description	
*n*X	**Parameters to be Skipped:** With the notation *n*X 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 function. A parameter that is to be skipped must be defined with the keyword OPTIONAL in the function's DEFINE DATA PARAMETER statement. OPTIONAL means that a value can - but need not - be passed from the invoking object to such a parameter.	
AD=	**Attribute Definition:** If *operand* is a variable, you can mark it in one of the following ways:	
	AD=O	**Non-modifiable:** See session parameter AD=O. **Note:** 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).
	AD=M	**Modifiable:** See session parameter AD=M. This is the default setting.
	AD=A	**Input only:** See session parameter AD=A.
	Note: If *operand* is a constant, the attribute definition AD cannot be explicitly specified. For constants, AD=O always applies.	

array-index-expression

If the result returned by the function call is an array, an index notation must be provided to address the demanded array occurrences.

For details, refer to ***Index Notation*** in *User-Defined Variables* in the *Programming Guide*.

Example

The example FUNCEX01 uses the functions F#ADDITION, F#CHAR, F#EVEN and F#TEXT, which are defined in the sample functions FUNCEX02, FUNCEX03, FUNCEX04 and FUNCEX05.

```
** Example 'FUNCEX01': Function call  (Program)
************************************************************************
DEFINE DATA LOCAL
  1 #NUM  (I2) INIT <5>
  1 #A    (I2) INIT <1>
  1 #B    (I2) INIT <2>
  1 #C    (I2) INIT <3>
  1 #CHAR (A1) INIT <'A'>
END-DEFINE
*
IF #NUM = F#ADDITION(<#A,#B,#C>)    /* Function with three parameters.
  WRITE 'Sum of #A,#B,#C' #NUM
ELSE
  IF #NUM = F#ADDITION(<1X,#B,#C>)  /* Function with optional parameters.
    WRITE 'Sum of #B,#C' #NUM
  END-IF
END-IF
*
DECIDE ON FIRST #CHAR
  VALUE F#CHAR (<>)(1)              /* Function with result array.
     WRITE 'Character A found'
  VALUE F#CHAR (<>)(2)
     WRITE 'Character B found'
  NONE
     IGNORE
END-DECIDE
*
IF F#EVEN(<#B>)                     /* Function with logical result value.
  WRITE #B 'is an even number'
END-IF
*
F#TEXT(<'Hello', '*'>)              /* Function used as a statement.
*
WRITE F#TEXT(<(IR=A12) 'Good'>)     /* Function with intermediate result.
*
END
```

The function F#ADDITION is defined in the sample function FUNCEX02 in library SYSEXPG.

```
** Example 'FUNCEX02': Function call  (Function)
************************************************************************
DEFINE FUNCTION F#ADDITION
  RETURNS (I2)
  DEFINE DATA PARAMETER
    1 #PARM1 (I2) OPTIONAL
    1 #PARM2 (I2) OPTIONAL
    1 #PARM3 (I2) OPTIONAL
  END-DEFINE
  /*
  RESET F#ADDITION
  IF #PARM1 SPECIFIED
    F#ADDITION := F#ADDITION + #PARM1                                    ↩

  END-IF
  IF #PARM2 SPECIFIED
    F#ADDITION := F#ADDITION + #PARM2                                    ↩

  END-IF
  IF #PARM3 SPECIFIED
    F#ADDITION := F#ADDITION + #PARM3                                    ↩

  END-IF
  /*
END-FUNCTION
*
END ↩
```

The function F#CHAR is defined in the example function FUNCEX03 in library SYSEXPG.

```
** Example 'FUNCEX03': Function call  (Function)
************************************************************************
DEFINE FUNCTION F#CHAR
  RETURNS (A1/1:2)
  /*
  F#CHAR(1) := 'A'
  F#CHAR(2) := 'B'
  /*
END-FUNCTION
*
END  ↩
```

The function F#EVEN is defined in the example function FUNCEX04 in library SYSEXPG.

```
** Example 'FUNCEX04': Function call  (Function)
************************************************************************
DEFINE FUNCTION F#EVEN
  RETURNS (L)
  DEFINE DATA
  PARAMETER
    1 #NUM  (N4) BY VALUE
  LOCAL
    1 #REST (I2)
  END-DEFINE
  /*
  DIVIDE 2 INTO #NUM REMAINDER #REST
  /*
  IF #REST = 0
    F#EVEN := TRUE
  ELSE
    F#EVEN := FALSE
  END-IF
  /*
END-FUNCTION
*
END ↩
```

The function F#TEXT is defined in the sample function FUNCEX05 in library SYSEXPG.

```
** Example 'FUNCEX05': Function call  (Function)
************************************************************************
DEFINE FUNCTION F#TEXT
  RETURNS (A20) BY VALUE
  DEFINE DATA
  PARAMETER
    1 #TEXT1 (A5) BY VALUE
    1 #TEXT2 (A1) BY VALUE OPTIONAL
  LOCAL
    1 #FRAME (A3)
  END-DEFINE
  /*
  IF #TEXT2 SPECIFIED
    MOVE ALL #TEXT2 TO #FRAME
    /*
    COMPRESS #FRAME #TEXT1 'world' #FRAME INTO F#TEXT
    /*
    WRITE F#TEXT
  ELSE
    COMPRESS #TEXT1 'morning' INTO F#TEXT
    /*
  END-IF
  /*
END-FUNCTION
*
END ↩
```

Output of Program FUNCEX01

```
Sum of #B,#C      5
Character A found
     2 is an even number
*** Hello world ***
Good morning
```

16 Introduction to Dynamic Variables and Fields

- Purpose of Dynamic Variables 120
- Definition of Dynamic Variables 120
- Value Space Currently Used for a Dynamic Variable 121
- Size Limitation Check 121
- Allocating/Freeing Memory Space for a Dynamic Variable 122

Purpose of Dynamic Variables

In that the maximum size of large data structures (for example, pictures, sounds, videos) may not exactly be known at application development time, Natural additionally provides for the definition of alphanumeric and binary variables with the attribute `DYNAMIC`. The value space of variables which are defined with this attribute will be extended dynamically at execution time when it becomes necessary (for example, during an assignment operation: `#picture1 := #picture2`). This means that large binary and alphanumeric data structures may be processed in Natural without the need to define a limit at development time. The execution-time allocation of dynamic variables is of course subject to available memory restrictions. If the allocation of dynamic variables results in an insufficent memory condition being returned by the underlying operating system, the `ON ERROR` statement can be used to intercept this error condition; otherwise, an error message will be returned by Natural.

The Natural system variable `*LENGTH` can be used obtain the length (in terms of code units) of the value space which is currently used for a given dynamic variable. For A and B formats, the size of one code unit is 1 byte. For U format, the size of one code unit is 2 bytes (UTF-16). Natural automatically sets `*LENGTH` to the length of the source operand during assignments in which the dynamic variable is involved. `*LENGTH(`*`field`*`)` therefore returns the length (in terms of code units) currently used for a dynamic Natural field or variable.

If the dynamic variable space is no longer needed, the `REDUCE` or `RESIZE` statements can be used to reduce the space used for the dynamic variable to zero (or any other desired size). If the upper limit of memory usage is known for a specific dynamic variable, the `EXPAND` statement can be used to set the space used for the dynamic variable to this specific size.

If a dynamic variable is to be initialized, the `MOVE ALL UNTIL` statement should be used for this purpose.

Definition of Dynamic Variables

Because the actual size of large alphanumeric and binary data structures may not be exactly known at application development time, the definition of *dynamic* variables of format A, B or U can be used to manage these structures. The dynamic allocation and extension (reallocation) of large variables is transparent to the application programming logic. Dynamic variables are defined without any length. Memory will be allocated either implicitly at execution time, when the dynamic variable is used as a target operand, or explicitly with an `EXPAND` or `RESIZE` statement.

Dynamic variables can only be defined in a `DEFINE DATA` statement using the following syntax:

```
level variable-name( A ) DYNAMIC
level variable-name( B ) DYNAMIC
level variable-name( U ) DYNAMIC
```

Restrictions:

The following restrictions apply to a dynamic variable:

- A redefinition of a dynamic variable is not allowed.
- A dynamic variable may not be contained in a `REDEFINE` clause.

Value Space Currently Used for a Dynamic Variable

The length (in terms of code units) of the currently used value space of a dynamic variable can be obtained from the system variable `*LENGTH`. `*LENGTH` is set to the (used) length of the source operand during assignments automatically.

Caution: Due to performance considerations, the storage area that is allocated to hold the value of the dynamic variable may be larger than the value of `*LENGTH` (used size available to the programmer). You should not rely on the storage that is allocated beyond the used length as indicated by `*LENGTH`: it may be released at any time, even if the respective dynamic variable is not accessed. It is not possible for the Natural programmer to obtain information about the currently allocated size. This is an internal value.

`*LENGTH(`*`field`*`)` returns the used length (in terms of code units) of a dynamic Natural field or variable. For A and B formats, the size of one code unit is 1 byte. For U format, the size of one code unit is 2 bytes (UTF-16). `*LENGTH` may be used only to get the currently used length for dynamic variables.

Size Limitation Check

Profile Parameter USIZE

For dynamic variables, a size limitation check at compile time is not possible because no length is defined for dynamic variables. The size of user buffer area (`USIZE`) indicates the size of the user buffer in virtual memory. The user buffer contains all data dynamically allocated by Natural. If a dynamic variable is allocated or extended at execution time and the `USIZE` limitation is exceeded, an error message will be returned.

Allocating/Freeing Memory Space for a Dynamic Variable

The statements EXPAND, REDUCE and RESIZE are used to explicitly allocate and free memory space for a dynamic variable.

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
```

- where *operand1* is a dynamic variable and *operand2* is a non-negative numeric size value.

EXPAND

Function

The EXPAND statement is used to increase the allocated length of the dynamic variable (*operand1*) to the specified length (*operand2*).

Changing the Specified Size

The length currently used (as indicated by the Natural system variable *LENGTH, see **above**) for the dynamic variable is not modified.

If the specified length (*operand2*) is less than the allocated length of the dynamic variable, the statement will be ignored.

REDUCE

Function

The REDUCE statement is used to reduce the allocated length of the dynamic variable (*operand1*) to the specified length (*operand2*).

The storage allocated for the dynamic variable (*operand1*) beyond the specified length (*operand2*) may be released at any time, when the statement is executed or at a later time.

Changing the Specified Length

If the length currently used (as indicated by the Natural system variable *LENGTH, see **above**) for the dynamic variable is greater than the specified length (*operand2*), the system variable *LENGTH of this dynamic variable is set to the specified length. The content of the variable is truncated, but not modified.

If the given length is larger than the currently allocated storage of the dynamic variable, the statement will be ignored.

RESIZE

Function

The RESIZE statement adjusts the currently allocated length of the dynamic variable (*operand1*) to the specified length (*operand2*).

Changing the Specified Length

If the specified length is smaller then the used length (as indicated by the Natural system variable *LENGTH, see **above**) of the dynamic variable, the used length is reduced accordingly.

If the specified length is larger than the currently allocated length of the dynamic variable, the allocated length of the dynamic variable is increased. The currently used length (as indicated by the system variable *LENGTH) of the dynamic variable is not affected and remains unchanged.

If the specified length is the same as the currently allocated length of the dynamic variable, the execution of the RESIZE statement has no effect.

17 Using Dynamic and Large Variables

- General Remarks 126
- Assignments with Dynamic Variables 127
- Initialization of Dynamic Variables 129
- String Manipulation with Dynamic Alphanumeric Variables 129
- Logical Condition Criterion (LCC) with Dynamic Variables 130
- AT/IF-BREAK of Dynamic Control Fields 132
- Parameter Transfer with Dynamic Variables 132
- Work File Access with Large and Dynamic Variables 135
- DDM Generation and Editing for Varying Length Columns 136
- Accessing Large Database Objects 138
- Performance Aspects with Dynamic Variables 139
- Outputting Dynamic Variables 140
- Dynamic X-Arrays 141

General Remarks

Generally, the following rules apply:

- A dynamic alphanumeric field may be used wherever an alphanumeric field is allowed.
- A dynamic binary field may be used wherever a binary field is allowed.
- A dynamic Unicode field may be used wherever a Unicode field is allowed.

Exception:

Dynamic variables are not allowed within the `SORT` statement. To use dynamic variables in a `DISPLAY`, `WRITE`, `PRINT`, `REINPUT` or `INPUT` statement, you must use either the session parameter `AL` or `EM` to define the length of the variable.

The used length (as indicated by the Natural system variable `*LENGTH`, see ***Value Space Currently Used for a Dynamic Variable***) and the size of the allocated storage of dynamic variables are equal to zero until the variable is accessed as a target operand for the first time. Due to assignments or other manipulation operations, dynamic variables may be firstly allocated or extended (reallocated) to the exact size of the source operand.

The size of a dynamic variable may be extended if it is used as a modifiable operand (target operand) in the following statements:

`ASSIGN`	*`operand1`* (destination operand in an assignment).
`CALLNAT`	See *Parameter Transfer with Dynamic Variables* (except if `AD=O`, or if `BY VALUE` exists in the corresponding parameter data area).
`COMPRESS`	*`operand2`*, see *Processing*.
`EXAMINE`	*`operand1`* in the `DELETE REPLACE` clause.
`MOVE`	*`operand2`* (destination operand), see *Function*.
`PERFORM`	(except if `AD=O`, or if `BY VALUE` exists in the corresponding parameter data area).
`READ WORK FILE`	*`operand1`* and *`operand2`*, see *Handling of Large and Dynamic Variables*.
`SEPARATE`	*`operand4`*.
`SELECT` (SQL)	*`parameter`* in the *INTO Clause*.
`SEND METHOD`	*`operand3`* (except if `AD=O`).

Currently, there is the following limit concerning the usage of large variables:

CALL	Parameter size less than 64 KB per parameter (no limit for CALL with INTERFACE4 option).

In the following sections, the use of dynamic variables is discussed in more detail on the basis of examples.

Assignments with Dynamic Variables

Generally, an assignment is done in the current used length (as indicated by the Natural system variable *LENGTH) of the source operand. If the destination operand is a dynamic variable, its current allocated size is possibly extended in order to move the source operand without truncation.

Example:

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

MOVE ALL, MOVE ALL UNTIL with dynamic target operands are defined as follows:

- MOVE ALL moves the source operand repeatedly to the target operand until the used length (*LENGTH) of the target operand is reached. The system variable *LENGTH is not modified. If *LENGTH is zero, the statement will be ignored.
- MOVE ALL *operand1* TO *operand2* UNTIL operand3 moves *operand1* repeatedly to *operand2* until the length specified in *operand3* is reached. If *operand3* is greater than *LENGTH(*operand2*), *operand2* is extended and *LENGTH(*operand2*) is set to *operand3*. If *operand3* is less than *LENGTH(*operand2*), the used length is reduced to *operand3*. If *operand3* equals *LENGTH(*operand2*), the behavior is equivalent to MOVE ALL.

Example:

```
#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 is rejected at compile time if the target operand is a dynamic variable.

MOVE SUBSTR and MOVE TO SUBSTR are allowed. MOVE SUBSTR will lead to a runtime error if a sub-string behind the used length of a dynamic variable (*LENGTH) is referenced. MOVE TO SUBSTR will lead to a runtime error if a sub-string position behind *LENGTH + 1 is referenced, because this would lead to an undefined gap in the content of the dynamic variable. If the target operand

should be extended by `MOVE TO SUBSTR` (for example if the second operand is set to `*LENGTH+1`), the third operand is mandatory.

Valid 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          ↩
```

Invalid 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
```

Assignment Compatibility

Example:

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

If the source operand is a static variable, the used length of the dynamic destination operand (`*LENGTH(#MYDYNTEXT1)`) is set to the format length of the static variable and the source value is copied in this length including trailing blanks (alphanumeric and Unicode fields) or binary zeros (for binary fields).

If the destination operand is static and the source operand is dynamic, the dynamic variable is copied in its currently used length. If this length is less than the format length of the static variable, the remainder is filled with blanks (for alphanumeric and Unicode fields) or binary zeros (for binary fields). Otherwise, the value will be truncated. If the currently used length of the dynamic variable is 0, the static target operand is filled with blanks (for alphanumeric and Unicode fields) or binary zeros (for binary fields).

Initialization of Dynamic Variables

Dynamic variables can be initialized with blanks (alphanumeric and Unicode fields) or zeros (binary fields) up to the currently used length (= *LENGTH) using the RESET statement. The system variable *LENGTH is not modified.

Example:

```
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  ↩
```

To initialize a dynamic variable with a specified value in a specified size, the MOVE ALL UNTIL statement may be used.

Example:

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

String Manipulation with Dynamic Alphanumeric Variables

If a modifiable operand is a dynamic variable, its current allocated size is possibly extended in order to perform the operation without truncation or an error message. This is valid for the concatenation (COMPRESS) and separation of dynamic alphanumeric variables (SEPARATE).

Example:

```
** 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
```

Note: In case of non-dynamic variables, an error message may be returned.

Logical Condition Criterion (LCC) with Dynamic Variables

Generally, a read-only operation (such as a comparison) with a dynamic variable is done with its currently used length. Dynamic variables are processed like static variables if they are used in a read-only (non-modifiable) context.

Example:

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

Trailing blanks for alphanumeric and Unicode variables or leading binary zeros for binary variables are processed in the same way for static and dynamic variables. For example, alphanumeric variables containing the values `AA` and `AA` followed by a blank will be considered being equal, and binary variables containing the values `H'0000031'` and `H'3031'` will be considered being equal. If a comparison result should only be `TRUE` in case of an exact copy, the used lengths of the dynamic variables have to be compared in addition. If one variable is an exact copy of the other, their used lengths are also equal.

Example:

```
#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
```

Two dynamic variables are compared position by position (from left to right for alphanumeric variables, and right to left for binary variables) up to the minimum of their used lengths. The first position where the variables are not equal determines if the first or the second variable is greater than, less than or equal to the other. The variables are equal if they are equal up to the minimum of their used lengths and the remainder of the longer variable contains only blanks for alphanumeric dynamic variables or binary zeros for binary dynamic variables. To compare two Unicode dynamic variables, trailing blanks are removed from both values before the ICU collation algorithm is used to compare the two resulting values. See also *Logical Condition Criteria* in the *Unicode and Code Page Support* documentation.

Example:

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

Comparison Compatibility

Comparisons between dynamic and static variables are equivalent to comparisons between dynamic variables. The format length of the static variable is interpreted as its used length.

Example:

```
#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 of Dynamic Control Fields

The comparison of the break control field with its old value is performed position by position from left to right. If the old and the new value of the dynamic variable are of different length, then for comparison, the value with shorter length is padded to the right (with blanks for alphanumeric and Unicode dynamic values or binary zeros for binary values).

In case of an alphanumeric or Unicode break control field, trailing blanks are not significant for the comparison, that is, trailing blanks do not mean a change of the value and no break occurs.

In case of a binary break control field, trailing binary zeros are not significant for the comparison, that is, trailing binary zeros do not mean a change of the value and no break occurs.

Parameter Transfer with Dynamic Variables

Dynamic variables may be passed as parameters to a called program object (`CALLNAT`, `PERFORM`). A call-by-reference is possible because the value space of a dynamic variable is contiguous. A call-by-value causes an assignment with the variable definition of the caller as the source operand and the parameter definition as the destination operand. A call-by-value result causes in addition the movement in the opposite direction.

For a call-by-reference, both definitions must be `DYNAMIC`. If only one of them is `DYNAMIC`, a runtime error is raised. In the case of a call-by-value (result), all combinations are possible. The following table illustrates the valid combinations:

Call By Reference

Caller	Parameter	
	Static	Dynamic
Static	Yes	No
Dynamic	No	Yes

The formats of dynamic variables A or B must match.

Call by Value (Result)

Caller	Parameter	
	Static	Dynamic
Static	Yes	Yes
Dynamic	Yes	Yes

Note: In the case of static/dynamic or dynamic/static definitions, a value truncation may occur according to the data transfer rules of the appropriate assignments.

Example 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                                                                    ↩
```

Subprogram `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                                                                ↩
```

Example 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                                                              ↩
```

Subprogram `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                                                                ↩
```

CALL 3GL Program

Dynamic and large variables can sensibly be used with the `CALL` statement when the option `INTERFACE4` is used. Using this option leads to an interface to the 3GL program with a different parameter structure.

This usage requires some minor changes in the 3GL program, but provides the following significant benefits as compared with the older `FINFO` structure.

- No limitation on the number of passed parameters (former limit 40).
- No limitation on the parameter's data size (former limit 64 KB per parameter).

- Full parameter information can be passed to the 3GL program including array information. Exported functions are provided which allow secure access to the parameter data (formerly you had to take care not to overwrite memory inside of Natural)

For further information on the FINFO structure, see the CALL INTERFACE4 statement.

Before calling a 3GL program with dynamic parameters, it is important to ensure that the necessary buffer size is allocated. This can be done explicitly with the EXPAND statement.

If an initialized buffer is required, the dynamic variable can be set to the initial value and to the necessary size by using the MOVE ALL UNTIL statement. Natural provides a set of functions that allow the 3GL program to obtain information about the dynamic parameter and to modify the length when parameter data is passed back.

Example:

```
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
```

For a more detailed description, refer to the CALL statement in the *Statements* documentation.

Work File Access with Large and Dynamic Variables

The following topics are covered below:

- PORTABLE and UNFORMATTED
- ASCII, ASCII-COMPRESSED and SAG
- Special Conditions for TRANSFER and ENTIRE CONNECTION

PORTABLE and UNFORMATTED

Large and dynamic variables can be written into work files or read from work files using the two work file types PORTABLE and UNFORMATTED. For these types, there is no size restriction for dynamic variables. However, large variables may not exceed a maximum field/record length of 32766 bytes.

For the work file type PORTABLE, the field information is stored within the work file. The dynamic variables are resized during READ if the field size in the record is different from the current size.

The work file type UNFORMATTED can be used, for example, to read a video from a database and store it in a file directly playable by other utilities. In the WRITE WORK statement, the fields are written to the file with their byte length. All data types (DYNAMIC or not) are treated the same. No

structural information is inserted. Note that Natural uses a buffering mechanism, so you can expect the data to be completely written only after a CLOSE WORK. This is especially important if the file is to be processed with another utility while Natural is running.

With the READ WORK statement, fields of fixed length are read with their whole length. If the end-of-file is reached, the remainder of the current field is filled with blanks. The following fields are unchanged. In the case of DYNAMIC data types, all the remainder of the file is read unless it exceeds 1073741824 bytes. If the end of file is reached, the remaining fields (variables) are kept unchanged (normal Natural behavior).

ASCII, ASCII-COMPRESSED and SAG

The work file types ASCII, ASCII-COMPRESSED and SAG (binary) cannot handle dynamic variables and will produce an error. Large variables for these work file types pose no problem unless the maximum field/record length of 32766 bytes is exceeded.

Special Conditions for TRANSFER and ENTIRE CONNECTION

In conjunction with the READ WORK FILE statement, the work file type TRANSFER can handle dynamic variables. There is no size limit for dynamic variables. The work file type ENTIRE CONNECTION cannot handle dynamic variables. They can both, however, handle large variables with a maximum field/record length of 1073741824 bytes.

In conjunction with the WRITE WORK FILE statement, the work file type TRANSFER can handle dynamic variables with a maximum field/record length of 32766 bytes. The work file type ENTIRE CONNECTION cannot handle dynamic variables. They can both, however, handle large variables with a maximum field/record length of 1073741824 bytes.

DDM Generation and Editing for Varying Length Columns

Depending on the data types, the related database format A or format B is generated. For the databases' data type VARCHAR the Natural length of the column is set to the maximum length of the data type as defined in the DBMS. If a data type is very large, the keyword DYNAMIC is generated at the length field position.

For all varying length columns, an LINDICATOR field L@<*column-name*> will be generated. For the databases' data type VARCHAR, an LINDICATOR field with format/length I2 will be generated. For large data types (see list below) the format/length will be I4.

In the context of database access, the LINDICATOR handling offers the chance to get the length of the field to be read or to set the length of the field to be written independent of a defined buffer length (or independent of *LENGTH). Usually, after a retrieval function, *LENGTH will be set to the corresponding length indicator value.

Example 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
```

The generated formats are varying length formats. The Natural programmer has the chance to change the definition from `DYNAMIC` to a fixed length definition (extended field editing) and can change, for example, the corresponding DDM field definition for `VARCHAR` data types to a multiple value field (old generation).

Example:

```
  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
```

Accessing Large Database Objects

To access a database with large objects (CLOBs or BLOBs), a DDM with corresponding large alphanumeric, Unicode or binary fields is required. If a fixed length is defined and if the database large object does not fit into this field, the large object is truncated. If the programmer does not know the definitive length of the database object, it will make sense to work with dynamic fields. As many reallocations as necessary are done to hold the object. No truncation is performed.

Example Program:

```
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 ↩
```

If a format-length definition is omitted in the view, this is taken from the DDM. In reporting mode, it is now possible to specify any length, if the corresponding DDM field is defined as `DYNAMIC`. The dynamic field will be mapped to a field with a fixed buffer length. The other way round is not possible.

DDM format/length definition	VIEW format / length definition	
(An)	-	valid
	(An)	valid
	(Am)	only valid in reporting mode
	(A) DYNAMIC	invalid
(A) DYNAMIC	-	valid
	(A) DYNAMIC	valid
	(An)	only valid in reporting mode
	(Am / i : j)	only valid in reporting mode

(equivalent for Format B variables)

Parameter with LINDICATOR Clause in SQL Statements

If the `LINDICATOR` field is defined as an I2 field, the SQL data type `VARCHAR` is used for sending or receiving the corresponding column. If the `LINDICATOR` host variable is specified as I4, a large object data type (CLOB/BLOB) is used.

If the field is defined as `DYNAMIC`, the column is read in an internal loop up to its real length. The `LINDICATOR` field and the system variable `*LENGTH` are set to this length. In the case of a fixed-length field, the column is read up to the defined length. In both cases, the field is written up to the value defined in the `LINDICATOR` field.

Performance Aspects with Dynamic Variables

If a dynamic variable is to be expanded in small quantities multiple times (for example, byte-wise), use the `EXPAND` statement before the iterations if the upper limit of required storage is (approximately) known. This avoids additional overhead to adjust the storage needed.

Use the `REDUCE` or `RESIZE` statement if the dynamic variable will no longer be needed, especially for variables with a high value of the system variable `*LENGTH`. This enables Natural you to release or reuse the storage. Thus, the overall performance may be improved.

The amount of the allocated memory of a dynamic variable may be reduced using the `REDUCE DYNAMIC VARIABLE` statement. In order to (re)allocate a variable to a specified length, the `EXPAND` statement can be used. (If the variable should be initialized, use the `MOVE ALL UNTIL` statement.)

Example:

```
** 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
```

Rules:

- Use dynamic operands where it makes sense.
- Use the `EXPAND` statement if upper limit of memory usage is known.
- Use the `REDUCE` statement if the dynamic operand will no longer be needed.

Outputting Dynamic Variables

Dynamic variables may be used inside output statements such as the following:

Statement	Notes
DISPLAY	With these statements, you must set the format of the output or input of dynamic variables using the AL (Alphanumeric Length for Output) or EM (Edit Mask) session parameters.
WRITE	
INPUT	
REINPUT	--
PRINT	Because the output of the PRINT statement is unformatted, the output of dynamic variables in the PRINT statement need not be set using AL and EM parameters. In other words, these parameters may be omitted.

Dynamic X-Arrays

A dynamic X-array may be allocated by first specifying the number of occurrences and then expanding the length of the previously allocated array occurrences.

Example:

```
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'
```

18 User-Defined Constants

- Numeric Constants 144
- Alphanumeric Constants 145
- Unicode Constants 146
- Date and Time Constants 149
- Hexadecimal Constants 150
- Logical Constants 152
- Floating Point Constants 152
- Attribute Constants 153
- Handle Constants 154
- Defining Named Constants 154

Constants can be used throughout Natural programs. This document discusses the types of constants that are supported and how they are used.

Numeric Constants

The following topics are covered below:

- Numeric Constants
- Validation of Numeric Constants

Numeric Constants

A numeric constant may contain 1 to 29 numeric digits, a special character as decimal separator (period or comma) and a sign.

Examples:

```
1234   +1234   -1234

12.34  +12.34  -12.34
```

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

Note: Internally, numeric constants without decimal digits are represented in integer form (format I), while numeric constants with decimal digits, as well as numeric constants without decimal digits that are too large to fit into format I, are represented in packed form (format P).

Example:

Numeric Constant		Format	Length
From	To		
	<= -2147483649	P	>=10
-2147483648	-32769	I	4
-32768	32767	I	2
32768	2147483647	I	4
>= 2147483648		P	>=10

Validation of Numeric Constants

When numeric constants are used within one of the statements `COMPUTE`, `MOVE`, or `DEFINE DATA` with `INIT` option, Natural checks at compilation time whether a constant value fits into the corresponding field. This avoids runtime errors in situations where such an error condition can already be detected during compilation.

Alphanumeric Constants

The following topics are covered below:

- Alphanumeric Constants
- Apostrophes Within Alphanumeric Constants
- Concatenation of Alphanumeric Constants

Alphanumeric Constants

An alphanumeric constant may contain 1 to 1 1073741824 bytes (1 GB) of alphanumeric characters.

An alphanumeric constant must be enclosed in either apostrophes (')

```
'text'
```

or quotation marks (")

```
"text"
```

Examples:

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

Note: An alphanumeric constant that is used to assign a value to a **user-defined variable** must not be split between statement lines.

Apostrophes Within Alphanumeric Constants

If you want an apostrophe to be part of an alphanumeric constant that is enclosed in apostrophes, you must write this as two apostrophes or as a single quotation mark.

If you want an apostrophe to be part of an alphanumeric constant that is enclosed in quotation marks, you write this as a single apostrophe.

Example:

If you want the following to be output:

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

you can use any of the following notations:

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

Note: If quotation marks are not converted to apostrophes as shown above, this is due to the setting of profile parameter `TQMARK` (Translate Quotation Marks); ask your Natural administrator for details.

Concatenation of Alphanumeric Constants

Alphanumeric constants may be concatenated to form a single value by use of a hyphen.

Examples:

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

In this way, alphanumeric constants can also be concatenated with **hexadecimal constants**.

Unicode Constants

The following topics are covered below:

- Unicode Text Constants
- Apostrophes Within Unicode Text Constants
- Unicode Hexadecimal Constants

- Concatenation of Unicode Constants

Unicode Text Constants

A Unicode text constant must be preceded by the character `U` and enclosed in either apostrophes (')

```
U'text'
```

or quotation marks (")

```
U"text"
```

Example:

```
U'HELLO'
```

The compiler stores this text constant in the generated program in Unicode format (UTF-16).

Apostrophes Within Unicode Text Constants

If you want an apostrophe to be part of a Unicode text constant that is enclosed in apostrophes, you must write this as two apostrophes or as a single quotation mark.

If you want an apostrophe to be part of a Unicode text constant that is enclosed in quotation marks, you write this as a single apostrophe.

Example:

If you want the following to be output:

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

you can use any of the following notations:

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

Note: If quotation marks are not converted to apostrophes as shown above, this is due to the setting of the profile parameter `TQ` (Translate Quotation Marks); ask your Natural administrator for details.

Unicode Hexadecimal Constants

The following syntax is used to supply a Unicode character or a Unicode string by its hexadecimal notation:

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

where *h* represents a hexadecimal digit (0-9, A-F). Since a UTF-16 Unicode character consists of a double-byte, the number of hexadecimal characters supplied has to be a multiple of four.

Example:

This example defines the string `45`.

```
UH'00340035'
```

Concatenation of Unicode Constants

Concatenation of Unicode text constants (U) and Unicode hexadecimal constants (UH) is allowed.

Valid Example:

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

Unicode text constants or Unicode hexadecimal constants cannot be concatenated with code page alphanumeric constants or H constants.

Invalid Example:

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

Further Valid Example:

```
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
```

Date and Time Constants

The following topics are covered below:

- Date Constant
- Time Constant
- Extended Time Constant

Date Constant

A date constant may be used in conjunction with a format D variable.

Date constants may have the following formats:

D'*yyyy-mm-dd*'	International date format
D'*dd.mm.yyyy*'	German date format
D'*dd/mm/yyyy*'	European date format
D'*mm/dd/yyyy*'	US date format

where *dd* represents the number of the day, *mm* the number of the month and *yyyy* the year.

Example:

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

The default date format is controlled by the profile parameter DTFORM (Date Format) as set by the Natural administrator.

Time Constant

A time constant may be used in conjunction with a format T variable.

A time constant has the following format:

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

where *hh* represents hours, *ii* minutes and *ss* seconds.

Example:

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

Extended Time Constant

A time variable (format T) can contain date and time information, date information being a subset of time information; however, with a "normal" time constant (prefix `T`) only the time information of a time variable can be handled:

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

With an extended time constant (prefix `E`), it is possible to handle the full content of a time variable, including the date information:

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

Apart from that, the use of an extended time constant in conjunction with a time variable is the same as for a normal time constant.

Note: The format in which the date information has to be specified in an extended time constant depends on the setting of the profile parameter `DTFORM`. The extended time constant shown above assumes `DTFORM=I` (international date format).

Hexadecimal Constants

The following topics are covered below:

- Hexadecimal Constants

- Concatenation of Hexadecimal Constants

Hexadecimal Constants

A hexadecimal constant may be used to enter a value which cannot be entered as a standard keyboard character.

A hexadecimal constant may contain 1 to 1073741824 bytes (1 GB) of alphanumeric characters.

A hexadecimal constant is prefixed with an `H`. The constant itself must be enclosed in apostrophes and may consist of the hexadecimal characters 0 - 9, A - F. Two hexadecimal characters are required to represent one byte of data.

The hexadecimal representation of a character varies, depending on whether your computer uses an ASCII or EBCDIC character set. When you transfer hexadecimal constants to another computer, you may therefore have to convert the characters.

ASCII examples:

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

EBCDIC examples:

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

When a hexadecimal constant is transferred to another field, it will be treated as an alphanumeric value (format A).

The data transfer of an alphanumeric value (format A) to a field which is defined with a format other than A,U or B is not allowed. Therefore, a hexadecimal constant used as initial value in a `DEFINE DATA` statement is rejected with the syntax error NAT0094 if the corresponding variable is not of format A, U or B.

Example:

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

Concatenation of Hexadecimal Constants

Hexadecimal constants may be concatenated by using a hyphen between the constants.

ASCII example:

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

EBCDIC example:

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

In this way, hexadecimal constants can also be concatenated with alphanumeric constants.

Logical Constants

The logical constants `TRUE` and `FALSE` may be used to assign a logical value to a field defined with Format L.

Example:

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

Floating Point Constants

Floating point constants can be used with variables defined with format F.

Example:

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

Attribute Constants

Attribute constants can be used with variables defined with format C (control variables). This type of constant must be enclosed within parentheses.

The following attributes may be used:

Attribute	Description
AD=D	default
AD=B	blinking
AD=I	intensified
AD=N	non-display
AD=V	reverse video
AD=U	underlined
AD=C	cursive/italic
AD=Y	dynamic attribute
AD=P	protected
CD=BL	blue
CD=GR	green
CD=NE	neutral
CD=PI	pink
CD=RE	red
CD=TU	turquoise
CD=YE	yellow

See also session parameters AD and CD.

Example:

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

Handle Constants

The handle constant `NULL-HANDLE` can be used with GUI handles and object handles.

For further information on GUI handles, see ***How To Define Dialog Elements***.

For further information on object handles, see the section ***NaturalX***.

Defining Named Constants

If you need to use the same constant value several times in a program, you can reduce the maintenance effort by defining a named constant:

- Define a field in the `DEFINE DATA` statement,
- assign a constant value to it, and
- use the field name in the program instead of the constant value.

Thus, when the value has to be changed, you only have to change it once in the `DEFINE DATA` statement and not everywhere in the program where it occurs.

You specify the constant value in angle brackets with the **keyword** `CONSTANT` after the field definition in the `DEFINE DATA` statement.

- If the value is alphanumeric, it must be enclosed in apostrophes.
- If the value is text in Unicode format, it must be preceded by the character `U` and must be enclosed in apostrophes.
- If the value is in hexadecimal Unicode format, it must be preceded by the characters `UH` and must be enclosed in apostrophes.

Example:

```
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
...
```

During the execution of the program, the value of such a named constant cannot be modified.

19 Initial Values (and the RESET Statement)

- Default Initial Value of a User-Defined Variable/Array 158
- Assigning an Initial Value to a User-Defined Variable/Array 158
- Resetting a User-Defined Variable to its Initial Value 160

This chapter describes the default initial values of user-defined variables, explains how you can assign an initial value to a user-defined variable and how you can use the RESET statement to reset the field value to its default initial value or the initial value defined for that variable in the DEFINE DATA statement.

Default Initial Value of a User-Defined Variable/Array

If you specify no initial value for a field, the field will be initialized with a default initial value depending on its format:

Format	Default Initial Value
B, F, I, N, P	0
A, U	*blank*
L	F(ALSE)
D	D' '
T	T'00:00:00'
C	(AD=D)
GUI Handle	NULL-HANDLE
Object Handle	NULL-HANDLE

Assigning an Initial Value to a User-Defined Variable/Array

In the DEFINE DATA statement, you can assign an initial value to a user-defined variable. If the initial value is alphanumeric, it must be enclosed in apostrophes.

- Assigning a Modifiable Initial Value
- Assigning a Constant Initial Value
- Assigning a Natural System Variable as Initial Value

- Assigning Characters as Initial Value for Alphanumeric Variables

Assigning a Modifiable Initial Value

If the variable/array is to be assigned a modifiable initial value, you specify the initial value in angle brackets with the keyword `INIT` after the variable definition in the `DEFINE DATA` statement. The value(s) assigned will be used each time the variable/array is referenced. The value(s) assigned can be modified during program execution.

Example:

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

Assigning a Constant Initial Value

If the variable/array is to be treated as a named constant, you specify the initial value in angle brackets with the keyword `CONSTANT` after the variable definition in the `DEFINE DATA` statement. The constant value(s) assigned will be used each time the variable/array is referenced. The value(s) assigned cannot be modified during program execution.

Example:

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

Assigning a Natural System Variable as Initial Value

The initial value for a field may also be the value of a **Natural system variable**.

Example:

In this example, the system variable `*DATX` is used to provide the initial value.

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

Assigning Characters as Initial Value for Alphanumeric Variables

As initial value, a variable can also be filled, entirely or partially, with a specific single character or string of characters (only possible for alphanumeric variables).

- **Filling an entire field:**
  With the option `FULL LENGTH <character(s)>`, the entire field is filled with the specified character(s).

  In this example, the entire field will be filled with asterisks.

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

- **Filling the first*n* positions of a field:**
  With the option `LENGTH n <character(s)>`, the first *n* positions of the field are filled with the specified character(s).

  In this example, the first 4 positions of the field will be filled with exclamation marks.

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

Resetting a User-Defined Variable to its Initial Value

The `RESET` statement is used to reset the value of a field. Two options are available:

- Reset to Default Initial Value
- Reset to Initial Value Defined in DEFINE DATA

Notes:

1. A field declared with a `CONSTANT` clause in the `DEFINE DATA` statement may not be referenced in a `RESET` statement, since its content cannot be changed.

2. In reporting mode, the RESET statement may also be used to define a variable, provided that the program contains no DEFINE DATA LOCAL statement.

Reset to Default Initial Value

RESET (without INITIAL) sets the content of each specified field to its **default initial value** depending on its format.

Example:

```
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
... ↩
```

Reset to Initial Value Defined in DEFINE DATA

RESET INITIAL sets each specified field to the initial value as defined for the field in the DEFINE DATA statement.

For a field declared without INIT clause in the DEFINE DATA statement, RESET INITIAL has the same effect as RESET (without INITIAL).

Example:

```
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
...
```

20 Redefining Fields

- Using the REDEFINE Option of DEFINE DATA 164
- Example Program Illustrating the Use of a Redefinition 165

Redefinition is used to change the format of a field, or to divide a single field into segments.

Using the REDEFINE Option of DEFINE DATA

The REDEFINE option of the DEFINE DATA statement can be used to redefine a single field - either a user-defined variable or a database field - as one or more new fields. A group can also be re-defined.

Important: Dynamic variables are not allowed in a redefinition.

The REDEFINE option redefines byte positions of a field from left to right, regardless of the format. Byte positions must match between original field and redefined field(s).

The redefinition must be specified immediately after the definition of the original field.

Example 1:

In the following example, the database field BIRTH is redefined as three new user-defined variables:

```
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
   ...
```

Example 2:

In the following example, the group #VAR2, which consists of two user-defined variables of format N and P respectively, is redefined as a variable of format A:

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

With the notation FILLER *n*X you can define *n* filler bytes - that is, segments which are not to be used - in the field that is being redefined. (The definition of trailing filler bytes is optional.)

Example 3:

In the following example, the user-defined variable #FIELD is redefined as three new user-defined variables, each of format/length A2. The FILLER notations indicate that the 3rd and 4th and 7th to 10th bytes of the original field are not be used.

```
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
...
```

Example Program Illustrating the Use of a Redefinition

The following program illustrates the use of a 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 #000     (N3)
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  MOVE SALARY (1) TO #PAY
  DISPLAY NAME FIRST-NAME #PAY #USD #000
END-READ
END
```

Output of Program DDATAX01:

Note how #PAY and the fields resulting from its definition are displayed:

```
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
            ↩
```

21 Arrays

- Defining Arrays 168
- Initial Values for Arrays 169
- Assigning Initial Values to One-Dimensional Arrays 169
- Assigning Initial Values to Two-Dimensional Arrays 170
- A Three-Dimensional Array 174
- Arrays as Part of a Larger Data Structure 176
- Database Arrays 177
- Using Arithmetic Expressions in Index Notation 177
- Arithmetic Support for Arrays 178

Natural supports the processing of arrays. Arrays are multi-dimensional tables, that is, two or more logically related elements identified under a single name. Arrays can consist of single data elements of multiple dimensions or hierarchical data structures which contain repetitive structures or individual elements.

Defining Arrays

In Natural, an array can be one-, two- or three-dimensional. It can be an independent variable, part of a larger data structure or part of a database view.

Important: Dynamic variables are not allowed in an array definition.

To define a one-dimensional array

- After the format and length, specify a slash followed by a so-called “index notation”, that is, the number of occurrences of the array.

  For example, the following one-dimensional array has three occurrences, each occurrence being of format/length A10:

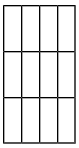
```
DEFINE DATA LOCAL
1 #ARRAY (A10/1:3)
END-DEFINE
...
```

To define a two-dimensional array

- Specify an index notation for both dimensions:

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

  A two-dimensional array can be visualized as a table. The array defined in the example above would be a table that consists of 3 “rows” and 4 “columns”:



Initial Values for Arrays

To assign initial values to one or more occurrences of an array, you use an `INIT` specification, similar to that for **"ordinary" variables**, as shown in the following examples.

Assigning Initial Values to One-Dimensional Arrays

The following examples illustrate how initial values are assigned to a one-dimensional array.

- To assign an initial value to one occurrence, you specify:

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

  `A` is assigned to the second occurrence.

- To assign the same initial value to all occurrences, you specify:

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

  `A` is assigned to every occurrence. Alternatively, you could specify:

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

- To assign the same initial value to a range of several occurrences, you specify:

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

  `A` is assigned to the second to third occurrence.

- To assign a different initial value to every occurrence, you specify:

```
1 #ARRAY (A1/1:3) INIT <'A','B','C'>
```

`A` is assigned to the first occurrence, `B` to the second, and `C` to the third.

- To assign different initial values to some (but not all) occurrences, you specify:

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

`A` is assigned to the first occurrence, and `C` to the third; no value is assigned to the second occurrence.

Alternatively, you could specify:

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

- If fewer initial values are specified than there are occurrences, the last occurrences remain empty:

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

`A` is assigned to the first occurrence, and `B` to the second; no value is assigned to the third occurrence.

Assigning Initial Values to Two-Dimensional Arrays

This section illustrates how initial values are assigned to a two-dimensional array. The following topics are covered:

- Preliminary Information
- Assigning the Same Value
- Assigning Different Values

Preliminary Information

For the examples shown in this section, let us assume a two-dimensional array with three occurrences in the first dimension (“rows”) and four occurrences in the second dimension (“columns”):

```
1 #ARRAY (A1/1:3,1:4)
```

Vertical: First Dimension (1:3), Horizontal: Second 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)

The first set of examples illustrates how the *same* initial value is assigned to occurrences of a two-dimensional array; the second set of examples illustrates how *different* initial values are assigned.

In the examples, please note in particular the usage of the notations `*` and `V`. Both notations refer to *all* occurrences of the dimension concerned: `*` indicates that all occurrences in that dimension are initialized with the *same* value, while `V` indicates that all occurrences in that dimension are initialized with *different* values.

Assigning the Same Value

- To assign an initial value to one occurrence, you specify:

```
1 #ARRAY (A1/1:3,1:4) INIT (2,3) <'A'>
```

		A	

- To assign the same initial value to one occurrence in the second dimension - in all occurrences of the first dimension - you specify:

```
1 #ARRAY (A1/1:3,1:4) INIT (*,3) <'A'>
```

		A	
		A	
		A	

- To assign the same initial value to a range of occurrences in the first dimension - in all occurrences of the second dimension - you specify:

```
1 #ARRAY (A1/1:3,1:4) INIT (2:3,*) <'A'>
```

A	A	A	A
A	A	A	A

- To assign the same initial value to a range of occurrences in each dimension, you specify:

```
1 #ARRAY (A1/1:3,1:4) INIT (2:3,1:2) <'A'>
```

A	A		
A	A		

- To assign the same initial value to all occurrences (in both dimensions), you specify:

```
1 #ARRAY (A1/1:3,1:4) INIT ALL <'A'>
```

A	A	A	A
A	A	A	A
A	A	A	A

Alternatively, you could specify:

```
1 #ARRAY (A1/1:3,1:4) INIT (*,*) <'A'>
```

Assigning Different Values

- `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'>`

	A	A	
	B	B	
	C	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'>`

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'>`

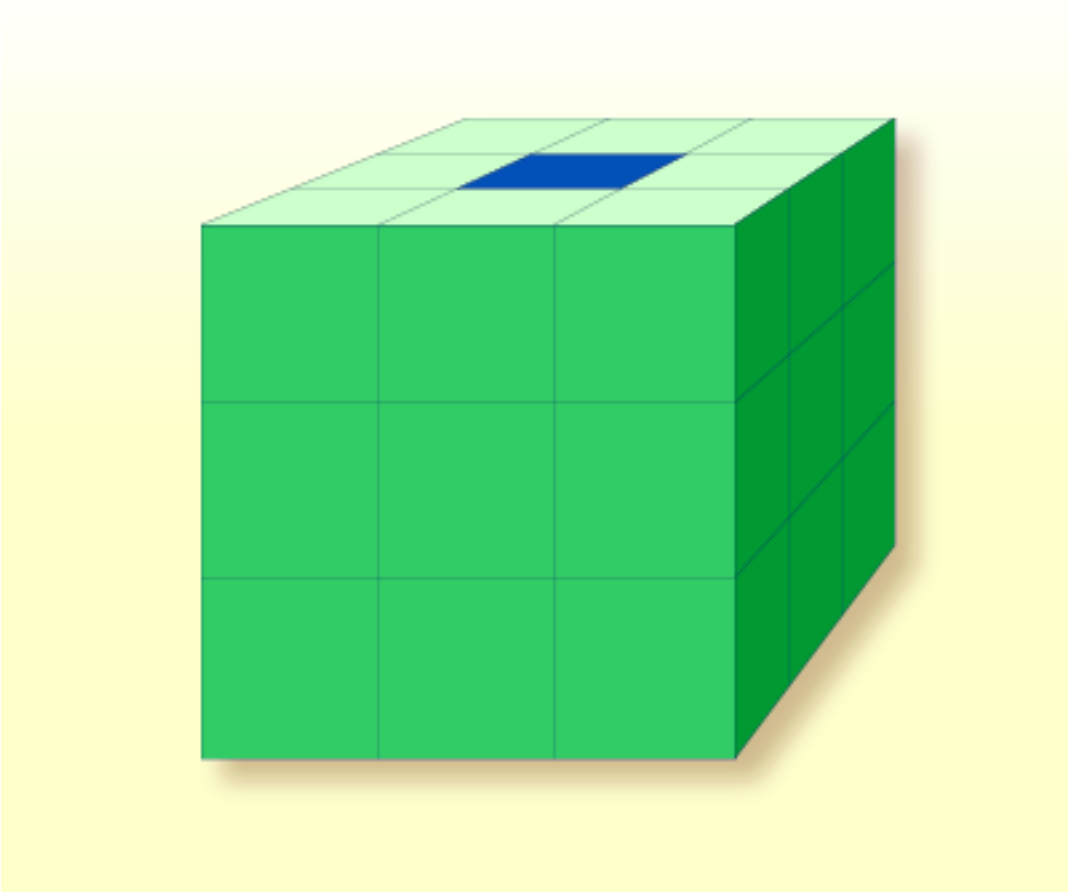
	B		
A	B		
	B	C	D

- `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

A Three-Dimensional Array

A three-dimensional array could be visualized as follows:



The array illustrated here would be defined as follows (at the same time assigning an initial value to the highlighted field in Row 1, Column 2, Plane 2):

```
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
...
```

If defined as a local data area in the data area editor, the same array would look as follows:

```
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 as Part of a Larger Data Structure

The multiple dimensions of an array make it possible to define data structures analogous to COBOL or PL1 structures.

Example:

```
DEFINE DATA LOCAL
1 #AREA
  2 #FIELD1 (A10)
  2 #GROUP1 (1:10)
    3 #FIELD2 (P2)
    3 #FIELD3 (N1/1:4)
END-DEFINE
...
```

In this example, the data area #AREA has a total size of:

10 + (10 * (2 + (1 * 4))) bytes = 70 bytes

#FIELD1 is alphanumeric and 10 bytes long. #GROUP1 is the name of a sub-area within #AREA, which consists of 2 fields and has 10 occurrences. #FIELD2 is packed numeric, length 2. #FIELD3 is the second field of #GROUP1 with four occurrences, and is numeric, length 1.

To reference a particular occurrence of #FIELD3, two indices are required: first, the occurrence of #GROUP1 must be specified, and second, the particular occurrence of #FIELD3 must also be specified. For example, in an ADD statement later in the same program, #FIELD3 would be referenced as follows:

```
ADD 2 TO #FIELD3 (3,2)
```

Database Arrays

Adabas supports array structures within the database in the form of **multiple-value fields** and **periodic groups**. These are described under ***Database Arrays***.

The following example shows a `DEFINE DATA` view containing a multiple-value field:

```
DEFINE DATA LOCAL
1 EMPLOYEES-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 ADDRESS-LINE (1:10) /* <--- MULTIPLE-VALUE FIELD
END-DEFINE
...
```

The same view in a local data area would look as follows:

```
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
```

Using Arithmetic Expressions in Index Notation

A simple arithmetic expression may also be used to express a range of occurrences in an array.

Examples:

`MA (I:I+5)`	Values of the field `MA` are referenced, beginning with value `I` and ending with value `I+5`.
`MA (I+2:J-3)`	Values of the field `MA` are referenced, beginning with value `I+2` and ending with value `J-3`.

Only the arithmetic operators plus (+) and minus (-) may be used in index expressions.

Arithmetic Support for Arrays

Arithmetic support for arrays include operations at array level, at row/column level, and at individual element level.

Only simple arithmetic expressions are permitted with array variables, with only one or two operands and an optional third variable as the receiving field.

Only the arithmetic operators plus (+) and minus (-) are allowed for expressions defining index ranges.

Examples of Array Arithmetics

The following examples assume the following field definitions:

```
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(*,*)`

   The result operand, array #B, contains the addition, element by element, of the array #A and the original value of array #B.

2. `ADD 4 TO #A(*,2)`

   The second column of the array #A is replaced by its original value plus 4.

3. `ADD 2 TO #A(2,*)`

   The second row of the array #A is replaced by its original value plus 2.

4. `ADD #A(2,*) TO #B(4,*)`

   The value of the second row of array #A is added to the fourth row of array #B.

5. `ADD #A(2,*) TO #B(*,2)`

   This is an illegal operation and will result in a syntax error. Rows may only be added to rows and columns to columns.

6. `ADD #A(2,*) TO #C`

   All values in the second row of the array #A are added to the scalar value #C.

7. `ADD #A(2,5:7) TO #C`

   The fifth, sixth, and seventh column values of the second row of array #A are added to the scalar value #C.

22 X-Arrays

- Definition 182
- Storage Management of X-Arrays 183
- Storage Management of X-Group Arrays 183
- Referencing an X-Array 185
- Parameter Transfer with X-Arrays 186
- Parameter Transfer with X-Group Arrays 187
- X-Array of Dynamic Variables 188
- Lower and Upper Bound of an Array 189

When an ordinary array field is defined, you have to specify the index bounds exactly, hence the number of occurrences for each dimension. At runtime, the complete array field is existent by default; each of its defined occurrences can be accessed without performing additional allocation operations. The size layout cannot be changed anymore; you may neither add nor remove field occurrences.

However, if the number of occurrences needed is unknown at development time, but you want to flexibly increase or decrease the number of the array fields at runtime, you should use what is called an X-array (eXtensible array).

An X-array can be resized at runtime and can help you manage memory more efficiently. For example, you can use a large number of array occurrences for a short time and then reduce memory when the application is no longer using the array.

Definition

An X-array is an array of which the number of occurrences is undefined at compile time. It is defined in a `DEFINE DATA` statement by specifying an asterisk (*) for at least one index bound of at least one array dimension. An asterisk (*) character in the index definition represents a variable index bound which can be assigned to a definite value during program execution. Only one bound - either upper or lower - may be defined as variable, but not both.

An X-array can be defined whenever a (fixed) array can be defined, i.e. at any level or even as an indexed group. It cannot be used to access MU-/PE-fields of a database view. A multidimensional array may have a mixture of constant and variable bounds.

Example:

```
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
```

Storage Management of X-Arrays

Occurrences of an X-array must be allocated explicitly before they can be accessed. To increase or decrease the number of occurrences of a dimension, the statements EXPAND, RESIZE and REDUCE may be used.

However, the number of dimensions of the X-array (1, 2 or 3 dimensions) cannot be changed.

Example:

```
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
```

Storage Management of X-Group Arrays

If you want to increase or decrease occurrences of X-group arrays, you must distinguish between independent and dependent dimensions.

A dimension which is specified directly (not inherited) for an X-(group) array is *independent*.

A dimension which is *not* specified directly, but inherited for an array is *dependent*.

Only independent dimensions of an X-array can be changed in the statements EXPAND, RESIZE and REDUCE; dependent dimensions must be changed using the name of the corresponding X-group array which owns this dimension as independent dimension.

Example - Independent/Dependent Dimensions:

```
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
```

The following table shows whether the dimensions in the above program are independent or dependent.

Name	Dependent Dimension	Independent 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:*)

The only index notation permitted for a dependent dimension is either a single asterisk (*), a range defined with asterisks (*:*) or the index bounds defined.

This is to indicate that the bounds of the dependent dimension must be kept as they are and cannot be changed.

The occurrences of the dependent dimensions can only be changed by manipulating the corresponding array groups.

```
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
```

The `EXPAND` statements may be coded in an arbitrary order.

The following use of the `EXPAND` statement is not allowed, since the arrays only have dependent dimensions.

```
EXPAND ARRAY #X-ARR1 TO ...
EXPAND ARRAY #X-GROUP-ARR2 TO ...
EXPAND ARRAY #X-ARR3 TO ...
```

Referencing an X-Array

The occurrences of an X-array must be allocated by an `EXPAND` or `RESIZE` statement before they can be accessed. The statements `READ`, `FIND` and `GET` allocate occurrences implicitly if values are obtained from Tamino.

As a general rule, an attempt to address a non existent X-array occurrence leads to a runtime error. In some statements, however, the access to a non materialized X-array field does not cause an error situation if all occurrences of an X-array are referenced using the complete range notation, for example: `#X-ARR(*)`. This applies to

- parameters used in a `CALL` statement,
- parameters used in the statements `CALLNAT`, `PERFORM`, `SEND EVENT` or `OPEN DIALOG`, if defined as optional parameters,
- source fields used in a `COMPRESS` statement,
- output fields supplied in a `PRINT` statement,
- fields referenced in a `RESET` statement.

If individual occurrences of a non materialized X-array are referenced in one of these statements, a corresponding error message is issued.

Example:

```
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  ↩
```

The asterisk (*) notation in an array reference stands for the complete range of a dimension. If the array is an X-array, the asterisk is the index range of the currently allocated lower and upper bound values, which are determined by the system variables `*LBOUND` and `*UBOUND`.

Parameter Transfer with X-Arrays

X-arrays that are used as parameters are treated in the same way as constant arrays with regard to the verification of the following:

- format,
- length,
- dimension or
- number of occurrences.

In addition, X-array parameters can also change the number of occurrences using the statement `RESIZE`, `REDUCE` or `EXPAND`. The question if a resize of an X-array parameter is permitted depends on three factors:

- the type of parameter transfer used, that is by reference or by value,
- the definition of the caller or parameter X-array, and
- the type of X-array range being passed on (complete range or subrange).

The following tables demonstrate when an `EXPAND`, `RESIZE` or `REDUCE` statement can be applied to an X-array parameter.

Example with Call By Value

Caller	Parameter		
	Static	Variable (1:V)	X-Array
Static	no	no	yes
X-array subrange, for example: `CALLNAT...#XA(1:5)`	no	no	yes
X-array complete range, for example: `CALLNAT...#XA(*)`	no	no	yes

Call By Reference/Call By Value Result

Caller	Parameter			
	Static	Variable (1:V)	X-Array with a fixed lower bound, e.g. `DEFINE DATA ↩ PARAMETER 1 #PX (A10/1:*)`	X-Array with a fixed upper bound, e.g. `DEFINE DATA ↩ PARAMETER 1 #PX (A10/*:1)`
Static	no	no	no	no
X-array subrange, for example: `CALLNAT...#XA(1:5)`	no	no	no	no
X-Array with a fixed lower bound, complete range, for example: `DEFINE DATA LOCAL 1 #XA(A10/1:*) ... CALLNAT...#XA(*)`	no	no	yes	no
X-Array with a fixed upper bound, complete range, for example: `DEFINE DATA LOCAL 1 #XA(A10/*:1) ... CALLNAT...#XA(*)`	no	no	no	yes

Parameter Transfer with X-Group Arrays

The declaration of an X-group array implies that each element of the group will have the same values for upper boundary and lower boundary. Therefore, the number of occurrences of dependent dimensions of fields of an X-group array can only be changed when the group name of the X-group array is given with a `RESIZE`, `REDUCE` or `EXPAND` statement (see *Storage Management of X-Group Arrays* above).

Members of X-group arrays may be transferred as parameters to X-group arrays defined in a parameter data area. The group structures of the caller and the callee need not necessarily be

identical. A RESIZE, REDUCE or EXPAND done by the callee is only possible as far as the X-group array of the caller stays consistent.

Example - Elements of X-Group Array Passed as Parameters:

Program:

```
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
```

Subprogram:

```
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
```

The RESIZE statement in the subprogram is not possible. It would result in an inconsistent number of occurrences of the fields defined in the X-group arrays of the program.

X-Array of Dynamic Variables

An X-array of dynamic variables may be allocated by first specifying the number of occurrences using the EXPAND statement and then assigning a value to the previously allocated array occurrences.

Example:

```
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
```

Lower and Upper Bound of an Array

The system variables `*LBOUND` and `*UBOUND` contain the current lower and upper bound of an array for the specified dimension(s): (1,2 or 3).

If no occurrences of an X-array have been allocated, the access to `*LBOUND` or `*UBOUND` is undefined for the variable index bounds, that is, for the boundaries that are represented by an asterisk (*) character in the index definition, and leads to a runtime error. In order to avoid a runtime error, the system variable `*OCCURRENCE` may be used to check against zero occurrences before `*LBOUND` or `*UBOUND` is evaluated:

Example:

```
IF *OCCURRENCE (#A) NE 0  AND  *UBOUND(#A) < 100 THEN ...
```

IV User-Defined Functions

23 User-Defined Functions

- Introduction to User-Defined Functions 194
- Restrictions 195
- Function Call versus Subprogram Call 195
- Function Definition (DEFINE FUNCTION) 198
- Symbolic and Variable Function Call 198
- Function Result and Parameters 198
- Explicit Prototype Definition (DEFINE PROTOTYPE) 199
- Implicit (Automatic) Prototype Definition 199
- Prototype Cast (PT Clause) 200
- Intermediate Result Definition (IR Clause) 200
- Combinations of Possible Prototype Definitions 200
- Evaluation Sequence of Functions in Statements 202
- Using a Function as a Statement 204

Related topics:

- Natural object type **function**
- **Function Call**
- Natural statements `DEFINE FUNCTION`, `DEFINE PROTOTYPE`

Introduction to User-Defined Functions

A user-defined function is a programming object of type **function**, containing Natural statements which implement a specific functional task. The invocation of a user-defined function ("function call") usually has a number of parameters and returns a result.

The syntactical representation of a **function call** is the function name (*`call-name`*) followed by a special bracket notation containing the parameters, which is, for example: `FCTNAME(<...>)`.

A function can be called from any place within a Natural statement where an operand is expected which is only *read*, but not *modified*. The result returned by the function is processed by the statement at the place where the function call is embedded, like a field containing the same value.

A function can also be called in a stand-alone mode, outside of any Natural statement. In this case, the function result is not processed.

Usually, the result value returned by the function depends on the parameters provided with the function call. As well as with a Natural subprogram, a parameter can be defined within a function as a "by reference", "by value" or "by value result" field. This makes it possible to provide parameter values in a function call which are only transfer compatible to what is defined in the function definition. Moreover, it allows you to exchange data between the calling object and the function not only via the function result, but also via parameters. The correctness of the parameter list and the compatibility of the result value is checked at compilation, either by means of an existing cataloged function object or by means of a result and parameter layout definition described with a `DEFINE PROTOTYPE` statement.

All function calls used in a Natural statement are evaluated in a separate step before the statement execution is started. They are evaluated in the same order in which they appear in the statement.

Restrictions

At some places in a programming object, function calls cannot be used. This includes

- positions where the operand value is changed by the Natural statement;
- all kinds of database calls (for example, `FIND`, `READ`);
- `DEFINE DATA` statement;
- `IF BREAK` statement;
- `AT BREAK` statement;
- array index expressions;
- Natural system functions (for example, `AVER`, `SUM`);
- parameters of a function call.

Function Call versus Subprogram Call

The following is a comparison of the characteristics of a function call with those of a subprogram call.

- What is similar?
- What is different?
- Example of a Function Call
- Example of a Subprogram Call

What is similar?

The following similarities exist between a function call and a subprogram call:

- The programming code forming the routine logic is coded inside a separate object, either in a function or a subprogram.
- Parameters are defined in the object using a `DEFINE DATA PARAMETER` statement, with various communication modes (for example, “by value”).

What is different?

The following differences exist between a function call and a subprogram call:

- A function call can be used at any position in a Natural statement where a read-only operand is possible (for exceptions, see ***Restrictions***), whereas subprograms can be invoked only via the CALLNAT statement.
- A function returns a result value which can instantly be processed by the statement that includes the function call. The use of a temporary variable is not required. A CALLNAT statement can only return data via its parameters. To process such a value with another statement, it needs to be declared as an explicit variable.
- Parameters and the result of a function call are always verified if the called function already exists at compilation time. Subprogram calls are checked only if the compiler option PCHECK is set to ON.
- The name of a function, which is used to call the function, is defined within the DEFINE FUNCTION statement and may differ from the name of the object containing the function. Similar to subroutines, the name of a function may have up to 32 characters. A subprogram is called by the name of the subprogram object. Therefore, the maximum name length is limited to 8 characters.

Example of a Function Call

The following example shows a program calling a function, and the function definition created with a DEFINE FUNCTION statement.

Program Calling the Function

```
** Example 'FUNCAX01': Calling a function  (Program)
************************************************************************
*
WRITE 'Function call' F#ADD(< 2,3 >)  /* Function call.
                                      /* No temporary variables needed.
*
END
```

Definition of Function F#ADD

```
** Example 'FUNCAX02': Calling a function  (Function)
************************************************************************
DEFINE FUNCTION F#ADD
  RETURNS #RESULT (I4)
  DEFINE DATA PARAMETER
    1 #SUMMAND1 (I4) BY VALUE
    1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE
  /*
  #RESULT := #SUMMAND1 + #SUMMAND2
  /*
```

```
END-FUNCTION
*
END ↩
```

Example of a Subprogram Call

To implement the same functionality as shown in the example of a function call by using a subprogram call instead, you need to specify temporary variables.

Program Calling the Subprogram

The following example shows a program calling a subprogram, involving the use of a temporary variable.

```
** Example 'FUNCAX03': Calling a subprogram  (Program)
************************************************************************
DEFINE DATA LOCAL
  1 #RESULT (I4) INIT <0>
END-DEFINE
*
CALLNAT 'FUNCAX04' #RESULT 2 3    /* Result is stored in #RESULT.
*
WRITE '=' #RESULT                 /* Print out the result of the
                                  /* subprogram.
*
END
```

Called Subprogram FUNCAX04

```
** Example 'FUNCAX04': Calling a subprogram  (Subprogram)
************************************************************************
DEFINE DATA PARAMETER
  1 #RESULT   (I4) BY VALUE RESULT
  1 #SUMMAND1 (I4) BY VALUE
  1 #SUMMAND2 (I4) BY VALUE
END-DEFINE
*
#RESULT := #SUMMAND1 + #SUMMAND2
*
END ↩
```

Function Definition (DEFINE FUNCTION)

A function is created in a separate Natural object of type **function**. It contains a single `DEFINE FUNCTION` statement, which defines the parameter interface, the result value and the program code forming the operation logic.

Symbolic and Variable Function Call

There are two modes to call a function, either in a direct form or indirect form. In the direct form (denoted as **"symbolic"** function call), the function name specified in the call is exactly the name of the function itself. In the indirect form (denoted as **"variable"** function call), the name specified in the function call is an alphanumeric variable with any name, which contains the name of the called function at runtime.

To define a variable function call, it is always necessary to use a `DEFINE PROTOTYPE VARIABLE` statement. Otherwise, the function call is assumed to be a symbolic function call; this means, the name itself is regarded as the function name.

See ***Function Call*** for more details about this topic.

Function Result and Parameters

Usually, function calls are used within Natural statements instead of variables or constant values. Since the compiler strictly requires the layout of the operands involved, it is essential to get the format, length and array structure of a function result. Moreover, if the parameter structure of a function is known, the parameters supplied in a function call can be checked for correctness at compilation time.

There are three options to provide this information

- with a `DEFINE PROTOTYPE` statement;
- implicit by the existent object (cataloged version) of the called function, which is automatically loaded by the compiler if no `DEFINE PROTOTYPE` statement was found before;
- with an explicit **(IR=)** clause specified in the function call.

The simplest way is certainly to use the second option, as it always takes the information from the object which is called at runtime. However, this is only possible if the cataloged object of the called function is available. If a function is called with a variable function call, via a variable containing the function name, the name of the function object is unknown and as a consequence cannot be

identified and loaded. In this case a DEFINE PROTOTYPE statement can be used to describe the function interface. The first two options comprise a full validation of the result and the parameters.

If neither a DEFINE PROTOTYPE statement nor the existent object of a function call is available, two casting options can be used in a function call. These are

- **(IR=)** to specify the function result format/length/array structure. This option does not incorporate any parameter checks.
- (PT=) to use a previously defined prototype with a name other than the function name.

Explicit Prototype Definition (DEFINE PROTOTYPE)

To specify the interface of a certain function, a DEFINE PROTOTYPE statement can be used. This statement defines the layout of the parameters, which are to be passed in a function call, and the format/length of the result field returned by the function. Furthermore, it indicates whether a function call is a **"symbolic"** or a **"variable"** function call.

When a function call was found, Natural tries to locate a prototype with the same name as the used function name. If such a prototype is found, the function result and parameter layouts of this prototype are used to resolve the function call. Especially, the decision on the function call mode ("symbolic" or "variable") is made at this place.

Implicit (Automatic) Prototype Definition

If a function call is resolved in a program, the compiler searches for a DEFINE PROTOTYPE statement with the function name, which has been defined before. If such a statement cannot be found, Natural tries to load the called function into the buffer pool. If successful, the function layout of the result and and the parameters is extracted from the object data and kept as if it was provided by an explicit DEFINE PROTOTYPE statement for this function. This manner of use is denoted as *automatic prototype definition*. It assures great conformity between the interface definition (at compile time) and the passing accuracy at runtime.

Prototype Cast (PT Clause)

In order to get the interface layout of a called function, Natural tries to locate a `DEFINE PROTOTYPE` statement with the same name as the function identifier. If such a statement is not available, and the called function object cannot be loaded (see ***Implicit (Automatic) Prototype Definition***), a `(PT=)` clause can be specified in the function call. If such a clause is applied, the `DEFINE PROTOTYPE` statement with the referenced name (which is different from the function name) is used to describe the function result and to validate the parameters.

Example:

```
#I := #MULT(<(PT=#ADD) 2 , 3>)
```

In this example, the function `#MULT` is called, but the result and parameter layouts are used from a prototype whose name is `#ADD`.

Intermediate Result Definition (IR Clause)

Usually, the function result is specified by a `DEFINE PROTOTYPE` statement, which is either coded explicitly or will be created automatically with the function object (see ***Implicit (Automatic) Prototype Definition***). If such a definition is not available, the result layout can be specified by using the **(IR=)** clause in the function call. If this clause is used, it determines which format/length the compiler should use for the result field in the statement generation. This clause can also be specified if the prototype definition is available for a function call. In this case, the result layout in the prototype is overruled by the `(IR=)` clause specification; the parameter checks, however, are performed according to the prototype definition.

Combinations of Possible Prototype Definitions

In order to resolve a function call, the compiler needs information on

- the function call mode (**symbolic** or **variable**);
- the layout (format/length) of the function result;
- the layout (format/length) of the function parameters.

Different options allow you to provide this data, which are the explicit prototype definition, the implicit prototype definition, the `(PT=)` option, and the **(IR=)** option. But which one has an effect if multiple of these clauses are used?

A function call is used as a **variable** function call if there is a related prototype with the same name, which contains a `VARIABLE` clause. In all other cases, the function call is treated as a **symbolic** function call.

The *result* is determined in the following order:

- the definition provided in **(IR=)**, if this clause is specified;
- the `RETURNS` definition in the prototype referenced in `(PT=)`, if this clause is specified;
- the explicit prototype definition (`DEFINE PROTOTYPE`) with the same name as used in the function call, if it exists;
- the implicit prototype definition, which is loaded automatically from the existing function object.

If none of these options applies, a syntax error is raised.

The *parameter* checks are performed according to the definition in:

- the prototype definition referenced in `(PT=)`, if this clause is specified;
- the explicit prototype definition (`DEFINE PROTOTYPE`) with the same name as used in the function call, if it exists;
- the implicit prototype definition, which is loaded automatically from the existing function object.

If none of these options applies, the parameter validation is not performed. This allows you to supply any number and layout of parameters in the function call, without receiving a syntax error.

Example with Multiple Definitions in a Function Call

Program:

```
** Example 'FUNCBX01': Declare result value and parameters  (Program)
************************************************************************
*
DEFINE DATA LOCAL
  1 #PROTO-NAME (A20)
  1 #PARM1      (I4)
  1 #PARM2      (I4)
END-DEFINE
*
DEFINE PROTOTYPE VARIABLE #PROTO-NAME
  RETURNS (I4)
  DEFINE DATA PARAMETER
    1 #P1 (I4) BY VALUE OPTIONAL
    1 #P2 (I4) BY VALUE
  END-DEFINE
END-PROTOTYPE
*
#PROTO-NAME := 'F#MULTI'
#PARM1      := 3
```

```
#PARM2      := 5
*
WRITE #PROTO-NAME(<#PARM1, #PARM2>)
WRITE #PROTO-NAME(<1X ,5>)
*
WRITE F#MULTI(<(PT=#PROTO-NAME) #PARM1,#PARM2>)
*
WRITE F#MULTI(<(IR=N20) #PARM1, #PARM2>)
*
END
```

Function F#MULTI:

```
** Example 'FUNCBX02': Declare result value and parameters  (Function)
************************************************************************
DEFINE FUNCTION F#MULTI
  RETURNS #RESULT (I4) BY VALUE
  DEFINE DATA PARAMETER
    1 #FACTOR1 (I4) BY VALUE OPTIONAL
    1 #FACTOR2 (I4) BY VALUE
  END-DEFINE
  /*
  IF #FACTOR1 SPECIFIED
    #RESULT := #FACTOR1 * #FACTOR2
  ELSE
    #RESULT := #FACTOR2 * 10
  END-IF
  /*
END-FUNCTION
*
END ↩
```

Evaluation Sequence of Functions in Statements

Instead of operands, function calls can be used directly in statements. However, this is only allowed with operands which are only *read*, but not *modified* by the statement.

All function calls are evaluated before the statement execution starts. The returned result values are stored in temporary fields and passed to the statement. The functions are executed in the same order in which they appear in the statement. If a function call has parameters which are modified by the function execution, you should consider that this can influence the statement result. This may apply if the same parameter is used at another place in the same statement.

Example:

Before the COMPUTE statement is started, variable #I has the value 1. In a first step, function F#RETURN is executed. This changes #I to value 2 and returns the same value as the function result. After

this, the COMPUTE operation starts and adds the incremented #I (2) and the temporary field (2) to a sum of 4.

Program:

```
** Example 'FUNCCX01': Parameter changed within function (Program)
************************************************************************
DEFINE DATA LOCAL
  1 #I      (I2) INIT <1>
  1 #RESULT (I2)
END-DEFINE
*
COMPUTE #RESULT := #I + F#RETURN(<#I>)  /* First evaluate function call,
                                        /* then execute the addition.
*
WRITE '#I     :' #I /
      '#RESULT:' #RESULT
*
END
```

Function:

```
** Example 'FUNCCX02': Parameter changed within function (Function)
************************************************************************
DEFINE FUNCTION F#RETURN
  RETURNS #RESULT (I2) BY VALUE
  DEFINE DATA PARAMETER
    1 #PARM1 (I2) BY VALUE RESULT
  END-DEFINE
  /*
  #PARM1  := #PARM1 + 1       /* Increment parameter.
  #RESULT := #PARM1           /* Set result value.
  /*
END-FUNCTION
*
END ↩
```

Output of Program FUNCCX01:

```
#I     :      2
#RESULT:      4 ↩
```

Using a Function as a Statement

A function can also be called stand-alone, without being embedded in other statements. In this case, the function return value is completely ignored.

If such an execution mode is desired, only the function call is coded, which then stands for a statement. In order to prevent an unwanted link to the previous statement in the source code, a semicolon must be used to explicitly separate the function call from this statement.

Example:

Program:

```
** Example 'FUNCDX01': Using a function as a statement (Program)
************************************************************************
DEFINE DATA LOCAL
  1 #A (I4) INIT <1>
  1 #B (I4) INIT <2>
END-DEFINE
*
*
WRITE 'Write:' #A #B
F#PRINT-ADD(< 2,3 >)     /* Function call belongs to operand list
                        /* immediately preceding it.
*
WRITE // '************************' //
*
WRITE 'Write:' #A #B;    /* Semicolon separates operands and function.
F#PRINT-ADD(< 2,3 >)     /* Function call does not belong to the
                         /* operand list.
*
END
```

Function:

```
** Example 'FUNCDX02': Using a function as a statement (Function)
************************************************************************
DEFINE FUNCTION F#PRINT-ADD
  RETURNS (I4)
  DEFINE DATA PARAMETER
    1 #SUMMAND1 (I4) BY VALUE
    1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE
  /*
  F#PRINT-ADD := #SUMMAND1 + #SUMMAND2    /* Result of function call.
  WRITE 'Function call:' F#PRINT-ADD
  /*
END-FUNCTION
```

```
*
END ↩
```

Output of Program `FUNCDX01`:

```
Function call:           5
Write:           1           2           5

************************

Write:           1           2
Function call:           5 ↩
```

V Accessing Data in a Database

This part describes various aspects of accessing data in a database with Natural.

Natural and Database Access

Accessing Data in an Adabas Database

Accessing Data in an SQL Database

Accessing Data in a Tamino Database

24 Natural and Database Access

- Database Management Systems Supported by Natural 210
- Profile Parameters Influencing Database Access 211
- Access through Data Definition Modules 211
- Natural's Data Manipulation Language 212
- Natural's Special SQL Statements 213

This chapter gives an overview of the facilities that Natural provides for accessing different types of database management systems.

Database Management Systems Supported by Natural

Natural offers specific database interfaces for the following types of database management systems (DBMS):

- Nested-relational DBMS (Adabas)
- SQL-type DBMS (Oracle, Sybase, Informix, MS SQL Server)
- XML-type DBMS (Tamino)

The following topics are covered below:

- Adabas
- Tamino
- SQL Databases

Adabas

Via its integrated Adabas interface, Natural can access Adabas databases either on a local machine or on remote computers. For remote access, an additional routing and communication software such as Entire Net-Work is necessary. In any case, the type of host machine running the Adabas database is transparent for the Natural user.

Tamino

Natural for Tamino offers the possibility to access a Tamino database server on a local machine or on a remote host using a native HTTP protocol. The Tamino database can be accessed in the same manner as data access is done with Adabas or SQL databases.

SQL Databases

Natural acesses SQL database systems via Entire Access, a generic interface and routing software that supports various SQL database management systems such as Oracle, MS SQL Server or standardized ODBC connections. For a complete overview of the SQL database management systems and platforms supported, refer to the Entire Access documentation. Information on Natural configuration aspects is contained in the document Natural and Entire Access.

Profile Parameters Influencing Database Access

There are various Natural profile parameters to define how Natural handles the access to databases.

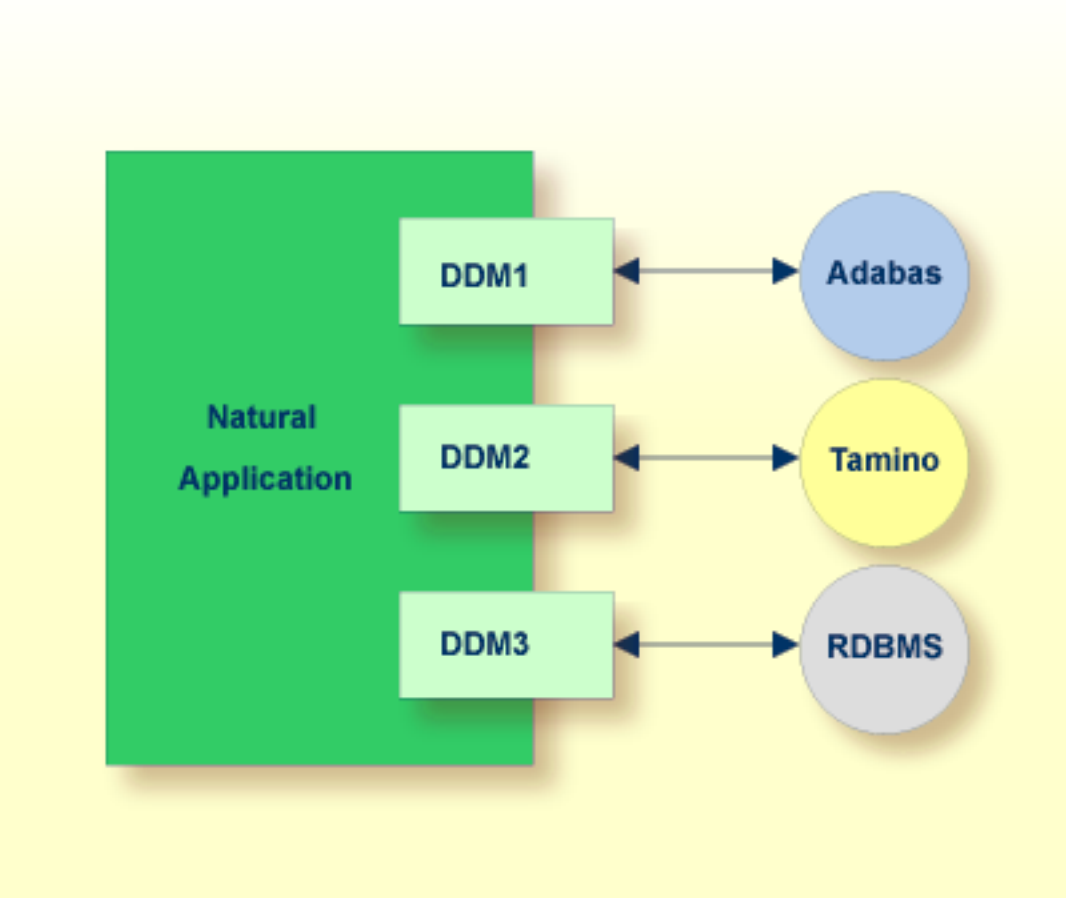
For an overview of these profile parameters, see the section *Database Management* in *Overview of Profile Parameters* in the *Configuration Utility* documentation.

For a detailed parameter description, refer to the corresponding section in the *Parameter Reference*.

Access through Data Definition Modules

To enable convenient and transparent access to the different database management systems, a special object, the "data definition module" (DDM), is used in Natural. This DDM establishes the connection between the Natural data structures and the data structures in the database system to be used. Such a database structure might be a table in an SQL database, a file in an Adabas database or a doctype in a Tamino database. Hence, the DDM hides the real structure of the database accessed from the Natural application. DDMs are created using the Natural DDM editor.

Natural is capable of accessing multiple types of databases (Adabas, Tamino, RDBMS) from within a single application by using references to DDMs that represent the specific data structures in the specific database system. The diagram below shows an application that connects to different types of database.



Natural's Data Manipulation Language

Natural has a built-in data manipulation language (DML) that allows Natural applications to access all database systems supported by Natural using the same language statements such as `FIND`, `READ`, `STORE` or `DELETE`. These statements can be used in a Natural application without knowing the type of database that is going to be accessed.

Natural determines the real type of database system from its configuration files and translates the DML statements into database-specific commands; that is, it generates direct commands for Adabas, SQL statement strings and host variable structures for SQL databases and XQuery requests for a Tamino database.

Because some of the Natural DML statements provide functionality that cannot be supported for all database types, the use of this functionality is restricted to specific database systems. Please, note the corresponding database-specific considerations in the statements documentation.

Natural's Special SQL Statements

In addition to the “normal” Natural DML statements, Natural provides a set of SQL statements for a more specific use in conjunction with SQL database systems; see *SQL Statements Overview* (in the *Statements* documentation).

Flexible SQL and facilities for working with stored procedures complete the set of SQL commands. These statements can be used for SQL database access only and are not valid for Adabas or other non-SQL-databases.

25 Accessing Data in an Adabas Database

- Adabas Database Management Interfaces ADA and ADA2 216
- Data Definition Modules - DDMs 216
- Database Arrays 218
- Defining a Database View 224
- Statements for Database Access 227
- Multi-Fetch Clause 239
- Database Processing Loops 240
- Database Update - Transaction Processing 246
- Selecting Records Using ACCEPT/REJECT 253
- AT START/END OF DATA Statements 257
- Unicode Data 259

This chapter describes various aspects of accessing data in an Adabas database with Natural.

Adabas Database Management Interfaces ADA and ADA2

Natural's Adabas database management interfaces ADA and ADA2 are considered distinct database interfaces like, for example, ADA and SQL.

Database type ADA is Natural's default interface to Adabas databases. It is the appropriate choice if new Adabas functionality as introduced with Adabas Version 6 on Open Systems and Adabas Version 8 on mainframes is not concerned.

Database type ADA2 is provided as an extended interface to Adabas databases as of Version 6 on Open Systems and Adabas as of Version 8 on mainframes. In particular, it supports Adabas LA fields, Adabas large object fields and extended Adabas buffer lengths. The support of Adabas LA and large object fields implies the use of Natural format `(A) DYNAMIC` in a view definition, the support of extended Adabas buffer lengths enables the definition of view sizes that exceed 64 KB. For further information, refer to ***Defining a Database View***.

Database type ADA2 does not support **multi-fetch processing**. Corresponding global and local definitions are ignored at runtime.

Software AG products which have their own system files require a corresponding physical database of database type ADA.

Natural objects that were compiled with database type ADA can be executed in an environment where the corresponding Adabas database is defined as database type ADA2.

Data Definition Modules - DDMs

For Natural to be able to access a database file, a logical definition of the physical database file is required. Such a logical file definition is called a data definition module (DDM).

This section covers the following topics:

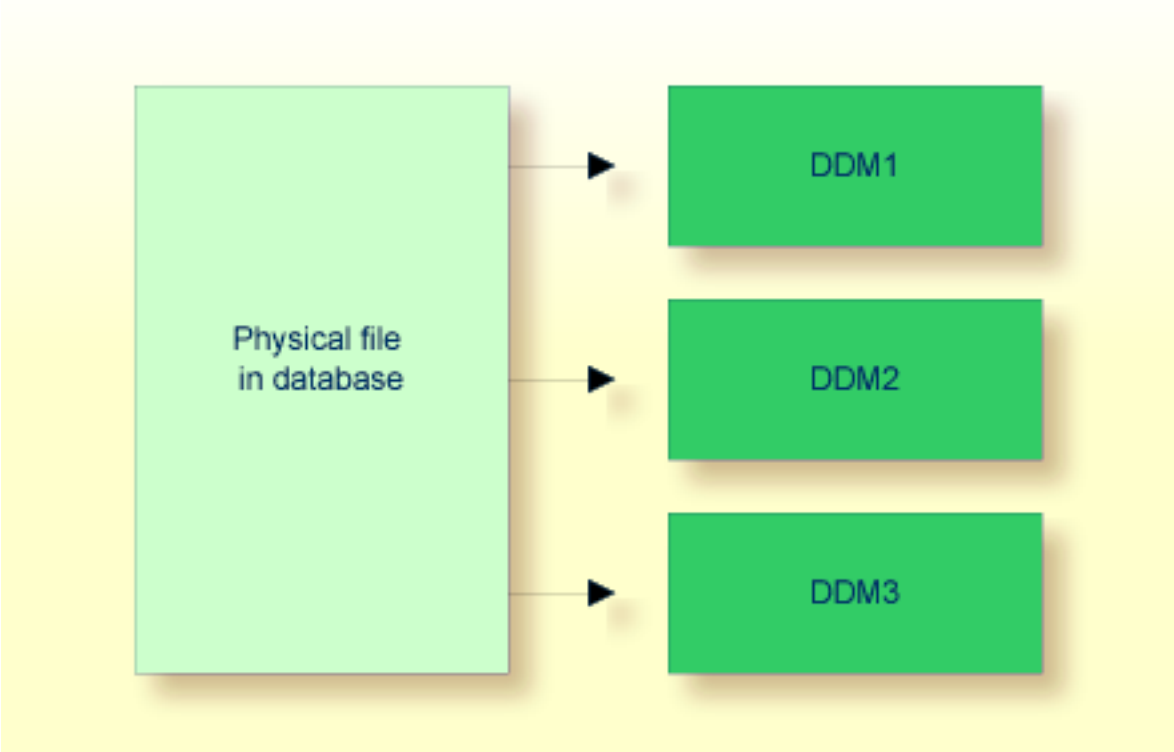
- Use of Data Definition Modules
- Maintaining DDMs

- Listing/Displaying DDMs

Use of Data Definition Modules

The data definition module contains information about the individual fields of the file - information which is relevant for the use of these fields in a Natural program. A DDM constitutes a logical view of a physical database file.

For each physical file of a database, one or more DDMs can be defined. And for each DDM one or more data views can be defined as described *View Definition* in the `DEFINE DATA` statement documentation and explained in the section ***Defining a Database View***.



DDMs are defined by the Natural administrator with Predict (or, if Predict is not available, with the corresponding Natural function).

Maintaining DDMs

For each database field, a DDM contains the database-internal field name as well as the "external" field name, that is, the name of the field as used in a Natural program. Moreover, the formats and lengths of the fields are defined in the DDM, as well as various specifications that are used when the fields are output with a `DISPLAY` or `WRITE` statement (column headings, edit masks, etc.).

For the field attributes defined in a DDM, refer to *Using the DDM Editor* in the section *DDM Editor* of the *Editors* documentation.

Listing/Displaying DDMs

If you do not know the name of the DDM you want, you can use the system command `LIST VIEW` to get a list of all existing DDMs that are available in the current library. From the list, you can then select a DDM for display.

System command	Meaning
`LIST VIEW`	Displays a list of all views (DDMs).
`LIST VIEW` *`view-name`*	If you specify a single view name, the specified view will be displayed. For the *`view-name`* you can use asterisk notation to display a list of all views (`*`) or a certain range of views (for example: `A*`).

Database Arrays

Adabas supports array structures within the database in the form of multiple-value fields and periodic groups.

This section covers the following topics:

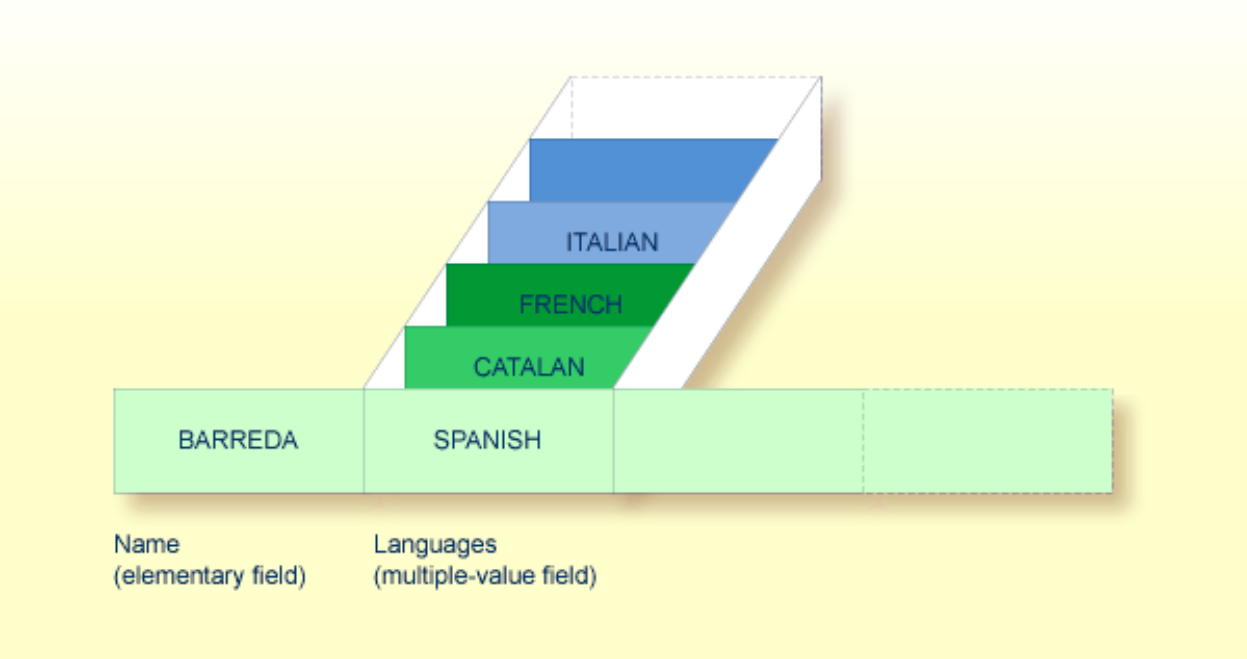
- Multiple-Value Fields
- Periodic Groups
- Referencing Multiple-Value Fields and Periodic Groups
- Multiple-Value Fields within Periodic Groups
- Referencing Multiple-Value Fields within Periodic Groups

- Referencing the Internal Count of a Database Array

Multiple-Value Fields

A multiple-value field is a field which can have more than one value (up to 65534, depending on the Adabas version and definition of the FDT) within a given record.

Example:



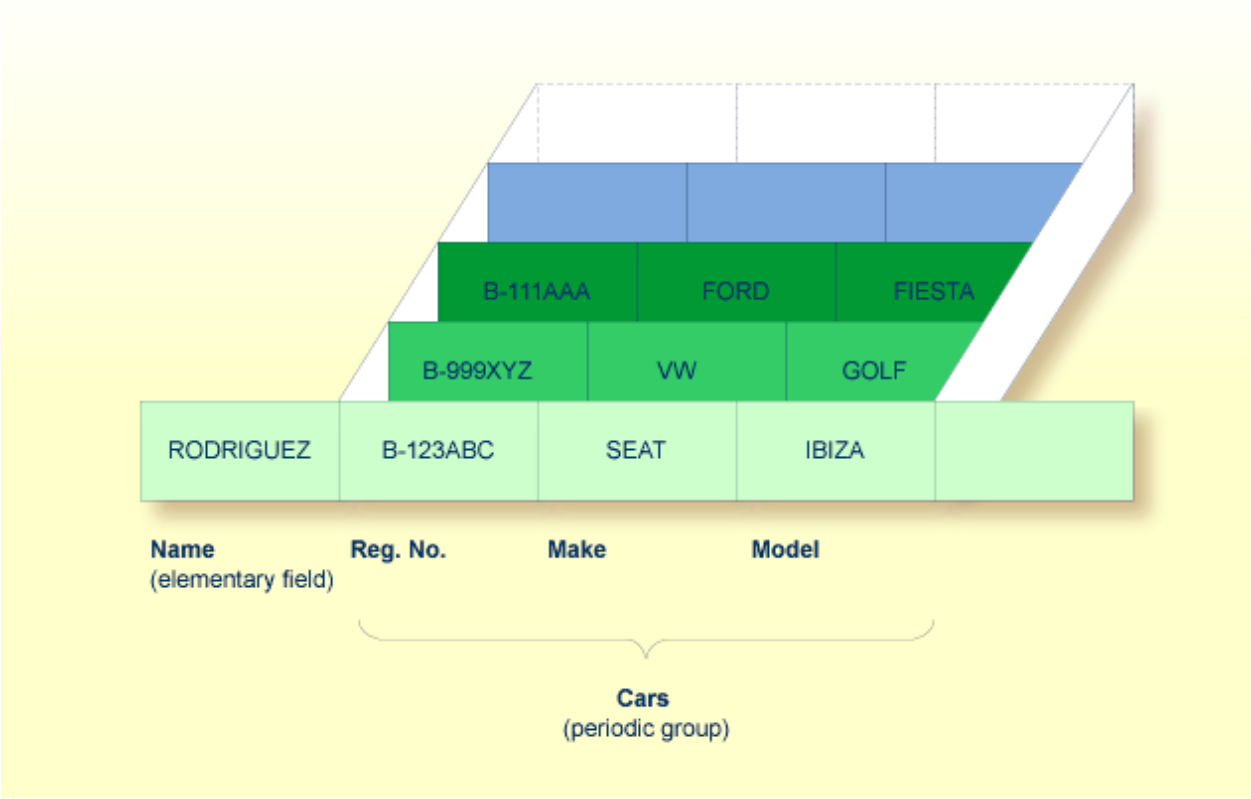
Assuming that the above is a record in an employees file, the first field (Name) is an elementary field, which can contain only one value, namely the name of the person; whereas the second field (Languages), which contains the languages spoken by the person, is a multiple-value field, as a person can speak more than one language.

Periodic Groups

A periodic group is a group of fields (which may be elementary fields and/or multiple-value fields) that may have more than one occurrence (up to 65534, depending on the Adabas version and definition of the field definition table (FDT)) within a given record.

The different values of a multiple-value field are usually called “occurrences”; that is, the number of occurrences is the number of values which the field contains, and a specific occurrence means a specific value. Similarly, in the case of periodic groups, occurrences refer to a group of values.

Example:



Assuming that the above is a record in a vehicles file, the first field (Name) is an elementary field which contains the name of a person; Cars is a periodic group which contains the automobiles owned by that person. The periodic group consists of three fields which contain the registration number, make and model of each automobile. Each occurrence of Cars contains the values for one automobile.

Referencing Multiple-Value Fields and Periodic Groups

To reference one or more occurrences of a multiple-value field or a periodic group, you specify an "index notation" after the field name.

Examples:

The following examples use the multiple-value field `LANGUAGES` and the periodic group `CARS` from the previous examples.

The various values of the multiple-value field `LANGUAGES` can be referenced as follows.

Example	Explanation
LANGUAGES (1)	References the first value (SPANISH).
LANGUAGES (X)	The value of the variable X determines the value to be referenced.
LANGUAGES (1:3)	References the first three values (SPANISH, CATALAN and FRENCH).
LANGUAGES (6:10)	References the sixth to tenth values.
LANGUAGES (X:Y)	The values of the variables X and Y determine the values to be referenced.

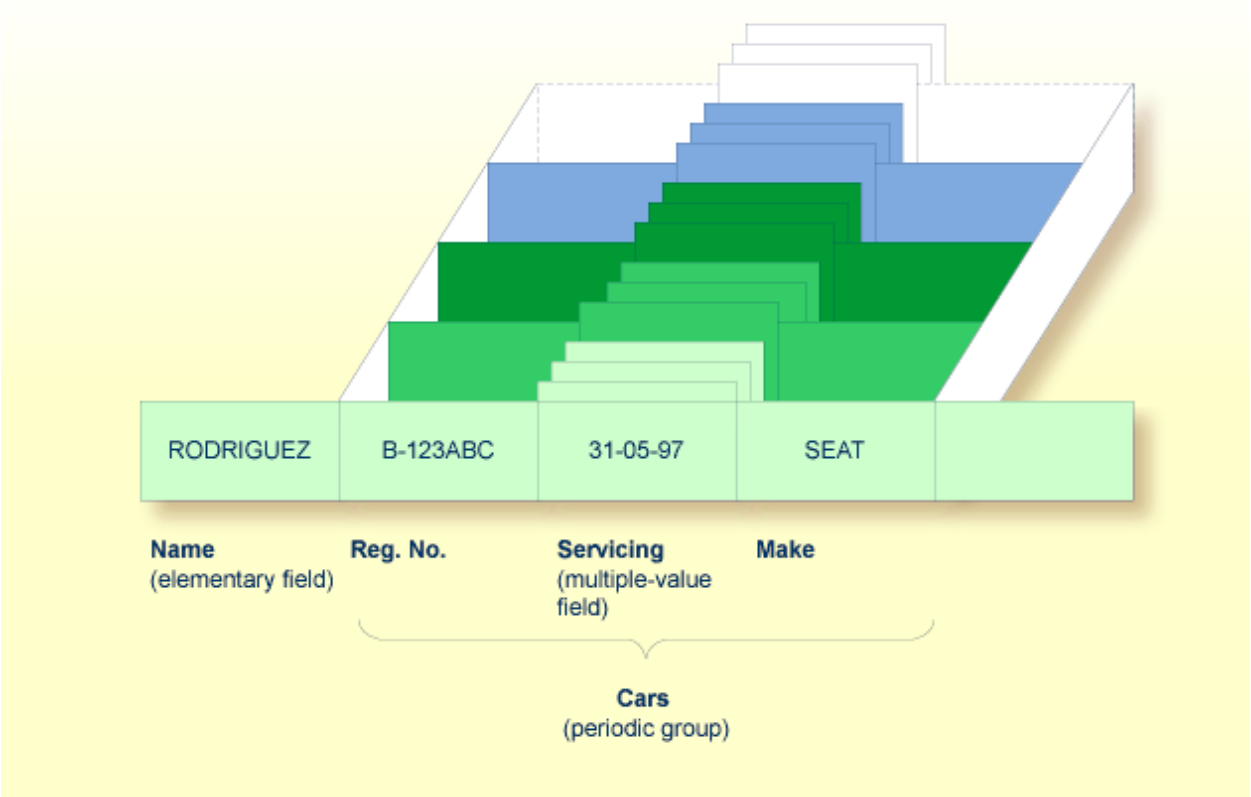
The various occurrences of the periodic group CARS can be referenced in the same manner:

Example	Explanation
CARS (1)	References the first occurrence (B-123ABC/SEAT/IBIZA).
CARS (X)	The value of the variable X determines the occurrence to be referenced.
CARS (1:2)	References the first two occurrences (B-123ABC/SEAT/IBIZA and B-999XYZ/VW/GOLF).
CARS (4:7)	References the fourth to seventh occurrences.
CARS (X:Y)	The values of the variables X and Y determine the occurrences to be referenced.

Multiple-Value Fields within Periodic Groups

An Adabas array can have up to two dimensions: a multiple-value field within a periodic group.

Example:



Assuming that the above is a record in a vehicles file, the first field (Name) is an elementary field which contains the name of a person; Cars is a periodic group, which contains the automobiles owned by that person. This periodic group consists of three fields which contain the registration number, servicing dates and make of each automobile. Within the periodic group Cars, the field Servicing is a multiple-value field, containing the different servicing dates for each automobile.

Referencing Multiple-Value Fields within Periodic Groups

To reference one or more occurrences of a multiple-value field within a periodic group, you specify a "two-dimensional" index notation after the field name.

Examples:

The following examples use the multiple-value field `SERVICING` within the periodic group `CARS` from the example above. The various values of the multiple-value field can be referenced as follows:

Example	Explanation
`SERVICING (1,1)`	References the first value of `SERVICING` in the first occurrence of `CARS` (`31-05-97`).
`SERVICING (1:5,1)`	References the first value of `SERVICING` in the first five occurrences of `CARS`.
`SERVICING (1:5,1:10)`	References the first ten values of `SERVICING` in the first five occurrences of `CARS`.

Referencing the Internal Count of a Database Array

It is sometimes necessary to reference a multiple-value field or a periodic group without knowing how many values/occurrences exist in a given record. Adabas maintains an internal count of the number of values in each multiple-value field and the number of occurrences of each periodic group. This count may be read in a `READ` statement by specifying `C*` immediately before the field name.

The count is returned in format/length N3. See *Referencing the Internal Count for a Database Array* for further details.

Example	Explanation
`C*LANGUAGES`	Returns the number of values of the multiple-value field `LANGUAGES`.
`C*CARS`	Returns the number of occurrences of the periodic group `CARS`.
`C*SERVICING (1)`	Returns the number of values of the multiple-value field `SERVICING` in the first occurrence of a periodic group (assuming that `SERVICING` is a multiple-value field within a periodic group.)

Defining a Database View

To be able to use database fields in a Natural program, you must specify the fields in a database view.

In the view, you specify the name of the data definition module (see *Data Definition Modules - DDMs*) from which the fields are to be taken, and the names of the **database fields** themselves (that is, their long names, not their database-internal short names).

The view may comprise an entire DDM or only a subset of it. The order of the fields in the view need not be the same as in the underlying DDM.

As described in the section *Statements for Database Access*, the view name is used in the statements `READ`, `FIND`, `HISTOGRAM` to determine which database is to be accessed.

For further information on the complete syntax of the view definition option or on the definition/re-definition of a group of fields, see *View Definition* in the description of the `DEFINE DATA` statement in the *Statements* documentation.

Basically, you have the following options to define a database view:

- **Inside the Program**
  You can define a database view inside the program, that is, directly within the `DEFINE DATA` statement of the program.
- **Outside the Program**
  You can define a database view outside the program, that is, in a separate programming object: either a local data area (LDA) or a global data area (GDA), with the `DEFINE DATA` statement of the program referencing that data area.

To define a database view inside the program

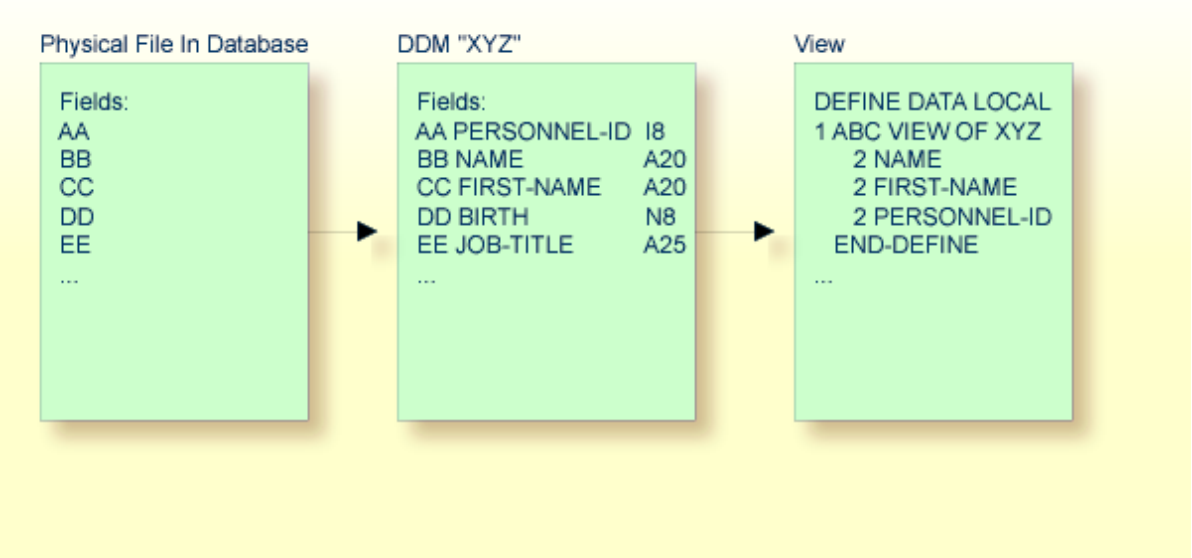
1 At Level 1, specify the view name as follows:

```
1 view-name VIEW OF ddm-name
```

where *view-name* is the name you choose for the view, *ddm-name* is the name of the DDM from which the fields specified in the view are taken.

2 At Level 2, specify the names of the database fields from the DDM.

In the illustration below, the name of the view is `ABC`, and it comprises the fields `NAME`, `FIRST-NAME` and `PERSONNEL-ID` from the DDM `XYZ`.



In the view, the format and length of a database field need not be specified, as these are already defined in the underlying DDM.

Sample Program:

In this example, the *view-name* is VIEWEMP, and the *ddm-name* is EMPLOYEES, and the names of the database fields taken from the DDM are NAME, FIRST-NAME and 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
...
```

To define a database view outside the program

1 In the program, specify:

```
DEFINE DATA LOCAL
       USING <data-area-name>
END-DEFINE
  ...
```

where *data-area-name* is the name you choose for the local or global data area, for example, LDA39.

2 In the data area to be referenced:

1. At Level 1 in the Name column, specify the name you choose for the view, and in the Miscellaneous column, the name of the DDM from which the fields specified in the view are taken.
2. At Level 2, specify the names of the database fields from the DDM.

   Example LDA39:

   In this example, the view name is VIEWEMP, the DDM name is EMPLOYEES, and the names of the database fields taken from the DDM are PERSONNEL-ID, FIRST-NAME and 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                    ↩
↩
```

Considerations Concerning Databases of Type ADA2

With databases of type `ADA2` (specified in the table DBMS Assignments in the Configuration Utility, see *Database Management System Assignments* in the *Configuration Utility* documentation), the following applies:

- If large alphanumeric (LA) or large object (LOB) fields (Adabas LA/LB option) are to be used, these fields can be specified within the view definition with both fixed format/length, for example, `A20` or `U20`, and dynamic format/length, for example, `(A)DYNAMIC` or `U(DYNAMIC)`.
- Length indicator fields `L@...` can also be specified within views if they are related to LA or LOB fields.

Statements for Database Access

To read data from a database, the following statements are available:

Statement	Meaning
`READ`	Select a range of records from a database in a specified sequence.
`FIND`	Select from a database those records which meet a specified search criterion.
`HISTOGRAM`	Read only the values of one database field, or determine the number of records which meet a specified search criterion.

READ Statement

The following topics are covered:

- Use of READ Statement
- Basic Syntax of READ Statement
- Example of READ Statement
- Limiting the Number of Records to be Read
- STARTING/ENDING Clauses
- WHERE Clause

- Further Example of READ Statement

Use of READ Statement

The READ statement is used to read records from a database. The records can be retrieved from the database

- in the order in which they are physically stored in the database (READ IN PHYSICAL SEQUENCE), or
- in the order of Adabas Internal Sequence Numbers (READ BY ISN), or
- in the order of the values of a descriptor field (READ IN LOGICAL SEQUENCE).

In this document, only READ IN LOGICAL SEQUENCE is discussed, as it is the most frequently used form of the READ statement.

For information on the other two options, please refer to the description of the READ statement in the *Statements* documentation.

Basic Syntax of READ Statement

The basic syntax of the READ statement is:

```
READ view IN LOGICAL SEQUENCE BY descriptor
```

or shorter:

```
READ view LOGICAL BY descriptor
```

- where

view	is the name of a view defined in the DEFINE DATA statement and as explained in ***Defining a Database View***.
descriptor	is the name of a database field defined in that view. The values of this field determine the order in which the records are read from the database.

If you specify a descriptor, you need not specify the **keyword** LOGICAL:

```
READ view BY descriptor
```

If you do not specify a descriptor, the records will be read in the order of values of the field defined as default descriptor (under Default Sequence) in the **DDM**. However, if you specify no descriptor, you must specify the **keyword** LOGICAL:

```
READ view LOGICAL
```

Example of 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
```

Output of Program READX01:

With the READ statement in this example, records from the EMPLOYEES file are read in alphabetical order of their last names.

The program will produce the following output, displaying the information of each employee in alphabetical order of the employees' last names.

```
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
```

If you wanted to read the records to create a report with the employees listed in sequential order by date of birth, the appropriate READ statement would be:

```
READ MYVIEW BY BIRTH
```

You can only specify a field which is defined as a "descriptor" in the underlying **DDM** (it can also be a subdescriptor, superdescriptor, hyperdescriptor or phonetic descriptor or a non-descriptor).

Limiting the Number of Records to be Read

As shown in the previous example program, you can limit the number of records to be read by specifying a number in parentheses after the keyword `READ`:

```
READ (6) MYVIEW BY NAME
```

In that example, the `READ` statement would read no more than 6 records.

Without the limit notation, the above `READ` statement would read *all* records from the `EMPLOYEES` file in the order of last names from `A` to `Z`.

STARTING/ENDING Clauses

The `READ` statement also allows you to qualify the selection of records based on the *value* of a descriptor field. With an `EQUAL TO/STARTING FROM` option in the `BY` clause, you can specify the value at which reading should begin. (Instead of using the keyword `BY`, you may specify the keyword `WITH`, which would have the same effect). By adding a `THRU/ENDING AT` option, you can also specify the value in the logical sequence at which reading should end.

For example, if you wanted a list of those employees in the order of job titles starting with `TRAINEE` and continuing on to `Z`, you would use one of the following statements:

```
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'
```

Note that the value to the right of the equal sign (=) or `STARTING FROM` option must be enclosed in apostrophes. If the value is numeric, this **text notation** is not required.

The sequence of records to be read can be even more closely specified by adding an end limit with a `THRU/ENDING AT` clause.

To read just the records with the job title `TRAINEE`, you would specify:

```
READ MYVIEW BY JOB-TITLE STARTING FROM 'TRAINEE' THRU 'TRAINEE'
READ MYVIEW WITH JOB-TITLE EQUAL TO 'TRAINEE'
                           ENDING AT 'TRAINEE'
```

To read just the records with job titles that begin with `A` or `B`, you would specify:

```
READ MYVIEW BY JOB-TITLE = 'A' THRU 'C'
READ MYVIEW WITH JOB-TITLE STARTING FROM 'A' ENDING AT 'C'
```

The values are read up to and including the value specified after `THRU/ENDING AT`. In the two examples above, all records with job titles that begin with `A` or `B` are read; if there were a job title `C`, this would also be read, but not the next higher value `CA`.

WHERE Clause

The `WHERE` clause may be used to further qualify which records are to be read.

For instance, if you wanted only those employees with job titles starting from `TRAINEE` who are paid in US currency, you would specify:

```
READ MYVIEW WITH JOB-TITLE = 'TRAINEE'
            WHERE CURR-CODE = 'USD'
```

The `WHERE` clause can also be used with the `BY` clause as follows:

```
READ MYVIEW BY NAME
            WHERE SALARY = 20000
```

The `WHERE` clause differs from the `BY` clause in two respects:

- The field specified in the `WHERE` clause need not be a descriptor.
- The expression following the `WHERE` option is a logical condition.

The following logical operators are possible in a `WHERE` clause:

EQUAL	EQ	=
NOT EQUAL TO	NE	¬=
LESS THAN	LT	<
LESS THAN OR EQUAL TO	LE	<=
GREATER THAN	GT	>
GREATER THAN OR EQUAL TO	GE	>=

The following program illustrates the use of the `STARTING FROM`, `ENDING AT` and `WHERE` clauses:

```
** 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
```

Output of Program `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
```

Further Example of READ Statement

See the following example program:

- ***READX03 - READ statement***

FIND Statement

The following topics are covered:

- Use of FIND Statement
- Basic Syntax of FIND Statement
- Limiting the Number of Records to be Processed
- WHERE Clause
- Example of FIND Statement with WHERE Clause
- IF NO RECORDS FOUND Condition
- Further Examples of FIND Statement

Use of FIND Statement

The `FIND` statement is used to select from a database those records which meet a specified search criterion.

Basic Syntax of FIND Statement

The basic syntax of the `FIND` statement is:

```
FIND RECORDS IN view WITH field = value
```

or shorter:

```
FIND view WITH field = value
```

- where

view	is the name of a view as defined in the `DEFINE DATA` statement and as explained in ***Defining a Database View***.
field	is the name of a database field as defined in that view.

You can only specify a *field* which is defined as a "descriptor" in the underlying **DDM** (it can also be a subdescriptor, superdescriptor, hyperdescriptor or phonetic descriptor).

For the complete syntax, refer to the `FIND` statement documentation.

Limiting the Number of Records to be Processed

In the same way as with the READ statement described **above**, you can limit the number of records to be processed by specifying a number in parentheses after the keyword FIND:

```
FIND (6) RECORDS IN MYVIEW WITH NAME = 'CLEGG'
```

In the above example, only the first 6 records that meet the search criterion would be processed.

Without the limit notation, all records that meet the search criterion would be processed.

Note: If the FIND statement contains a WHERE clause (see below), records which are rejected as a result of the WHERE clause are *not* counted against the limit.

WHERE Clause

With the WHERE clause of the FIND statement, you can specify an additional selection criterion which is evaluated *after* a record (selected with the WITH clause) has been read and *before* any processing is performed on the record.

Example of FIND Statement with WHERE Clause

```
** 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
```

Note: In this example only those records which meet the criteria of the WITH clause *and* the WHERE clause are processed in the DISPLAY statement.

Output of Program 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 Condition

If no records are found that meet the search criteria specified in the `WITH` and `WHERE` clauses, the statements within the `FIND` processing loop are not executed (for the previous example, this would mean that the `DISPLAY` statement would not be executed and consequently no employee data would be displayed).

However, the `FIND` statement also provides an `IF NO RECORDS FOUND` clause, which allows you to specify processing you wish to be performed in the case that no records meet the search criteria.

Example:

```
** 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
```

The above program selects all records in which the field `NAME` contains the value `BLACKSMITH`. For each selected record, the name and first name are displayed. If no record with `NAME = 'BLACKSMITH'` is found on the file, the `WRITE` statement within the `IF NO RECORDS FOUND` clause is executed.

Output of Program `FINDX02`:

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

        NAME              FIRST-NAME
-------------------- --------------------

NO PERSON FOUND.
```

Further Examples of FIND Statement

See the following example programs:

- *FINDX07 - FIND (with several clauses)*
- *FINDX08 - FIND (with LIMIT)*
- *FINDX09 - FIND (using *NUMBER, *COUNTER, *ISN)*
- *FINDX10 - FIND (combined with READ)*
- *FINDX11 - FIND NUMBER (with *NUMBER)*

HISTOGRAM Statement

The following topics are covered:

- Use of HISTOGRAM Statement
- Syntax of HISTOGRAM Statement
- Limiting the Number of Values to be Read
- STARTING/ENDING Clauses
- WHERE Clause
- Example of HISTOGRAM Statement

Use of HISTOGRAM Statement

The `HISTOGRAM` statement is used to either read only the values of one database field, or determine the number of records which meet a specified search criterion.

The `HISTOGRAM` statement does not provide access to any database fields other than the one specified in the `HISTOGRAM` statement.

Syntax of HISTOGRAM Statement

The basic syntax of the HISTOGRAM statement is:

```
HISTOGRAM VALUE IN view FOR field
```

or shorter:

```
HISTOGRAM view FOR field
```

- where

view	is the name of a view as defined in the DEFINE DATA statement and as explained in ***Defining a Database View***.
field	is the name of a database field as defined in that view.

For the complete syntax, refer to the HISTOGRAM statement documentation.

Limiting the Number of Values to be Read

In the same way as with the READ statement, you can limit the number of values to be read by specifying a number in parentheses after the keyword HISTOGRAM:

```
HISTOGRAM (6) MYVIEW FOR NAME
```

In the above example, only the first 6 values of the field NAME would be read.

Without the limit notation, all values would be read.

STARTING/ENDING Clauses

Like the READ statement, the HISTOGRAM statement also provides a STARTING FROM clause and an ENDING AT (or THRU) clause to narrow down the range of values to be read by specifying a starting value and ending value.

Examples:

```
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 Clause

The HISTOGRAM statement also provides a WHERE clause which may be used to specify an additional selection criterion that is evaluated *after* a value has been read and *before* any processing is performed on the value. The field specified in the WHERE clause must be the same as in the main clause of the HISTOGRAM statement.

Example of 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 this program, the system variables *NUMBER and *COUNTER are also evaluated by the HISTOGRAM statement, and output with the DISPLAY statement. *NUMBER contains the number of database records that contain the last value read; *COUNTER contains the total number of values which have been read.

Output of Program 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 Clause

This section covers the multi-fetch record retrieval functionality for Adabas databases.

The multi-fetch functionality described in this section is only supported for databases of type `ADA`, which can be defined in the DBMS Assignments table in the Configuration Utility; see *Database Management System Assignments* in the *Configuration Utility* documentation. With database type ADA2, the multi-fetch clause is not supported.

The following topics are covered:

- Purpose of Multi-Fetch Feature
- Statements Supported
- Considerations for Multi-Fetch Usage

Purpose of Multi-Fetch Feature

In standard mode, Natural does not read multiple records with a single database call; it always operates in a one-record-per-fetch mode. This kind of operation is solid and stable, but can take some time if a large number of database records are being processed. To improve the performance of those programs, you can use multi-fetch processing.

By default, Natural uses single-fetch to retrieve data from Adabas databases. This default can be configured using the Natural profile parameter `MFSET`.

Values `ON` (multi-fetch) and `OFF` (single-fetch) define the default behavior. If `MFSET` is set to `NEVER`, Natural always uses single-fetch mode and ignores any settings at statement level.

The default processing mode can also be overridden at statement level.

Statements Supported

Multi-fetch processing is supported for the following statements that do not involve database modification:

- `FIND`
- `READ`
- `HISTOGRAM`

For more information on the syntax, see the description of the `MULTI-FETCH` clause of the `FIND`, `READ` or `HISTOGRAM` statements.

Considerations for Multi-Fetch Usage

If nested database loops that refer to the same Adabas file contain `UPDATE` statements in one of the inner loops, Natural continues processing the outer loops with the updated values. This implies in multi-fetch mode, that an outer logical `READ` loop has to be repositioned if an inner database loop updates the value of the descriptor that is used for sequence control in the outer loop. If this attempt leads to a conflict for the current descriptor, an error is returned. To avoid this situation, we recommend that you disable multi-fetch in the outer database loops.

In general, multi-fetch mode improves performance when accessing Adabas databases. In some cases, however, it might be advantageous to use single-fetch to enhance performance, especially if database modifications are involved.

Database Processing Loops

This section discusses processing loops required to process data that have been selected from a database as a result of a `FIND`, `READ` or `HISTOGRAM` statement.

The following topics are covered:

- Creation of Database Processing Loops
- Hierarchies of Processing Loops
- Example of Nested FIND Loops Accessing the Same File
- Further Examples of Nested READ and FIND Statements

Creation of Database Processing Loops

Natural automatically creates the necessary processing loops which are required to process data that have been selected from a database as a result of a `FIND`, `READ` or `HISTOGRAM` statement.

Example:

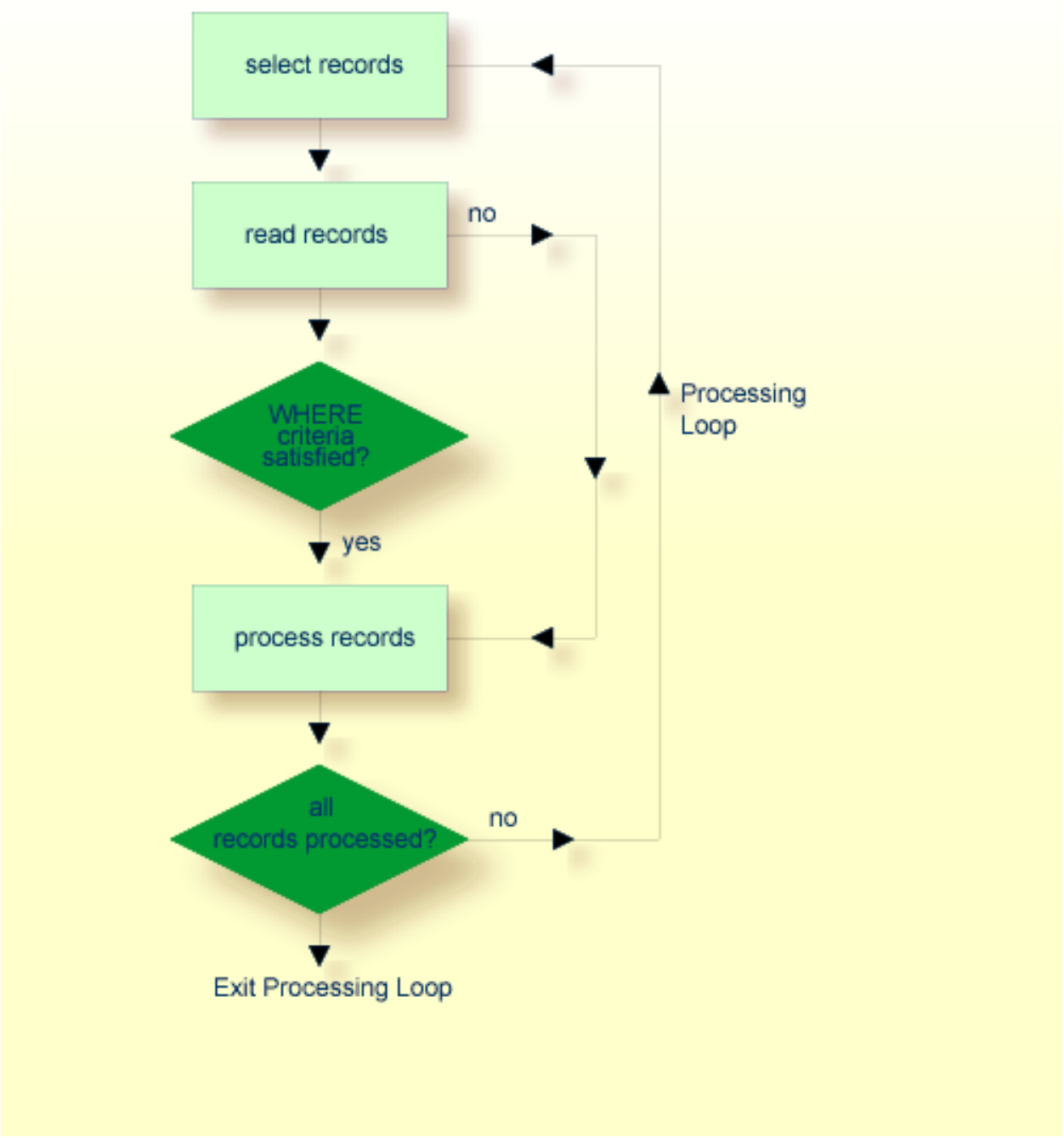
In the following exampe, the `FIND` loop selects all records from the `EMPLOYEES` file in which the field `NAME` contains the value `ADKINSON` and processes the selected records. In this example, the processing consists of displaying certain fields from each record selected.

```
** 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
```

If the FIND statement contained a WHERE clause in addition to the WITH clause, only those records that were selected as a result of the WITH clause *and* met the WHERE criteria would be processed.

The following diagram illustrates the flow logic of a database processing loop:


select records
read records
no
WHERE criteria satisfied?
yes
process records
all records processed?
no
Processing Loop
Exit Processing Loop

Hierarchies of Processing Loops

The use of multiple FIND and/or READ statements creates a hierarchy of processing loops, as shown in the following example:

Example of Processing Loop Hierarchy

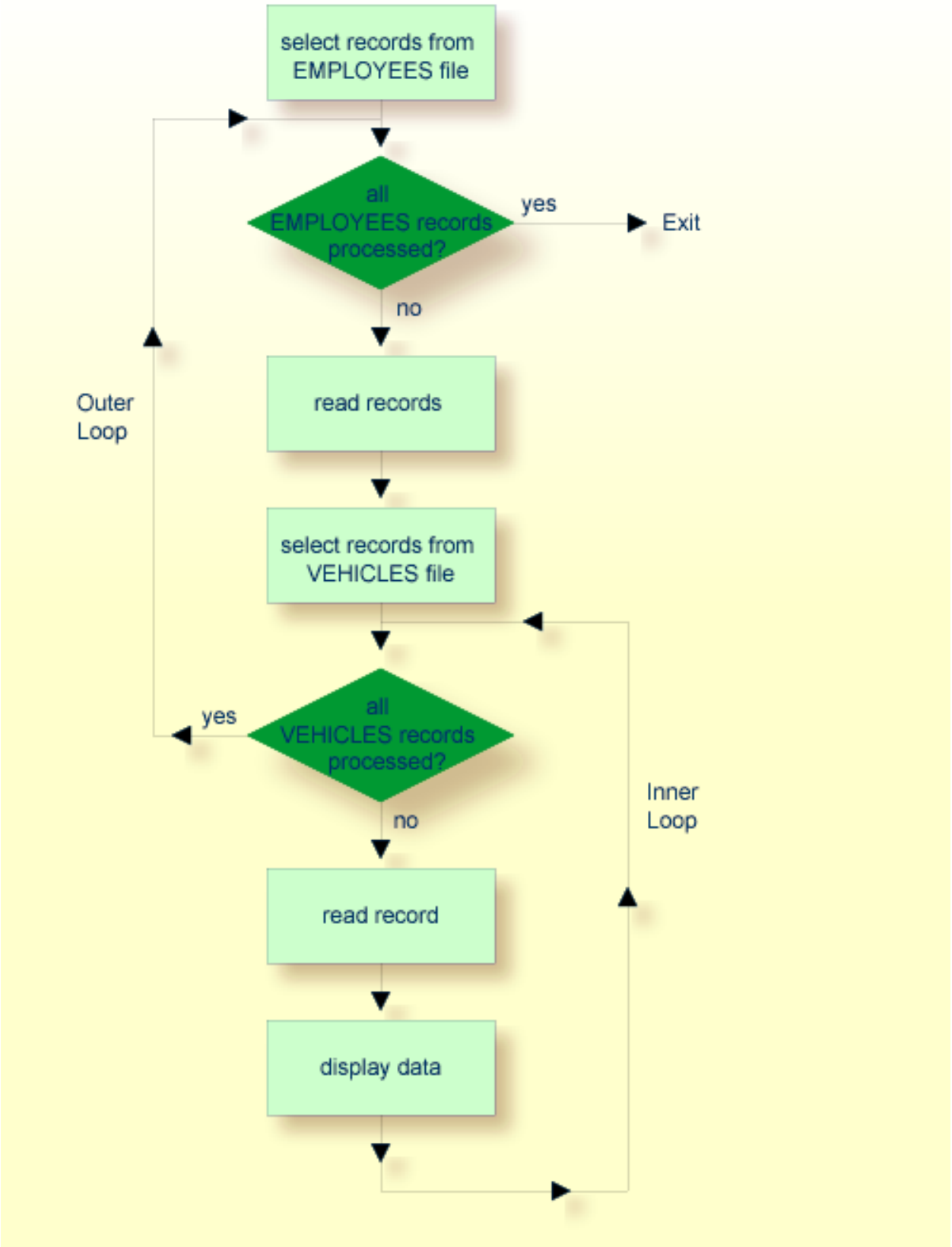
```
** Example 'FINDX04': FIND  (two FIND statements nested)
************************************************************************
DEFINE DATA LOCAL
1 PERSONVIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
1 AUTOVIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
  2 MODEL
END-DEFINE
*
EMP. FIND PERSONVIEW WITH NAME = 'ADKINSON'
  VEH. FIND AUTOVIEW WITH PERSONNEL-ID = PERSONNEL-ID (EMP.)
    DISPLAY NAME MAKE MODEL
  END-FIND
END-FIND
END
```

The above program selects from the EMPLOYEES file all people with the name ADKINSON. Each record (person) selected is then processed as follows:

1. The second FIND statement is executed to select the automobiles from the VEHICLES file, using as selection criterion the PERSONNEL-IDs from the records selected from the EMPLOYEES file with the first FIND statement.
2. The NAME of each person selected is displayed; this information is obtained from the EMPLOYEES file. The MAKE and MODEL of each automobile owned by that person is also displayed; this information is obtained from the VEHICLES file.

The second FIND statement creates an inner processing loop within the outer processing loop of the first FIND statement, as shown in the following diagram.

The diagram illustrates the flow logic of the hierarchy of processing loops in the previous example program:


select records from EMPLOYEES file
all EMPLOYEES records processed?
yes
Exit
no
Outer Loop
read records
select records from VEHICLES file
all VEHICLES records processed?
yes
Inner Loop
no
read record
display data

Example of Nested FIND Loops Accessing the Same File

It is also possible to construct a processing loop hierarchy in which the same file is used at both levels of the hierarchy:

```
** 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
```

The above program first selects all people with name `JONES` and first name `LAUREL` from the `EMPLOYEES` file. Then all who live in the same city are selected from the `EMPLOYEES` file and a list of these people is created. All field values displayed by the `DISPLAY` statement are taken from the second `FIND` statement.

Output of Program `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
```

Further Examples of Nested READ and FIND Statements

See the following example programs:

- *READX04 - READ statement (in combination with FIND and the system variables *NUMBER and *COUNTER)*
- *LIMITX01 - LIMIT statement (for READ, FIND loop processing)*

Database Update - Transaction Processing

This section describes how Natural performs database updating operations based on transactions.

The following topics are covered:

- Logical Transaction
- Record Hold Logic
- Backing Out a Transaction
- Restarting a Transaction
- Example of Using Transaction Data to Restart a Transaction

Logical Transaction

Natural performs database updating operations based on transactions, which means that all database update 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 update statements (`DELETE`, `STORE`, `UPDATE`) involving one or more database files. A logical transaction may also span multiple Natural programs.

A logical transaction begins when a record is put on "hold"; Natural does this automatically when the record is read for updating, for example, if a `FIND` loop contains an `UPDATE` or `DELETE` statement.

The end of a logical transaction is determined by an `END TRANSACTION` statement in the program. This statement ensures that all updates within the transaction have been successfully applied, and releases all records that were put on "hold" during the transaction.

Example:

```
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
END-DEFINE
FIND MYVIEW WITH NAME = 'SMITH'
   DELETE
   END TRANSACTION
END-FIND
END
```

Each record selected would be put on “hold”, deleted, and then - when the `END TRANSACTION` statement is executed - released from “hold”.

Note: The Natural profile parameter `ETEOP`, as set by the Natural administrator, determines whether or not Natural will generate an `END TRANSACTION` statement at the end of each Natural program. Ask your Natural administrator for details.

Example of STORE Statement:

The following example program adds new records to the `EMPLOYEES` file.

```
** 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
```

Output of Program `STOREX01`:

```
ENTER A PERSONNEL ID AND NAME (OR 'END' TO END)

PERSONNEL ID :

NAME         :
FIRST NAME   :
```

Record Hold Logic

If Natural is used with Adabas, any record which is to be updated will be placed in "hold" status until an `END TRANSACTION` or `BACKOUT TRANSACTION` statement is issued or the transaction time limit is exceeded.

When a record is placed in "hold" status for one user, the record is not available for update by another user. Another user who wishes to update the same record will be placed in "wait" status until the record is released from "hold" when the first user ends or backs out his/her transaction.

To prevent users from being placed in wait status, the session parameter `WH` (Wait for Record in Hold Status) can be used (see the *Parameter Reference*).

When you use update logic in a program, you should consider the following:

- The maximum time that a record can be in hold status is determined by the Adabas transaction time limit (Adabas parameter `TT`). If this time limit is exceeded, you will receive an error message and all database modifications done since the last `END TRANSACTION` will be made undone.
- The number of records on hold and the transaction time limit are affected by the size of a transaction, that is, by the placement of the `END TRANSACTION` statement in the program. Restart facilities should be considered when deciding where to issue an `END TRANSACTION`. For example, if a majority of records being processed are *not* to be updated, the `GET` statement is an efficient way of controlling the "holding" of records. This avoids issuing multiple `END TRANSACTION` statements and reduces the number of ISNs on hold. When you process large files, you should bear in mind that the `GET` statement requires an additional Adabas call. An example of a `GET` statement is shown below.

Example of Hold Logic:

```
** 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
```

Backing Out a Transaction

During an active logical transaction, that is, before the `END TRANSACTION` statement is issued, you can cancel the transaction by using a `BACKOUT TRANSACTION` statement. The execution of this statement removes all updates that have been applied (including all records that have been added or deleted) and releases all records held by the transaction.

Restarting a Transaction

With the `END TRANSACTION` statement, you can also store transaction-related information. If processing of the transaction terminates abnormally, you can read this information with a `GET TRANSACTION DATA` statement to ascertain where to resume processing when you restart the transaction.

Example of Using Transaction Data to Restart a Transaction

The following program updates the `EMPLOYEES` and `VEHICLES` files. After a restart operation, the user is informed of the last `EMPLOYEES` record successfully processed. The user can resume processing from that `EMPLOYEES` record. It would also be possible to set up the restart transaction message to include the last `VEHICLES` record successfully updated before the restart operation.

```
** 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
```

Selecting Records Using ACCEPT/REJECT

This section discusses the statements `ACCEPT` and `REJECT` which are used to select records based on user-specified logical criteria.

The following topics are covered:

- Statements Usable with ACCEPT and REJECT
- Example of ACCEPT Statement
- Logical Condition Criteria in ACCEPT/REJECT Statements
- Example of ACCEPT Statement with AND Operator
- Example of REJECT Statement with OR Operator
- Further Examples of ACCEPT and REJECT Statements

Statements Usable with ACCEPT and REJECT

The statements `ACCEPT` and `REJECT` can be used in conjunction with the database access statements:

- `READ`
- `FIND`
- `HISTOGRAM`

Example of 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
```

Output of Program 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                ↩
```

Logical Condition Criteria in ACCEPT/REJECT Statements

The statements ACCEPT and REJECT allow you to specify logical conditions in addition to those that were specified in WITH and WHERE clauses of the READ statement.

The logical condition criteria in the IF clause of an ACCEPT / REJECT statement are evaluated *after* the record has been selected and read.

Logical condition operators include the following (see ***Logical Condition Criteria*** for more detailed information):

EQUAL	EQ	:=
NOT EQUAL TO	NE	¬=
LESS THAN	LT	<
LESS EQUAL	LE	<=
GREATER THAN	GT	>
GREATER EQUAL	GE	>=

Logical condition criteria in ACCEPT / REJECT statements may also be connected with the Boolean operators AND, OR, and NOT. Moreover, parentheses may be used to indicate logical grouping; see the following examples.

Example of ACCEPT Statement with AND Operator

The following program illustrates the use of the Boolean operator AND in an 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
```

Output of Program ACCEPX02:

```
Page      1                                                 04-12-14  12:22:01

        NAME                  CURRENT            ANNUAL
                             POSITION            SALARY
-------------------- ------------------------- ----------

AFANASSIEV           DBA                           42800
ATHERTON             ANALYST                       43000
ATHERTON             MANAGER                       40000                    ↩
```

Example of REJECT Statement with OR Operator

The following program, which uses the Boolean operator `OR` in a `REJECT` statement, produces the same output as the `ACCEPT` statement in the example above, as the logical operators are reversed.

```
** 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
```

Output of Program `ACCEPX03`:

```
Page      1                                                  04-12-14  12:26:27

        NAME                  CURRENT            ANNUAL
                             POSITION            SALARY
-------------------- ------------------------- ----------

AFANASSIEV           DBA                            42800
ATHERTON             ANALYST                        43000
ATHERTON             MANAGER                        40000                    ↩
```

Further Examples of ACCEPT and REJECT Statements

See the following example programs:

- ***ACCEPX04 - ACCEPT IF ... LESS THAN ...***
- ***ACCEPX05 - ACCEPT IF ... AND ...***
- ***ACCEPX06 - REJECT IF ... OR ...***

AT START/END OF DATA Statements

This section discusses the use of the statements AT START OF DATA and AT END OF DATA.

The following topics are covered:

- AT START OF DATA Statement
- AT END OF DATA Statement
- Example of AT START OF DATA and AT END OF DATA Statements
- Further Examples of AT START OF DATA and AT END OF DATA

AT START OF DATA Statement

The AT START OF DATA statement is used to specify any processing that is to be performed after the first of a set of records has been read in a database processing loop.

The AT START OF DATA statement must be placed within the processing loop.

If the AT START OF DATA processing produces any output, this will be output *before the first field value*. By default, this output is displayed left-justified on the page.

AT END OF DATA Statement

The AT END OF DATA statement is used to specify processing that is to be performed after all records for a database processing loop have been processed.

The AT END OF DATA statement must be placed within the processing loop.

If the AT END OF DATA processing produces any output, this will be output *after the last field value*. By default, this output is displayed left-justified on the page.

Example of AT START OF DATA and AT END OF DATA Statements

The following example program illustrates the use of the statements AT START OF DATA and AT END OF DATA.

The Natural system variable *TIME has been incorporated into the AT START OF DATA statement to display the time of day.

The Natural system function OLD has been incorporated into the AT END OF DATA statement to display the name of the last person selected.

```
** 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
```

The program produces the following output:

```
                  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                                    ↩
```

Further Examples of AT START OF DATA and AT END OF DATA

See the following example programs:

- *ATENDX01 - AT END OF DATA*
- *ATSTAX02 - AT START OF DATA*
- *WRITEX09 - WRITE (in combination with AT END OF DATA)*

Unicode Data

Natural enables users to access wide-character fields (format W) in an Adabas database.

The following topics are covered:

- Data Definition Module
- Access Configuration
- Restrictions

Data Definition Module

Adabas wide-character fields (W) are mapped to Natural format U (Unicode).

The length definition for a Natural field of format U corresponds to half the size of the Adabas field of format W. An Adabas wide-character field of length `200` is, for example, mapped to `(U100)` in Natural.

Access Configuration

Natural receives data from Adabas and sends data to Adabas using UTF-16 as common encoding.

This encoding is specified with the `OPRB` parameter and sent to Adabas with the open request. It is used for wide-character fields and applies to the entire Adabas user session.

Restrictions

Wide-character fields (W) of variable length are not supported.

Collating descriptors are not supported.

For further information on Adabas and Unicode support refer to the specific Adabas product documentation.

26 Accessing Data in an SQL Database

- Generating Natural DDMs 262
- Setting Natural Profile Parameters 262
- Natural DML Statements 263
- Natural SQL Statements 269
- Flexible SQL 277
- RDBMS-Specific Requirements and Restrictions 278
- Data-Type Conversion 281
- Date/Time Conversion 281
- Obtaining Diagnostic Information about Database Errors 283
- SQL Authorization 283

This chapter describes how to use Natural with SQL databases via Entire Access. For information about installation and configuration, see *Natural and Entire Access* in the *Database Management System Interfaces* documentation and the separate Entire Access documentation.

Note: On principle, the features and examples contained in the document ***Accessing Data in an Adabas Database*** also apply to the SQL databases supported by Natural. Differences, if any, are described in the documents for the individual database access statements (see the *Statements* documentation) in paragraphs named *Database-Specific Considerations* or in the documents for the individual Natural parameters (see the *Parameter Reference*). In addition, Natural offers a specific set of statements to access SQL databasess.

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.

Note: 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

Note: 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.

Note: 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**.

Note: 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.

Notes:

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.

Note: 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`*.

Note: 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 ]
[ :G ] :host-variable
```

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 DBPROCESSes. DBPROCESSes 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 SELECTions from non-shared temporary tables.

Notes:

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.

Note: 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`

Notes:

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.

Note: 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.

Note: 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},... ]
```

Note: 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.

Note: 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 SYNONYMs 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 INSERTs, UPDATEs and DELETEs 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.

Note: 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.

27 Accessing Data in a Tamino Database

- Prerequisite 286
- DDM and View Definitions with Natural for Tamino 286
- Natural Statements for Tamino Database Access 290
- Natural for Tamino Restrictions 294

The following topics are covered:

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.

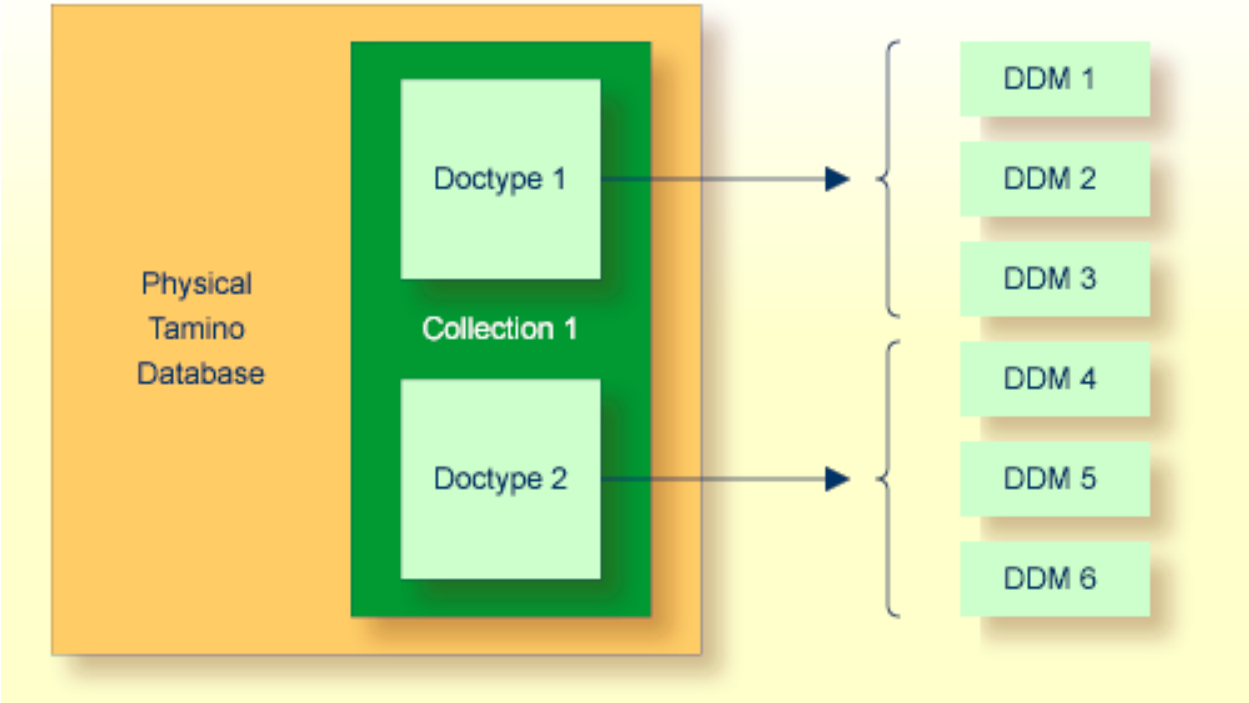
In Tamino, a collection is a container for a set of varying doctypes, so that a collection can be seen as the logical grouping of doctypes that belong together.

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*DDM Editor*, *Using the DDM Editor* 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.

Error Message NAT8400

```
NAT8400 Tamino error ... occurred
```

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
```

Error Message NAT8411

```
NAT8411 HTTP request failed with response code...
```

The error code from the HTTP server is delivered as additional information. See also `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`).

VI Controlling Data Output

This part describes how to proceed if a Natural program is to produce multiple reports. Furthermore, it discusses various aspects of how you can control the format of an output report created with Natural, that is, the way in which the data are displayed.

Report Specification - (*rep*) Notation

Layout of an Output Page

Statements DISPLAY and WRITE

Index Notation for Multiple-Value Fields and Periodic Groups

Page Titles, Page Breaks, Blank Lines

Column Headers

Parameters to Influence the Output of Fields

Edit Masks - EM Parameter

Unicode Edit Masks - EMU Parameter

Vertical Displays

28 Report Specification - (rep) Notation

- Use of Report Specifications 300
- Statements Concerned 300
- Examples of Report Specification 300

`(rep)` is the output report identifier for which a statement is applicable.

Use of Report Specifications

If a Natural program is to produce multiple reports, the notation `(rep)` must be specified with each output statement (see ***Statements Concerned***, below) which is to be used to create output for any report other than the first report (Report 0).

A value of `0 - 31` may be specified.

The value for `(rep)` may also be a logical name which has been assigned using the `DEFINE PRINTER` statement, see ***Example 2*** below.

Statements Concerned

The notation `(rep)` can be used with the following output statements:

`AT END OF PAGE` | `AT TOP OF PAGE` | `DISPLAY` | `EJECT` | `FORMAT` | `NEWPAGE` | `PRINT` | `SKIP` | `SUSPEND IDENTICAL SUPPRESS` | `WRITE` | `WRITE TITLE` | `WRITE TRAILER`

Examples of Report Specification

Example 1 - Multiple Reports

```
DISPLAY (1) NAME ...
WRITE (4) NAME ...
```

Example 2 - Using Logical Names

```
DEFINE PRINTER (LIST=5) OUTPUT 'LPT1'
WRITE (LIST) NAME ...
```

29 Layout of an Output Page

- Statements Influencing a Report Layout 302
- General Layout Example 302

The following topics are covered:

Statements Influencing a Report Layout

The following statements have an impact on the layout of the report:

Statement	Function
`WRITE TITLE`	With this statement, you can specify a page title, that is, text to be output at the top of a page. By default, page titles are centered and not underlined.
`WRITE TRAILER`	With this statement, you can specify a page trailer, that is, text to be output at the bottom of a page. By default, the trailer lines are centered and not underlined.
`AT TOP OF PAGE`	With this statement, you can specify any processing that is to be performed whenever a new page of the report is started. Any output from this processing will be output below the page title.
`AT END OF PAGE`	With this statement, you can specify any processing that is to be performed whenever an end-of-page condition occurs. Any output from this processing will be output below any page trailer (as specified with the `WRITE TRAILER` statement).
`AT START OF DATA`	With this statement, you specify processing that is to be performed after the first record has been read in a database processing loop. Any output from this processing will be output before the first field value. See note below.
`AT END OF DATA`	With this statement, you specify processing that is to be performed after all records for a processing loop have been processed. Any output from this processing will be output immediately after the last field value. See note below.
`DISPLAY / WRITE`	With these statements, you control the format in which the field values that have been read are to be output. See section ***Statements DISPLAY and WRITE***.

Note: The relevance of the statements `AT START OF DATA` and `AT END OF DATA` for the output of data is described under *Database Access*, ***AT START/END OF DATA Statements***. The other statements listed above are discussed in other sections of the part ***Controlling Data Output***.

General Layout Example

The following example program illustrates the general layout of an output page:

```
** 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
```

Output of Program `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 =====
```

30 Statements DISPLAY and WRITE

- DISPLAY Statement 306
- WRITE Statement 307
- Example of DISPLAY Statement 308
- Example of WRITE Statement 308
- Column Spacing - SF Parameter and nX Notation 309
- Tab Setting - nT Notation 310
- Line Advance - Slash Notation 311
- Further Examples of DISPLAY and WRITE Statements 314

The following topics are covered:

DISPLAY Statement

The DISPLAY statement produces output in column format; that is, the values for one field are output in a column underneath one another. If multiple fields are output, that is, if multiple columns are produced, these columns are output next to one another horizontally.

The order in which fields are displayed is determined by the sequence in which you specify the field names in the DISPLAY statement.

The DISPLAY statement in the following program displays for each person first the personnel number, then the name and then the 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
```

Output of Program DISPLX01:

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

PERSONNEL         NAME                  CURRENT
   ID                                   POSITION
--------- -------------------- -------------------------

30020013  GARRET               TYPIST
30016112  TAILOR               WAREHOUSEMAN
20017600  PIETSCH              SECRETARY
```

To change the order of the columns that appear in the output report, simply reorder the field names in the DISPLAY statement. For example, if you prefer to list employee names first, then job titles followed by personnel numbers, the appropriate DISPLAY statement would be:

```
** 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
```

Output of Program DISPLX02:

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

        NAME                  CURRENT          PERSONNEL
                             POSITION             ID
-------------------- ------------------------- ---------

GARRET               TYPIST                    30020013
TAILOR               WAREHOUSEMAN              30016112
PIETSCH              SECRETARY                 20017600
```

A header is output above each column. Various ways to influence this header are described in the document ***Column Headers***.

WRITE Statement

The WRITE statement is used to produce output in free format (that is, not in columns). In contrast to the DISPLAY statement, the following applies to the WRITE statement:

- If necessary, it automatically creates a line advance; that is, a field or text element that does not fit onto the current output line, is automatically output in the next line.
- It does not produce any headers.
- The values of a multiple-value field are output next to one another horizontally, and not underneath one another.

The two example programs shown below illustrate the basic differences between the DISPLAY statement and the WRITE statement.

You can also use the two statements in combination with one another, as described later in the document *Vertical Displays*, ***Combining DISPLAY and WRITE*** .

Example of 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
```

Output of Program `DISPLX03`:

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

        NAME              FIRST-NAME        ANNUAL
                                            SALARY
-------------------- -------------------- ----------

JONES                VIRGINIA                 46000
                                              42300
                                              39300
JONES                MARSHA                   50000
                                              46000
                                              42700
```

Example of 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
```

Output of Program WRITEX01:

```
Page      1                                                  04-11-11  14:15:55

JONES                VIRGINIA                  46000      42300      39300
JONES                MARSHA                    50000      46000      42700
```

Column Spacing - SF Parameter and nX Notation

By default, the columns output with a DISPLAY statement are separated from one another by *one* space.

With the session parameter SF, you can specify the default number of spaces to be inserted between columns output with a DISPLAY statement. You can set the number of spaces to any value from 1 to 30.

The parameter can be specified with a FORMAT statement to apply to the whole report, or with a DISPLAY statement at statement level, but not at element level.

With the *n*X notation in the DISPLAY statement, you can specify the number of spaces (*n*) to be inserted between two columns. An *n*X notation overrides the specification made with the SF parameter.

```
** 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
```

Output of Program DISPLX04:

The above example program produces the following output, where the first two columns are separated by 3 spaces due to the SF parameter in the FORMAT statement, while the second and third columns are separated by 5 spaces due to the notation 5X in the DISPLAY statement:

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

PERSONNEL           NAME                      CURRENT
   ID                                         POSITION
---------   --------------------     -------------------------

30020013    GARRET                   TYPIST
30016112    TAILOR                   WAREHOUSEMAN
20017600    PIETSCH                  SECRETARY
```

The *n*X notation is also available with the WRITE statement to insert spaces between individual output elements:

```
WRITE PERSONNEL-ID 5X NAME 3X JOB-TITLE
```

With the above statement, 5 spaces will be inserted between the fields PERSONNEL-ID and NAME, and 3 spaces between NAME and JOB-TITLE.

Tab Setting - nT Notation

With the *n*T notation, which is available with the DISPLAY and the WRITE statement, you can specify the position where an output element is to be output.

```
** 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
```

Output of Program DISPLX05:

The above program produces the following output, where the field NAME is output starting in the 5th position (counted from the left margin of the page), and the field FIRST-NAME starting in the 30th position:

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

            NAME                  FIRST-NAME
    --------------------     --------------------

    JONES                    VIRGINIA
    JONES                    MARSHA
    JONES                    ROBERT
```

Line Advance - Slash Notation

With a slash (/) in a DISPLAY or WRITE statement, you cause a line advance.

- In a DISPLAY statement, a slash causes a line advance *between fields* and *within text*.
- In a WRITE statement, a slash causes a line advance only when placed *between fields*; within text, it is treated like an ordinary text character.

When placed between fields, the slash must have a blank on either side.

For multiple line advances, you specify multiple slashes.

Example 1 - Line Advance in 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
```

Output of Program DISPLX06:

The above DISPLAY statement produces a line advance after each value of the field NAME and within the text DEPART-MENT:

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

        NAME         DEPART-
     FIRST-NAME       MENT
-------------------- -------

JONES                SALE
VIRGINIA
JONES                MGMT
MARSHA
JONES                TECH
ROBERT
```

Example 2 - Line Advance in 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
```

Output of Program `WRITEX02`:

The above `WRITE` statement produces a line advance after each value of the field `NAME`, and a double line advance after each value of the field `DEPARTMENT`, but none within the text `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
```

Example 3 - Line Advance in DISPLAY and 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
```

Output of Program `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
------------------------------------------------------------------------------
```

Further Examples of DISPLAY and WRITE Statements

See the following example programs:

- *DISPLX13 - DISPLAY (compare with WRITEX08 using WRITE)*
- *WRITEX08 - WRITE (compare with DISPLX13 using DISPLAY)*
- *DISPLX14 - DISPLAY (with AL, SF and nX)*
- *WRITEX09 - WRITE (in combination with AT END OF DATA)*

31 Index Notation for Multiple-Value Fields and Periodic Groups

- Use of Index Notation 316
- Example of Index Notation in DISPLAY Statement 316
- Example of Index Notation in WRITE Statement 317

This chapter describes how you can use the index notation (*n*:*n*) to specify how many values of a multiple-value field or how many occurrences of a periodic group are to be output.

Use of Index Notation

With the index notation (*n*:*n*) you can specify how many values of a multiple-value field or how many occurrences of a periodic group are to be output.

For example, the field INCOME in the DDM EMPLOYEES is a periodic group which keeps a record of the annual incomes of an employee for each year he/she has been with the company.

These annual incomes are maintained in chronological order. The income of the most recent year is in occurrence 1.

If you wanted to have the annual incomes of an employee for the last three years displayed - that is, occurrences 1 to 3 - you would specify the notation (1:3) after the field name in a DISPLAY or WRITE statement (as shown in the following example program).

Example of Index Notation in 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
```

Output of Program DISPLX07:

Note that a DISPLAY statement outputs multiple values of a multiple-value field underneath one another:

```
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
```

As a WRITE statement displays multiple values horizontally instead of vertically, this may cause a line overflow and a - possibly undesired - line advance.

If you use only a single field within a periodic group (for example, SALARY) instead of the entire periodic group, and if you also insert a slash (/) to cause a line advance (as shown in the following example between NAME and JOB-TITLE), the report format becomes manageable.

Example of Index Notation in 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
```

Output of Program 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
```

32 Page Titles, Page Breaks, Blank Lines

- Default Page Title 320
- Suppress Page Title - NOTITLE Option 320
- Define Your Own Page Title - WRITE TITLE Statement 321
- Logical Page and Physical Page 324
- Page Size - PS Parameter 326
- Page Advance 326
- New Page with Title 329
- Page Trailer - WRITE TRAILER Statement 330
- Generating Blank Lines - SKIP Statement 332
- AT TOP OF PAGE Statement 333
- AT END OF PAGE Statement 334
- Further Example 336

This chapter describes various ways of controlling page breaks in a report, the output of page titles at the top of each report page and the generation of empty lines in an output report.

Default Page Title

For each page output via a DISPLAY or WRITE statement, Natural automatically generates a single default title line. This title line contains the page number, the date and the time of day.

Example:

```
WRITE 'HELLO'
END
```

The above program produces the following output with default page title:

```
Page      1                                                    04-12-14  13:19:33

HELLO                                                                          ↩
```

Suppress Page Title - NOTITLE Option

If you wish your report to be output without page titles, you add the keyword NOTITLE to the statement DISPLAY or WRITE.

Example - DISPLAY with 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
```

Output of Program DISPLX20:

```
        NAME              FIRST-NAME              CITY
-------------------- -------------------- --------------------

SHAW                 LESLIE               BOSTON
STANWOOD             VERNON               BOSTON
CREMER               WALT                 BOSTON
PERREAULT            BRENDA               BOSTON
COHEN                JOHN                 BOSTON
```

Example - WRITE with NOTITLE:

```
WRITE NOTITLE 'HELLO'
END
```

The above program produces the following output without page title:

```
HELLO                                                                          ↩
```

Define Your Own Page Title - WRITE TITLE Statement

If you wish a page title of your own to be output instead of the Natural default page title, you use the statement `WRITE TITLE`.

The following topics are covered below:

- Specifying Text for Your Title
- Specifying Empty Lines after the Title
- Title Justification and/or Underlining
- Title with Page Number

Specifying Text for Your Title

With the statement `WRITE TITLE`, you specify the text for your title (in apostrophes).

```
WRITE TITLE 'THIS IS MY PAGE TITLE'
WRITE 'HELLO'
END
```

The above program produces the following output:

```
                            THIS IS MY PAGE TITLE
HELLO                                                                          ↩
```

Specifying Empty Lines after the Title

With the SKIP option of the WRITE TITLE statement, you can specify the number of empty lines to be output immediately below the title line. After the keyword SKIP, you specify the number of empty lines to be inserted.

```
WRITE TITLE 'THIS IS MY PAGE TITLE' SKIP 2
WRITE 'HELLO'
END
```

The above program produces the following output:

```
                            THIS IS MY PAGE TITLE



HELLO                                                                          ↩
```

SKIP is not only available as part of the WRITE TITLE statement, but also as a stand-alone statement.

Title Justification and/or Underlining

By default, the page title is centered on the page and not underlined.

The WRITE TITLE statement provides the following options which can be used independent of each other:

Option	Effect
LEFT JUSTIFIED	Causes the page trailer to be displayed left-justified.
UNDERLINED	Causes the title to be displayed underlined. The underlining runs the width of the line size (see also Natural profile and session parameter LS). By default, titles are underlined with a hyphen (-). However, with the UC session parameter you can specify another character to be used as underlining character (see *Underlining Character for Titles and Headers*).

The following example shows the effect of the LEFT JUSTIFIED and UNDERLINED options:

```
WRITE TITLE LEFT JUSTIFIED UNDERLINED 'THIS IS MY PAGE TITLE'
SKIP 2
WRITE 'HELLO'
END
```

The above program produces the following output:

```
THIS IS MY PAGE TITLE
---------------------------------------------------------------------------



HELLO                                                                      ↩
```

The `WRITE TITLE` statement is executed whenever a new page is initiated for the report.

Title with Page Number

In the following examples, the system variable `*PAGE-NUMBER` is used in conjunction with the `WRITE TITLE` statement to output the page number in the title line.

```
** 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
```

Output of Program `WTITLX01`:

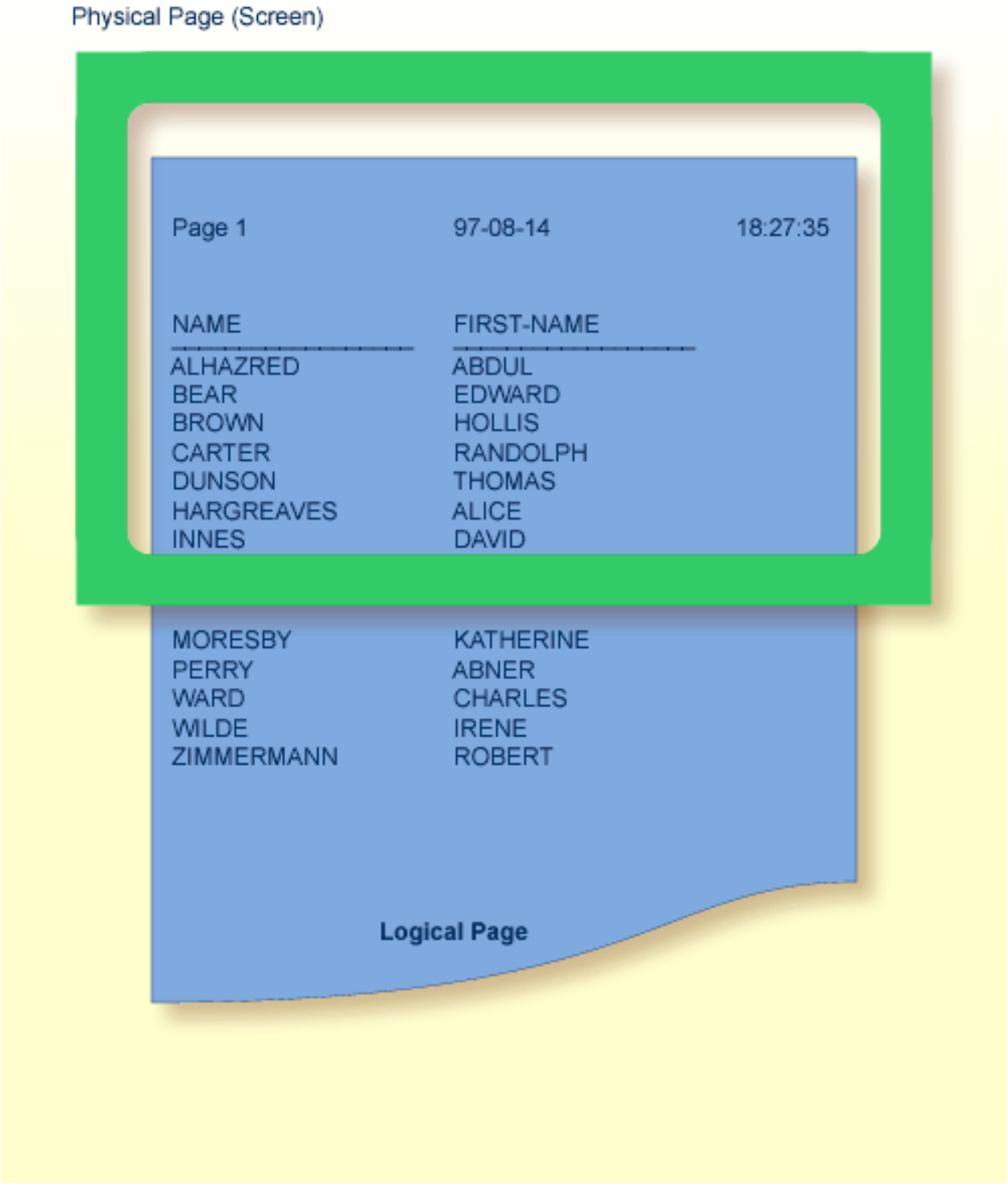
```
YEAR:  1980                PAGE      1
YEAR          MAKE          MAINT-COST
----- -------------------- ----------

 1980 RENAULT                   20000
 1980 RENAULT                   20000
 1980 PEUGEOT                   20000
```

Logical Page and Physical Page

A *logical page* is the output produced by a Natural program. A *physical page* is your terminal screen on which the output is displayed; or it may be the piece of paper on which the output is printed.

The size of the logical page is determined by the number of lines output by the Natural program.

If more lines are output than fit onto one screen, the logical page will exceed the physical screen, and the remaining lines will be displayed on the next screen.



Note: If information you wish to appear at the bottom of the screen (for example, output created by a `WRITE TRAILER` or `AT END OF PAGE` statement) is output on the next screen instead, reduce the logical page size accordingly (with the session parameter `PS`, which is discussed below).

Page Size - PS Parameter

With the parameter PS (Page Size for Natural Reports), you determine the maximum number of lines per (logical) page for a report.

When the number of lines specified with the PS parameter is reached, a page advance occurs (unless page advance is controlled with a NEWPAGE or EJECT statement; see *Page Advance Controlled by EJ Parameter* below).

The PS parameter can be set either at session level with the system command GLOBALS, or within a program with the following statements:

At report level:

- FORMAT PS=*nn*

At statement level:

- DISPLAY (PS=*nn*)
- WRITE (PS=*nn*)
- WRITE TITLE (PS=*nn*)
- WRITE TRAILER (PS=*nn*)
- INPUT (PS=*nn*)

Page Advance

A page advance can be triggered by one of the following methods:

- Page Advance Controlled by EJ Parameter
- Page Advance Controlled by EJECT or NEWPAGE Statements
- Eject/New Page when less than n Lines Left

These methods are discussed below.

Page Advance Controlled by EJ Parameter

With the session parameter EJ (Page Eject), you determine whether page ejects are to be performed or not. By default, EJ=ON applies, which means that page ejects will be performed as specified.

If you specify EJ=OFF, page break information will be ignored. This may be useful to save paper during test runs where page ejects are not needed.

The EJ parameter can be set at session level with the system command GLOBALS; for example:

```
GLOBALS EJ=OFF
```

The EJ parameter setting is overridden by the EJECT statement.

Page Advance Controlled by EJECT or NEWPAGE Statements

The following topics are covered below:

- Page Advance without Title/Header on Next Page
- Page Advance with End/Top-of-Page Processing

Page Advance without Title/Header on Next Page

The EJECT statement causes a page advance *without* a title or header line being generated on the next page. A new physical page is started *without* any top-of-page or end-of-page processing being performed (for example, no WRITE TRAILER or AT END OF PAGE, WRITE TITLE, AT TOP OF PAGE or *PAGE-NUMBER processing).

The EJECT statement overrides the EJ parameter setting.

Page Advance with End/Top-of-Page Processing

The NEWPAGE statement causes a page advance *with* associated end-of-page and top-of-page processing. A trailer line will be displayed, if specified. A title line, either default or user-specified, will be displayed on the new page, unless the NOTITLE option has been specified in a DISPLAY or WRITE statement (as described **above**).

If the NEWPAGE statement is not used, page advance is automatically controlled by the setting of the PS parameter; see ***Page Size - PS Parameter*** above).

Eject/New Page when less than n Lines Left

Both the NEWPAGE statement and the EJECT statement provide a WHEN LESS THAN *n* LINES LEFT option. With this option, you specify a number of lines (*n*). The NEWPAGE/EJECT statement will then be executed if - at the time the statement is processed - less than *n* lines are available on the current page.

Example 1:

```
FORMAT PS=55
...
NEWPAGE WHEN LESS THAN 7 LINES LEFT
...
```

In this example, the page size is set to 55 lines.

If only 6 or less lines are left on the current page at the time when the NEWPAGE statement is processed, the NEWPAGE statement is executed and a page advance occurs. If 7 or more lines are left, the NEWPAGE statement is not executed and no page advance occurs; the page advance then occurs depending on the session parameter PS (Page Size for Natural Reports), that is, after 55 lines.

Example 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
```

New Page with Title

The NEWPAGE statement also provides a WITH TITLE option. If this option is not used, a default title will appear at the top of the new page or a WRITE TITLE statement or NOTITLE clause will be executed.

The WITH TITLE option of the NEWPAGE statement allows you to override these with a title of your own choice. The syntax of the WITH TITLE option is the same as for the WRITE TITLE statement.

Example:

```
NEWPAGE WITH TITLE LEFT JUSTIFIED 'PEOPLE LIVING IN BOSTON:'
```

The following program illustrates the use of the session parameter PS (Page Size for Natural Reports) and the NEWPAGE statement. Moreover, the system variable *PAGE-NUMBER is used to display the current page number.

```
** 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
```

Output of Program NEWPAX01:

Note the position of the page breaks and the title line:

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

        NAME           DEPT        LOCATION
-------------------- ------ --------------------

FICKEN               TECH10 MADISON
KELLOGG              TECH10 MADISON
ALEXANDER            SALE20 MADISON
```

Page 2:

```
EMPLOYEES BY CITY - PAGE:      2
        NAME           DEPT        LOCATION
-------------------- ------ --------------------

DE JUAN              SALE03 MADRID
DE LA MADRID         PROD01 MADRID
```

Page 3:

```
EMPLOYEES BY CITY - PAGE: 3
```

Page Trailer - WRITE TRAILER Statement

The following topics are covered below:

- Specifying a Page Trailer
- Considering Logical Page Size
- Page Trailer Justification and/or Underlining

Specifying a Page Trailer

The `WRITE TRAILER` statement is used to output text (in apostrophes) at the bottom of a page.

```
WRITE TRAILER 'THIS IS THE END OF THE PAGE'
```

The statement is executed when an end-of-page condition is detected, or as a result of a `SKIP` or `NEWPAGE` statement.

Considering Logical Page Size

As the end-of-page condition is checked only *after* an entire DISPLAY or WRITE statement has been processed, it may occur that the logical page size (that is, the number of lines output by a DISPLAY or WRITE statement) causes the physical size of the output page to be exceeded before the WRITE TRAILER statement is executed.

To ensure that a page trailer actually appears at the bottom of a physical page, you should set the logical page size (with the PS session parameter) to a value less than the physical page size.

Page Trailer Justification and/or Underlining

By default, the page trailer is displayed centered on the page and not underlined.

The WRITE TRAILER statement provides the following options which can be used independent of each other:

Option	Effect
LEFT JUSTIFIED	Causes the page trailer to be displayed left justified.
UNDERLINED	The underlining runs the width of the line size (see also Natural profile and session parameter LS). By default, titles are underlined with a hyphen (-). However, with the UC session parameter you can specify another character to be used as underlining character (see *Underlining Character for Titles and Headers*).

The following examples show the use of the LEFT JUSTIFIED and UNDERLINED options of the WRITE TRAILER statement:

Example 1:

```
WRITE TRAILER LEFT JUSTIFIED UNDERLINED 'THIS IS THE END OF THE PAGE'
```

Example 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
```

Generating Blank Lines - SKIP Statement

The `SKIP` statement is used to generate one or more blank lines in an output report.

Example 1 - SKIP in conjunction with WRITE and 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
```

Example 2 - SKIP in conjunction with 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

The `AT TOP OF PAGE` statement is used to specify any processing that is to be performed whenever a new page of the report is started.

If the `AT TOP OF PAGE` processing produces any output, this will be output below the page title (with a skipped line in between).

By default, this output is displayed left-justified on the page.

Example:

```
** 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
```

Output of Program 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

The AT END OF PAGE statement is used to specify any processing that is to be performed whenever an end-of-page condition occurs.

If the AT END OF PAGE processing produces any output, this will be output after any **page trailer** (as specified with the WRITE TRAILER statement).

By default, this output is displayed left-justified on the page.

The same considerations **described above** for page trailers regarding physical and logical page sizes and the number of lines output by a `DISPLAY` or `WRITE` statement also apply to `AT END OF PAGE` output.

Example:

```
** 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
```

Output of Program `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
           ↩
```

Further Example

See the following example program:

- *DISPLX21 - DISPLAY (with slash '/' and compare with WRITE)*

33 Column Headers

- Default Column Headers 338
- Suppress Default Column Headers - NOHDR Option 338
- Define Your Own Column Headers 339
- Combining NOTITLE and NOHDR 340
- Centering of Column Headers - HC Parameter 340
- Width of Column Headers - HW Parameter 340
- Filler Characters for Headers - Parameters FC and GC 341
- Underlining Character for Titles and Headers - UC Parameter 342
- Suppressing Column Headers - Slash Notation 343
- Further Examples of Column Headers 344

The following topics are covered:

Default Column Headers

By default, each database field output with a DISPLAY statement is displayed with a default column header (which is defined for the field in the DDM).

```
** 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
```

Output of Program DISPLX01:

The above example program uses default headers and produces the following output.

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

PERSONNEL         NAME                  CURRENT
   ID                                   POSITION
--------- -------------------- ------------------------

30020013  GARRET               TYPIST
30016112  TAILOR               WAREHOUSEMAN
20017600  PIETSCH              SECRETARY
```

Suppress Default Column Headers - NOHDR Option

If you wish your report to be output without column headers, add the keyword NOHDR to the DISPLAY statement.

```
DISPLAY NOHDR PERSONNEL-ID NAME JOB-TITLE
```

Define Your Own Column Headers

If you wish column headers of your own to be output instead of the default headers, you specify '*text*' (in apostrophes) immediately before a field, *text* being the header to be used for the field.

```
** 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
```

Output of Program DISPLX08:

The above program contains the header EMPLOYEE for the field NAME, and the header POSITION for the field JOB-TITLE; for the field PERSONNEL-ID, the default header is used. The program produces the following output:

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

PERSONNEL       EMPLOYEE               POSITION
   ID
--------- -------------------- -------------------------

30020013  GARRET               TYPIST
30016112  TAILOR               WAREHOUSEMAN
20017600  PIETSCH              SECRETARY
```

Combining NOTITLE and NOHDR

To create a report that has neither page title nor column headers, you specify the NOTITLE and NOHDR options together in the following order:

```
DISPLAY NOTITLE NOHDR PERSONNEL-ID NAME JOB-TITLE
```

Centering of Column Headers - HC Parameter

By default, column headers are centered above the columns. With the HC parameter, you can influence the placement of column headers.

If you specify

HC=L	headers will be left-justified.
HC=R	headers will be right-justified.
HC=C	headers will be centered.

The HC parameter can be used in a FORMAT statement to apply to the whole report, or it can be used in a DISPLAY statement at both statement level and element level, for example:

```
DISPLAY (HC=L) PERSONNEL-ID NAME JOB-TITLE
```

Width of Column Headers - HW Parameter

With the HW parameter, you determine the width of a column output with a DISPLAY statement.

If you specify

HW=ON	the width of a DISPLAY column is determined by either the length of the header text or the length of the field, whichever is longer. This also applies by default.
HW=OFF	the width of a DISPLAY column is determined only by the length of the field. However, HW=OFF only applies to DISPLAY statements which do *not* create headers; that is, either a first DISPLAY statement with NOHDR option or a subsequent DISPLAY statement.

The HW parameter can be used in a FORMAT statement to apply to the entire report, or it can be used in a DISPLAY statement at both statement level and element (field) level.

Filler Characters for Headers - Parameters FC and GC

With the FC parameter, you specify the *filler character* which will appear on either side of a *header* produced by a DISPLAY statement across the full column width if the column width is determined by the field length and not by the header (see HW parameter **above**); otherwise FC will be ignored.

When a group of fields or a periodic group is output via a DISPLAY statement, a *group header* is displayed across all field columns that belong to that group above the headers for the individual fields within the group. With the GC parameter, you can specify the *filler character* which will appear on either side of such a group header.

While the FC parameter applies to the headers of individual fields, the GC parameter applies to the headers for groups of fields.

The parameters FC and GC can be specified in a FORMAT statement to apply to the whole report, or they can be specified in a DISPLAY statement at both statement level and element (field) level.

```
** 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
```

Output of Program 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
```

Underlining Character for Titles and Headers - UC Parameter

By default, titles and headers are underlined with a hyphen (-).

With the UC parameter, you can specify another character to be used as underlining character.

The UC parameter can be specified in a FORMAT statement to apply to the whole report, or it can be specified in a DISPLAY statement at both statement level and element (field) level.

```
** 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
```

In the above program, the UC parameter is specified at program level and at element (field) level: the underlining character specified with the FORMAT statement (=) applies for the whole report - except for the field PERSONNEL-ID, for which a different underlining character (*) is specified.

Output of Program FORMAX02:

```
EMPLOYEES REPORT
================================================================================

PERSONNEL         NAME                 CURRENT
   ID                                 POSITION
********* ==================== =========================

30020013  GARRET               TYPIST
30016112  TAILOR               WAREHOUSEMAN
20017600  PIETSCH              SECRETARY
```

Suppressing Column Headers - Slash Notation

With the notation apostrophe-slash-apostrophe ('/'), you can suppress default column headers for individual fields displayed with a `DISPLAY` statement. While the `NOHDR` option suppresses the headers of all columns, the notation `'/'` can be used to suppress the header for an individual column.

The apostrophe-slash-apostrophe ('/') notation is specified in the `DISPLAY` statement immediately before the name of the field for which the column header is to be suppressed.

Compare the following two examples:

Example 1:

```
DISPLAY NAME PERSONNEL-ID JOB-TITLE
```

In this case, the default column headers of all three fields will be displayed:

```
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          ↩
```

Example 2:

```
DISPLAY '/' NAME PERSONNEL-ID JOB-TITLE
```

In this case, the notation `'/'` causes the column header for the field `NAME` to be suppressed:

```
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     ↩
```

Further Examples of Column Headers

See the following example programs:

- ***DISPLX15 - DISPLAY (with FC, UC)***
- ***DISPLX16 - DISPLAY (with '/', 'text', 'text/text')***

34 Parameters to Influence the Output of Fields

- Overview of Field-Output-Relevant Parameters 346
- Leading Characters - LC Parameter 346
- Unicode Leading Characters - LCU Parameter 347
- Insertion Characters - IC Parameter 347
- Unicode Insertion Characters - ICU Parameter 348
- Trailing Characters - TC Parameter 348
- Unicode Trailing Characters - TCU Parameter 348
- Output Length - AL and NL Parameters 349
- Display Length for Output - DL Parameter 349
- Sign Position - SG Parameter 351
- Identical Suppress - IS Parameter 353
- Zero Printing - ZP Parameter 355
- Empty Line Suppression - ES Parameter 355
- Further Examples of Field-Output-Relevant Parameters 357

This chapter discusses the use of those Natural profile and/or session parameters which you can use to control the output format of fields.

Overview of Field-Output-Relevant Parameters

Natural provides several profile and/or session parameters you can use to control the format in which fields are output:

Parameter	Function
LC, IC and TC	With these session parameters, you can specify characters that are to be displayed before or after a field or before a field value.
LCU, ICU and TCU	With these session parameters, you can specify characters in Unicode format that are to be displayed before or after a field or before a field value.
AL and NL	With these session parameters, you can increase or reduce the output length of fields.
DL	With this session parameter, you can specify the default output length for an alphanumeric map field of format U.
SG	With this session parameter, you can determine whether negative values are to be displayed with or without a minus sign.
IS	With this session parameter, you can suppress the display of subsequent identical field values.
ZP	With this profile and session parameter, you can determine whether field values of 0 are to be displayed or not.
ES	With this session parameter, you can suppress the display of empty lines generated by a DISPLAY or WRITE statement.

These parameters are discussed in the following sections.

Leading Characters - LC Parameter

With the session parameter LC, you can specify leading characters that are to be displayed immediately *before a field* that is output with a DISPLAY statement. The width of the output column is enlarged accordingly. You can specify 1 to 10 characters.

By default, values are displayed left-justified in alphanumeric fields and right-justified in numeric fields. (These defaults can be changed with the AD parameter; see the *Parameter Reference*). When a leading character is specified for an alphanumeric field, the character is therefore displayed immediately before the field value; for a numeric field, a number of spaces may occur between the leading character and the field value.

The LC parameter can be used with the following statements:

- FORMAT
- DISPLAY

The LC parameter can be set at statement level and at element level.

Unicode Leading Characters - LCU Parameter

The session parameter LCU is identical to the session parameter LC. The difference is that the leading characters are always stored in Unicode format.

This allows you to specify leading characters with mixed characters from different code pages, and assures that always the correct character is displayed independent of the installed system code page.

For further information, see *Unicode and Code Page Support in the Natural Programming Language*, *Session Parameters*, section *EMU, ICU, LCU, TCU versus EM, IC, LC, TC*.

The parameters LCU and ICU cannot both be applied to one field.

Insertion Characters - IC Parameter

With the session parameter IC, you specify the characters to be inserted in the column immediately *preceding the value of a field* that is output with a DISPLAY statement. You can specify 1 to 10 characters.

For a numeric field, the insertion characters will be placed immediately before the first significant digit that is output, with no intervening spaces between the specified character and the field value. For alphanumeric fields, the effect of the IC parameter is the same as that of the LC parameter.

The parameters LC and IC cannot both be applied to one field.

The IC parameter can be used with the following statements:

- FORMAT
- DISPLAY

The IC parameter can be set at statement level and at element level.

Unicode Insertion Characters - ICU Parameter

The session parameter `ICU` is identical to the session parameter `IC`. The difference is that the insertion characters are always stored in Unicode format.

This allows you to specify insertion characters with mixed characters from different code pages, and assures that always the correct character is displayed independent of the installed system code page.

For further information, see *Unicode and Code Page Support in the Natural Programming Language*, *Session Parameters*, section *EMU, ICU, LCU, TCU versus EM, IC, LC, TC*.

The parameters `LCU` and `ICU` cannot both be applied to one field.

Trailing Characters - TC Parameter

With the session parameter `TC`, you can specify trailing characters that are to be displayed immediately *to the right of a field* that is output with a `DISPLAY` statement. The width of the output column is enlarged accordingly. You can specify 1 to 10 characters.

The `TC` parameter can be used with the following statements:

- `FORMAT`
- `DISPLAY`

The `TC` parameter can be set at statement level and at element level.

Unicode Trailing Characters - TCU Parameter

The session parameter `TCU` is identical to the session parameter `TC`. The difference is that the trailing characters are always stored in Unicode format.

This allows you to specify trailing characters with mixed characters from different code pages, and assures that always the correct character is displayed independent of the installed system code page.

For further information, see *Unicode and Code Page Support in the Natural Programming Language*, *Session Parameters*, section *EMU, ICU, LCU, TCU versus EM, IC, LC, TC*.

Output Length - AL and NL Parameters

With the session parameter AL, you can specify the *output length for an alphanumeric field*; with the NL parameter, you can specify the *output length for a numeric field*. This determines the length of a field as it will be output, which may be shorter or longer than the actual length of the field (as defined in the DDM for a database field, or in the DEFINE DATA statement for a user-defined variable).

Both parameters can be used with the following statements:

- FORMAT
- DISPLAY
- WRITE
- PRINT
- INPUT

Both parameters can be set at statement level and at element level.

Note: If an edit mask is specified, it overrides an NL or AL specification. **Edit masks** are described in ***Edit Masks - EM Parameter***.

Display Length for Output - DL Parameter

With the session parameter DL, you can specify the *display length for a field of format A or U*, since the display width of a Unicode string can be twice the length of the string, and the user must be able to display the whole string. The default will be the length, for example, for a format/length U10, the display length can be 10 to 20, whereas the default length (when DL is not specified) is 10.

The session parameter DL can be used with the following statements:

- FORMAT
- DISPLAY
- WRITE
- PRINT
- INPUT

The session parameter DL can be set at statement level and at element level.

The difference between the session parameters AL and DL is that AL defines the data length of a field whereas DL defines the number of columns which are used on the screen for displaying the field. The user can scroll in input fields to view the entire content of a field if the value specified with the DL session parameter is less than the length of the field data.

Note: DL is allowed for A-format fields as well. This would allow making the edit control size smaller than the content of a field.

Example:

```
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
```

The above program produces the following output where the content of the field #U2 is incomplete:

```
#U1 latintxt00 #U2 特別是伺服
```

When the session parameter DL is used with the field #U2 and is specified accordingly, the content of this field will be displayed correctly:

```
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
```

Result:

```
#U1 latintxt00 #U2 特别是伺服器都需要支
```

Sign Position - SG Parameter

With the session parameter SG, you can determine whether or not a sign position is to be allocated for numeric fields.

- By default, SG=ON applies, which means that a sign position is allocated for numeric fields.
- If you specify SG=OFF, negative values in numeric fields will be output without a minus sign (-).

The SG parameter can be used with the following statements:

- FORMAT
- DISPLAY
- PRINT
- WRITE
- INPUT

The SG parameter can be set at both statement level and element level.

Note: If an edit mask is specified, it overrides an SG specification. **Edit masks** are described in *Edit Masks - EM Parameter*.

Example Program without Parameters

```
** 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
```

The above program contains no parameter settings and produces the following output:

```
  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
```

Example Program with Parameters AL, NL, LC, IC and TC

In this example, the session parameters `AL`, `NL`, `LC`, `IC` and `TC` are used.

```
** 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
```

The above program produces the following output. Compare the layout of this output with that of the previous program to see the effect of the individual parameters:

```
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
```

As you can see in the above example, any output length you specify with the AL or NL parameter does not include any characters specified with the LC, IC and TC parameters: the width of the NAME column, for example, is 11 characters - 10 for the field value (AL=10) plus 1 leading character.

The width of the SALARY and BONUS columns is 8 characters - 6 for the field value (NL=6), plus 1 leading/inserted character, plus 1 sign position (because SG=ON applies).

Identical Suppress - IS Parameter

With the session parameter IS, you can suppress the display of identical information in successive lines created by a WRITE or DISPLAY statement.

- By default, IS=OFF applies, which means that identical field values will be displayed.
- If IS=ON is specified, a value which is identical to the previous value of that field will not be displayed.

The IS parameter can be specified

- with a FORMAT statement to apply to the whole report, or
- in a DISPLAY or WRITE statement at both statement level and element level.

The effect of the parameter IS=ON can be suspended for one record by using the statement SUSPEND IDENTICAL SUPPRESS; see the *Statements* documentation for details.

Compare the output of the following two example programs to see the effect of the IS parameter. In the second one, the display of identical values in the NAME field is suppressed.

Example Program without 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
```

The above program produces the following output:

```
  Page      1                                                  04-11-11  11:11:11

           NAME              FIRST-NAME
  ------------------- --------------------

  JONES                VIRGINIA
  JONES                MARSHA
  JONES                ROBERT             ↩
```

Example Program with 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
```

The above program produces the following output:

```
Page      1                                                  04-11-11  11:54:01

        NAME              FIRST-NAME
-------------------- --------------------

JONES                VIRGINIA
                     MARSHA
                     ROBERT
```

Zero Printing - ZP Parameter

With the profile and session parameter ZP, you determine how a field value of zero is to be displayed.

- By default, ZP=ON applies, which means that one 0 (for numeric fields) or all zeros (for time fields) will be displayed for each field value that is zero.
- If you specify ZP=OFF, the display of each field value which is zero will be suppressed.

The ZP parameter can be specified

- with a FORMAT statement to apply to the whole report, or
- in a DISPLAY or WRITE statement at both statement level and element level.

Compare the output of the following two **example programs** to see the effect of the parameters ZP and ES.

Empty Line Suppression - ES Parameter

With the session parameter ES, you can suppress the output of empty lines created by a DISPLAY or WRITE statement.

- By default, ES=OFF applies, which means that lines containing all blank values will be displayed.
- If ES=ON is specified, a line resulting from a DISPLAY or WRITE statement which contains all blank values will not be displayed. This is particularly useful when displaying multiple-value fields or fields which are part of a periodic group if a large number of empty lines are likely to be produced.

The ES parameter can be specified

- with a FORMAT statement to apply to the whole report, or
- in a DISPLAY or WRITE statement at statement level.

Note: To achieve empty suppression for numeric values, in addition to `ES=ON` the parameter `ZP=OFF` must also be set for the fields concerned in order to have null values turned into blanks and thus not output either.

Compare the output of the following two example programs to see the effect of the parameters `ZP` and `ES`.

Example Program without Parameters ZP and 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
```

The above program produces the following output:

```
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
```

Example Program with Parameters ZP and 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
```

The above program produces the following output:

```
Page      1                                                 04-11-11  11:59:09

        NAME              FIRST-NAME        BONUS
-------------------- -------------------- ----------

JONES                VIRGINIA                  9000
                                               6750
JONES                MARSHA
JONES                ROBERT
JONES                LILLY
```

Further Examples of Field-Output-Relevant Parameters

For further examples of the parameters `LC`, `IC`, `TC`, `AL`, `NL`, `IS`, `ZP` and `ES`, and the `SUSPEND IDENTICAL SUPPRESS` statement, see the following example programs:

- ***DISPLX17 - DISPLAY (with NL, AL, IC, LC, TC)***
- ***DISPLX18 - DISPLAY (using default settings for SF, AL, UC, LC, IC, TC and compare with DISPLX19)***
- ***DISPLX19 - DISPLAY (with SF, AL, LC, IC, TC and compare with DISPLX18)***
- ***SUSPEX01 - SUSPEND IDENTICAL SUPPRESS (in conjunction with parameters IS, ES, ZP in DISPLAY)***
- ***SUSPEX02 - SUSPEND IDENTICAL SUPPRESS (in conjunction with parameters IS, ES, ZP in DISPLAY). Identical to SUSPEX01, but with IS=OFF.***

- ***COMPRX03 - COMPRESS***

35 Code Page Edit Masks - EM Parameter

- Use of EM Parameter 360
- Edit Masks for Numeric Fields 360
- Edit Masks for Alphanumeric Fields 361
- Length of Fields 361
- Edit Masks for Date and Time Fields 362
- Customizing Separator Character Displays 362
- Examples of Edit Masks 364
- Further Examples of Edit Masks 366

This chapter describes how you can specify an edit mask for an alphanumeric or numeric field.

Use of EM Parameter

With the session parameter EM you can specify an edit mask for an alphanumeric or numeric field, that is, determine character by character the format in which the field values are to be output. Using the session parameter EMU, you can define edit masks with Unicode characters in the same way as described below for the EM session parameter.

Example:

```
DISPLAY NAME (EM=X^X^X^X^X^X^X^X^X^X)
```

In this example, each X represents one character of an alphanumeric field value to be displayed, and each ^ represents a blank. If displayed via the DISPLAY statement, the name JOHNSON would appear as follows:

```
J O H N S O N
```

You can specify the session parameter EM

- at report level (in a FORMAT statement),
- at statement level (in a DISPLAY, WRITE, INPUT, MOVE EDITED or PRINT statement) or
- at element level (in a DISPLAY, WRITE or INPUT statement).

An edit mask specified with the session parameter EM will override a default edit mask specified for a field in the **DDM**; see *Using the DDM Editor*, *Specifying Extended Field Attributes*.

If EM=OFF is specified, no edit mask at all will be used.

An edit mask specified at statement level will override an edit mask specified at report level.

An edit mask specified at element level will override an edit mask specified at statement level.

Edit Masks for Numeric Fields

An edit mask specified for a field of format N, P, I, or F must contain at least one 9 or Z. If more nines or Zs exist, the number of positions contained in the field value, the number of print positions in the edit mask will be adjusted to the number of digits defined for the field value. If fewer nines or Zs exist, the high-order digits before the decimal point and/or low-order digits after the decimal point will be truncated.

For further information, see session parameter EM, *Edit Masks for Numeric Fields* in the *Parameter Reference* documentation.

Edit Masks for Alphanumeric Fields

Edit masks for alphanumeric fields must include an X for each alphanumeric character that is to be output.

With a few exceptions, you may add leading, trailing and insertion characters (with or without enclosing them in apostrophes).

The circumflex character (^) is used to insert blanks in edit mask for both numeric and alphanumeric fields.

For further information, see session parameter EM, *Edit Masks for Alphanumeric Fields* in the *Parameter Reference* documentation.

Length of Fields

It is important to be aware of the length of the field to which you assign an edit mask.

- If the edit mask is longer than the field, this will yield unexpected results.
- If the edit mask is shorter than the field, the field output will be truncated to just those positions specified in the edit mask.

Examples:

Assuming an alphanumeric field that is 12 characters long and the field value to be output is JOHNSON, the following edit masks will yield the following results:

Edit Mask	Output
EM=X.X.X.X.X	J.O.H.N.S
EM=****XXXXXX****	****JOHNSO**

Edit Masks for Date and Time Fields

Edit masks for date fields can include the characters `D` (day), `M` (month) and `Y` (year) in various combinations.

Edit masks for time fields can include the characters `H` (hour), `I` (minute), `S` (second) and `T` (tenth of a second) in various combinations.

In conjunction with edit masks for date and time fields, see also the date and time system variables.

Customizing Separator Character Displays

Natural programs are used in business applications all over the world. Depending on the local conventions, it is usual to present numeric data fields and those with a date or time content in a special output style, when displayed in I/O statements. The different appearance should not be realized by alternate program coding that is processed selectively as a function of the locale where the program is being executed, but should be carried out with the same program image in conjunction with a set of runtime parameters to specify the decimal point character and the "thousands separator character".

The following topics are covered below:

- Decimal Separator
- Dynamic Thousands Separator
- Examples

Decimal Separator

The Natural parameter `DC` is available to specify the character to be inserted in place of any characters used to represent the decimal separator (also called "radix" character) in edit masks. This parameter enables the users of a Natural program or application to choose any (special) character to separate the integer positions from the decimal positions of a numeric data item and enables, for example, U.S. shops to use the decimal point (.) and European shops to use the comma (,).

Dynamic Thousands Separator

To structure the output of a large integer values, it is common practice to insert separators between every three digits of an integer to separate groups of thousands. This separator is called a “thousands separator”. For example, shops in the United States generally use a comma for this purpose (1,000,000), whereas shops in Germany use the period (1.000.000), in France a space (1 000 000), etc.

In a Natural edit mask, a “dynamic thousands separator” is a comma (or period) indicating the position where thousands separator characters (defined with the `THSEPCH` parameter) are inserted at runtime. At compile time, the Natural profile parameter `THSEP` or the option `THSEP` of system command `COMPOPT` enables or disables the interpretation of the comma (or period) as dynamic thousands separator.

If `THSEP` is set to `OFF` (default), any character used as thousands separator in the edit mask is treated as literal and displayed unchanged at runtime. This setting retains downwards compatibility.

If `THSEP` is set to `ON`, any comma (or period) in the edit mask is interpreted as dynamic thousands separators. In general, the dynamic thousands separator is a comma, but if the comma is already in use as decimal character (`DC`), the period is used as dynamic thousands separator.

At runtime the dynamic thousands separators are replaced by the current value of the `THSEPCH` parameter (thousands separator character).

Examples

A Natural program that is cataloged with parameter settings `DC='.'` and `THSEP=ON` uses the edit mask (`EM=ZZ,ZZZ,ZZ9.99`).

Parameter Settings at Runtime	Displays as
`DC='.'` and `THSEPCH=','`	`1,234,567.89`
`DC=','` and `THSEPCH='.'`	`1.234.567,89`
`DC=','` and `THSEPCH='/'`	`1/234/567,89`
`DC=','` and `THSEPCH=' '`	`1 234 567,89`
`DC=','` and `THSEPCH=''''`	`1'234'567,89`

Examples of Edit Masks

Some examples of edit masks, along with possible output they produce, are provided below.

In addition, the abbreviated notation for each edit mask is given. You can use either the abbreviated or the long notation.

Edit Mask	Abbreviation	Output A	Output 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

* Use a date system variable.

** Use a time system variable.

For further information about edit masks, see the session parameter EM in the *Parameter Reference*.

Example Program without EM Parameters

```
** 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
```

Output of Program EDITMX01:

The output of this program shows the default edit masks available.

```
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
```

Example Program with EM Parameters

```
** 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^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
```

Output of Program `EDITMX02`:

Compare the output with that of the previous program (*Example Program without EM Parameters*) to see how the `EM` specifications affect the way the fields are displayed.

```
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
```

Further Examples of Edit Masks

See the following example programs:

- ***EDITMX03 - Edit mask (different EM for alpha-numeric fields)***
- ***EDITMX04 - Edit mask (different EM for numeric fields)***
- ***EDITMX05 - Edit mask (EM for date and time system variables)***

36 Unicode Edit Masks - EMU Parameter

Unicode edit masks can be used similar to code page edit masks. The difference is that the edit mask is always stored in Unicode format.

This allows you to specify edit masks with mixed characters from different code pages and assures that always the correct character is displayed, independent of the installed system code page.

For the general usage of edit masks, see ***Edit Masks - EM Parameter***.

For information on the session parameter `EMU`, see *EMU - Unicode Edit Mask* (in the *Parameter Reference*).

37 Vertical Displays

- Creating Vertical Displays 370
- Combining DISPLAY and WRITE 370
- Tab Notation - T*field 371
- Positioning Notation x/y 372
- DISPLAY VERT Statement 373
- Further Example of DISPLAY VERT with WRITE Statement 379

This chapter describes how you can combine the features of the statements DISPLAY and WRITE to produce vertical displays of field values.

Creating Vertical Displays

There are two ways of creating vertical displays:

- You can use a combination of the statements DISPLAY and WRITE.
- You can use the VERT option of the DISPLAY statement.

Combining DISPLAY and WRITE

As described in *Statements DISPLAY and WRITE*, the DISPLAY statement normally presents the data in columns with default headers, while the WRITE statement presents data horizontally without headers.

You can combine the features of the two statements to produce vertical displays of field values.

The DISPLAY statement produces the values of different fields for the same record across the page with a column for each field. The field values for each record are displayed below the values for the previous record.

By using a WRITE statement after a DISPLAY statement, you can insert text and/or field values specified in the WRITE statement between records displayed via the DISPLAY statement.

The following program illustrates the combination of DISPLAY and 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
```

Output of Program 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
```

Tab Notation - T*field

In the previous example, the position of the field DEPT is determined by the tab notation *n*T (in this case 20T, which means that the display begins in column 20 on the screen).

Field values specified in a WRITE statement can be lined up automatically with field values specified in the first DISPLAY statement of the program by using the tab notation T**field* (where *field* is the name of the field to which the field is to be aligned).

In the following program, the output produced by the WRITE statement is aligned to the field JOB-TITLE by using the notation T*JOB-TITLE:

```
** 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
```

Output of Program 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
```

Positioning Notation x/y

When you use the `DISPLAY` and `WRITE` statements in sequence and multiple lines are to be produced by the `WRITE` statement, you can use the notation *x*/*y* (number-slash-number) to determine in which row/column something is to be displayed. The positioning notation causes the next element in the `DISPLAY` or `WRITE` statement to be placed *x* lines below the last output, beginning in column *y* of the output.

The following program illustrates the use of this 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
```

Output of Program `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

The standard display mode in Natural is horizontal.

With the `VERT` clause option of the `DISPLAY` statement, you can override the standard display and produce a vertical field display.

The `HORIZ` clause option, which can be used in the same `DISPLAY` statement, re-activates the standard horizontal display mode.

Column headings in vertical mode are controlled with various forms of the `AS` clause. The following example programs illustrate the use of the `DISPLAY VERT` statement:

- DISPLAY VERT without AS Clause
- DISPLAY with VERT AS CAPTIONED and HORIZ Clause
- DISPLAY with VERT AS 'text' Clause
- DISPLAY with VERT AS 'text' CAPTIONED Clause

- Tab Notation P*field

DISPLAY VERT without AS Clause

The following program has no `AS` clause, which means that no column headings are output.

```
** 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
```

Output of Program `DISPLX09`:

Note that all field values are displayed vertically underneath one another.

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


RUBIN
SYLVIA

NEW YORK


WALLACE
MARY

NEW YORK


KELLOGG
HENRIETTA

NEWARK
```

DISPLAY with VERT AS CAPTIONED and HORIZ Clause

The following program contains a `VERT` and a `HORIZ` clause, which causes some column values to be output vertically and others horizontally; moreover `AS CAPTIONED` causes the default column headers to be displayed.

```
** 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
```

Output of Program `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
```

DISPLAY with VERT AS 'text' Clause

The following program contains an `AS 'text'` clause, which displays the specified `'text'` as column header.

Note: A slash (/) within the text element in a `DISPLAY` statement causes a line advance.

```
** 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
```

Output of Program `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
```

DISPLAY with VERT AS 'text' CAPTIONED Clause

The `AS 'text' CAPTIONED` clause causes the specified text to be displayed as column heading, and the default column headings to be displayed immediately before the field value in each line that is output.

The following program contains an `AS 'text' CAPTIONED` clause.

```
** 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
```

Output of Program `DISPLX12`:

This clause causes the default column headers (`NAME` and `FIRST-NAME`) to be placed before the field values:

```
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
```

Tab Notation P*field

If you use a combination of DISPLAY VERT statement and subsequent WRITE statement, you can use the tab notation P*field-name* in the WRITE statement to align the position of a field to the column *and* line position of a particular field specified in the DISPLAY VERT statement.

In the following program, the fields SALARY and BONUS are displayed in the same column, SALARY in every first line, BONUS in every second line. The text ***SALARY PLUS BONUS*** is aligned to SALARY, which means that it is displayed in the same column as SALARY and in the first line, whereas the text (IN US DOLLARS) is aligned to BONUS and therefore displayed in the same column as BONUS and in the second line.

```
** 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
```

Output of Program 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)
```

Further Example of DISPLAY VERT with WRITE Statement

See the following example program:

- ***WRITEX10 - WRITE (with nT, T*field and P*field)***

VII Further Programming Aspects

End of Statement, Program or Application
Processing of Application Errors
Conditional Processing - IF Statement
Loop Processing
Control Breaks
Data Computation
System Variables and System Functions
Stack
Processing of Date Information
Text Notation
User Comments
Logical Condition Criteria
Rules for Arithmetic Assignment
Invoking Natural Subprograms from 3GL Programs
Issuing Operating System Commands from within a Natural Program
Statements for Internet and XML Access

38 End of Statement, Program or Application

- End of Statement 384
- End of Program 384
- End of Application 384

End of Statement

To explicitly mark the end of a statement, you can place a semicolon (;) between the statement and the next statement. This can be used to make the program structure clearer, but is not required.

End of Program

The `END` statement is used to mark the end of a Natural program, function, subprogram, external subroutine or helproutine.

Every one of these objects must contain an `END` statement as the last statement.

Every object may contain only one `END` statement.

End of Application

Ending the Execution of an Application by a STOP Statement

The `STOP` statement is used to terminate the execution of a Natural application. A `STOP` statement executed anywhere within an application immediately stops the execution of the entire application.

Ending the Execution of an Application by a TERMINATE Statement

The `TERMINATE` statement stops the execution of the Natural application and also ends the Natural session.

Interrupting a Running Natural Application

During the development of a Natural application and in test situations, the user should be able to interrupt a running Natural application that does not respond anymore, for example, due to an endless loop. As the Natural session should not need to be killed, the running Natural application can be interrupted via the typical system interrupt key combination (for example, CTRL+BREAK for Windows, CTRL+C for UNIX and OpenVMS). The Natural error NAT1016 is raised and the runtime error processing is activated. The error can be handled by an `ON ERROR` processing.

In a production environment, this feature will typically need to be disabled, because the application may not be able to recover from a user interrupt at an arbitrary program location.

The Natural profile parameter `RTINT` determines whether interrupts are allowed. By default, interrupts are not allowed.

If this parameter is set to `ON`, a running Natural application may be interrupted with the interrupt key combination of the operating system (for example, for Windows: CTRL+BREAK; for UNIX: typically CTRL+C, but can be reconfigured using the `stty` command; for OpenVMS: CTRL+C).

Natural catches this interrupt request and then offers the user the following possibilities:

- Perform standard error processing by raising a NAT1016 error.
- Continue application processing (cancel interrupt).

The choice is shown in a window that is opened after catching the interrupt signal.

Note: The Natural application will only be interruptible if the Natural application or Natural Studio that started the application has the input focus.

39 Processing of Application Errors

- Natural's Default Error Processing 388
- Application Specific Error Processing 388
- Using an ON ERROR Statement Block 389
- Using an Error Transaction Program 390
- Error Processing Related Features 393

This section discusses the two basic methods Natural offers for the handling of application errors: default processing and application-specific processing. Furthermore, it describes the options you have to enable the application specific error processing: coding an `ON ERROR` statement block within a programming object or using a separate error transaction program.

Finally, this section gives an overview of the features that are provided to configure Natural's error processing behavior, to retrieve information on an error, to process or debug an application error.

This document covers the following topics:

For information on error handling in a Natural RPC environment, see *Handling Errors* in the *Natural Remote Procedure Call* documentation.

Natural's Default Error Processing

When an error occurs in a Natural application, Natural will by default proceed in the following way:

1. Natural terminates the execution of the currently running application object;
2. Natural issues an error message;
3. Natural returns to command input mode.

"Command input mode" means that, depending on your Natural configuration, the Natural main menu, the `NEXT` command prompt, or a user-defined startup menu may appear.

The displayed error message contains the Natural error number, the corresponding message text and the affected Natural object and line number where the error has occurred.

Because after the occurrence of an error the execution of the affected application object is terminated, the status of any pending database transactions may be affected by actions required by the setting of the profile parameter `ETEOP`. Unless Natural has issued an `END TRANSACTION` statement as a result of the settings of these parameters, a `BACKOUT TRANSACTION` statement is issued when Natural returns to command input mode.

Application Specific Error Processing

Natural enables you to adapt the error processing if the default error processing does not meet your application's requirements. Possible reasons to establish an application specific error processing may be:

- The information on the error is to be stored for further analysis by the application developer.
- The application execution is to be continued after error recovery, if possible.

- A specific transaction handling is necessary.

Because the execution of the affected Natural application object is terminated after an application error has occurred, the status of the pending database transactions may be influenced by actions which are triggered by the settings of the profile parameter ETEOP. Therefore, further transaction handling (END TRANSACTION or BACKOUT TRANSACTION statement) has to be performed by the application's error processing.

To enable the application specific error processing, you have the following options:

- You may code an ON ERROR statement block within a programming object.
- You may use a separate error transaction program.

These options are described in the following sections.

Using an ON ERROR Statement Block

You may use the ON ERROR statement to intercept execution time errors within the application where an error occurs.

From within an ON ERROR statement block, it is possible to resume application execution on the current level or on a superior level.

Moreover, you may specify an ON ERROR statement in multiple objects of an application in order to process any errors that have occurred on subordinate levels. Thus, application specific error processing may exactly be tailored to the application's needs.

Exiting from an ON ERROR Statement Block

You may exit from an ON ERROR statement block by specifying one of the following statements:

- RETRY

  Application execution is resumed on the current level.
- ESCAPE ROUTINE

  Error processing is assumed to be complete and application execution is resumed on the superior level.
- FETCH

  Error processing is assumed to be complete and the "fetched" program is executed.

  STOP

Natural stops the execution of the affected program, ends the application and returns to command input mode.

- TERMINATE

  The execution of the Natural application is stopped and also the Natural session is terminated.

Error Processing Rules

- If the execution of the ON ERROR statement block is not terminated by one of these statements, the error is percolated to the Natural object on the superior level for processing by an ON ERROR statement block that exists there.
- If none of the Natural objects on any of the superior levels contains an ON ERROR statement block, but if an error transaction program has been specified (as described in the **next section**), this error transaction program will receive control.
- If none of the Natural objects on any of the superior levels contains an ON ERROR statement block and no error transaction program has been specified there, Natural's default error processing will be performed as described **above**.

Using an Error Transaction Program

You may specify an error transaction program in the following places:

- In the profile parameter ETA.
- If Natural Security is installed, within the Natural Security library profile; see *Components of a Library Profile* in the *Natural Security* documentation.
- Within a Natural object by assigning the name of the program which is to receive control in the event of an error condition as a value to the system variable *ERROR-TA, using an ASSIGN, COMPUTE or MOVE statement.

If you assign the name of an error transaction program to the system variable *ERROR-TA during the Natural session, this assignment supersedes an error transaction program specified using the profile parameter ETA. Regardless of whether you use the ETA profile parameter or assign a value to the system variable *ERROR-TA, the error transaction program names are not saved and restored by Natural for different levels of the call hierarchy. Therefore, if you assign a name to the system variable *ERROR-TA in a Natural object, the specified program will be invoked to process any error that occurs in the current Natural session after the assignment.

On the one hand, if you specify an error transaction program by using the profile parameter ETA, an error transaction is defined for the complete Natural session without having the need for individual assignments in Natural objects. On the other hand, the method of assigning a program to the system variable *ERROR-TA provides more flexibility and, for example, allows you to have different error transaction programs in different application branches.

If the system variable *ERROR-TA is reset to blank, Natural's default error processing will be performed as described **above**.

If an error transaction program is specified and an application error occurs, execution of the application is terminated, and the specified error transaction program receives control to perform the following actions:

- Analyze the error;
- Log the error information;
- Terminate the Natural session;
- Continue the application execution by calling a program using the FETCH statement.

Because the error transaction program receives control in the same way as if it had been called from the command prompt, it is not possible to resume application execution in one of the Natural objects that were active at the time when the error occurred.

If a syntax error occurs and the Natural profile parameter SYNERR is set to ON, the error transaction program will also receive control.

An error transaction program must be located in the library to which you are currently logged on or in a current steplib library.

When an error occurs, Natural executes a STACK TOP DATA statement and places the following information at the top of the stack:

Stack Data	Format/Length	Description	
Error number	N4	Natural error number. **Note:** If session parameter SG is set to ON, the format/length will be N5.	
Line number	N4	Number of the line where the error has occurred. If the status is C or L, the line number will be zero.	
Status	A1	Status code:	
		C	Command processing error
		L	Logon processing error
		O	Object (execution) time error
		R	Error on remote server (in conjunction with Natural RPC)
		S	Syntax error
Object name	A8	Name of the Natural object where the error has occurred.	
Level number	N2	Level number of the Natural object where the error has occurred. If a Natural syntax error occurs at compile time and profile parameter SYNERR is set to ON, the level number will be zero.	

Stack Data	Format/Length	Description
		If a Natural runtime error occurs and the level number of the Natural object is greater than 99, the value 99 will be stacked, and the current value will be stacked in the additional stack data "Level number enhanced".
If a Natural runtime error occurs and the level number of the Natural object is greater than 99:		
Level number enhanced	I4	Current level number (512 at maximum).
If a Natural syntax error occurs at compile time and profile parameter SYNERR is set to ON:		
Error position	N3	Position of the offending item in the source line.
Item length	N3	Length of the offending item.

This information can be retrieved in the error transaction program, using an INPUT statement.

Example:

```
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-ITEM     (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-ITEM
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
```

Some of the information placed on top of the stack is equivalent to the contents of several system variables that are available in an `ON ERROR` statement block:

Stack Data	Equivalent System Variable in ON ERROR Statement Block
Error number	`*ERROR-NR`
Line number	`*ERROR-LINE`
Object name	`*PROGRAM`
Level number	`*LEVEL`

Rules under Natural Security

If Natural Security is installed, the additional rules for the processing of logon errors apply. For further information, see *Transactions* in the *Natural Security* documentation.

Error Processing Related Features

Natural provides a variety of error processing related features that

- Enable you to configure Natural's error processing behavior;
- Help you in retrieving information about errors that have occurred;
- Support you in processing these errors;
- Support you in debugging application errors.

These features can be grouped as follows:

- **Profile parameters**
- **System variables**
- **Terminal Commands**
- **System commands**
- **Application programming interfaces**

Profile Parameters

The following profile parameters have an influence on the behavior of Natural in the event of an error:

Profile Parameter	Purpose
CPCVERR	Conversion Error
ETA	Error Transaction Program
ETEOP	Issue END TRANSACTION at End of Program
RCFIND	Handling of Response Code 113 for FIND Statement
RCGET	Handling of Response Code 113 for GET Statement
SYNERR	Control of Syntax Errors

System Variables

The following application related system variables can be used to locate an error or to obtain/specify the name of the program which is to receive control in the event of an error condition:

System Variable	Content
*ERROR-LINE	Source-code line number of the statement that caused an error. See **Example 1**.
*ERROR-NR	Error number of the error which caused an ON ERROR condition to be entered.
*ERROR-TA	Name of the program which is to receive control in the event of an error condition. See **Example 2**.
*LEVEL	Level number of the Natural object where the error has occurred.
*LIBRARY-ID	Name of the library to which the user is currently logged on.
*PROGRAM	Name of the Natural object that is currently being executed. See **Example 1**.

Example 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
    /*
...
```

Example 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
...
```

For further information on these system variables, see the corresponding sections in the *System Variables* documentation.

Terminal Commands

The following terminal command has an influence on the behavior of Natural in the event of an error:

Terminal Command	Purpose
%E=	Activate/Deactivate Error Processing

System Commands

The following system commands provide additional information on an error situation or invoke the utilities for debugging or logging database calls:

System Command	Purpose
LASTMSG	Display additional information on the error situation which has occurred last.
TECH	Display technical and other information about your Natural session, for example, information on the last error that occurred.

Application Programming Interfaces

The following application programming interfaces (APIs) are generally available for getting additional information on an error situation or to install an error transaction.

API	Purpose
USR0040N	Get type of last error
USR1016N	Get error level for error in nested copycodes
USR2001N	Get information on last error
USR2006N	Get information from error message collector
USR2007N	Get or set data for RPC default server
USR2010N	Get error information on last database call
USR2026N	Get TECH information
USR2030N	Get dynamic error message parts from the last error

API	Purpose
USR3320N	Find user short error message (including steplibs search)
USR4214N	Get program level information

For further information, see *SYSEXT - Natural Application Programming Interfaces* in the *Utilities* documentation.

40 Conditional Processing - IF Statement

- Structure of IF Statement 398
- Nested IF Statements 400

With the IF statement, you define a logical condition, and the execution of the statement attached to the IF statement then depends on that condition.

Structure of IF Statement

The IF statement contains three components:

IF	In the IF clause, you specify the logical condition which is to be met.
THEN	In the THEN clause you specify the statement(s) to be executed if this condition is met.
ELSE	In the (optional) ELSE clause, you can specify the statement(s) to be executed if this condition is *not* met.

So, an IF statement takes the following general form:

```
IF condition
   THEN execute statement(s)
   ELSE execute other statement(s)
END-IF
```

Note: If you wish a certain processing to be performed only if the IF condition is *not* met, you can specify the clause THEN IGNORE. The IGNORE statement causes the IF condition to be ignored if it is met.

Example 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
```

The IF statement block in the above program causes the following conditional processing to be performed:

- IF the salary is less than 40000, THEN the WRITE statement is to be executed;
- otherwise (ELSE), that is, if the salary is 40000 or more, the DISPLAY statement is to be executed.

Output of Program 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
```

Example 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
```

Output of Program `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
                ====================
```

Nested IF Statements

It is possible to use various nested `IF` statements; for example, you can make the execution of a `THEN` clause dependent on another `IF` statement which you specify in the `THEN` clause.

Example:

```
** 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
```

Output of Program `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
```

41 Loop Processing

- Use of Processing Loops 404
- Limiting Database Loops 404
- Limiting Non-Database Loops - REPEAT Statement 406
- Example of REPEAT Statement 407
- Terminating a Processing Loop - ESCAPE Statement 408
- Loops Within Loops 408
- Example of Nested FIND Statements 408
- Referencing Statements within a Program 409
- Example of Referencing with Line Numbers 411
- Example with Statement Reference Labels 412

A processing loop is a group of statements which are executed repeatedly until a stated condition has been satisfied, or as long as a certain condition prevails.

Use of Processing Loops

Processing loops can be subdivided into database loops and non-database loops:

- **Database processing loops**
  are those created automatically by Natural to process data selected from a database as a result of a `READ`, `FIND` or `HISTOGRAM` statement. These statements are described in the section ***Database Access***.
- **Non-database processing loops**
  are initiated by the statements `REPEAT`, `FOR`, `CALL FILE`, `CALL LOOP`, `SORT`, and `READ WORK FILE`.

More than one processing loop may be active at the same time. Loops may be embedded or nested within other loops which remain active (open).

A processing loop must be explicitly closed with a corresponding `END-...` statement (for example, `END-REPEAT`, `END-FOR`)

The `SORT` statement, which invokes the sort program of the operating system, closes all active processing loops and initiates a new processing loop.

Limiting Database Loops

The following topics are covered below:

- Possible Ways of Limiting Database Loops
- LT Session Parameter
- LIMIT Statement
- Limit Notation

- Priority of Limit Settings

Possible Ways of Limiting Database Loops

With the statements READ, FIND or HISTOGRAM, you have three ways of limiting the number of repetitions of the processing loops initiated with these statements:

- using the session parameter LT,
- using a LIMIT statement,
- or using a **limit notation** in a READ/FIND/HISTOGRAM statement itself.

LT Session Parameter

With the system command GLOBALS, you can specify the session parameter LT, which limits the number of records which may be read in a database processing loop.

Example:

```
GLOBALS LT=100
```

This limit applies to all READ, FIND and HISTOGRAM statements in the entire session.

LIMIT Statement

In a program, you can use the LIMIT statement to limit the number of records which may be read in a database processing loop.

Example:

```
LIMIT 100
```

The LIMIT statement applies to the remainder of the program unless it is overridden by another LIMIT statement or limit notation.

Limit Notation

With a READ, FIND or HISTOGRAM statement itself, you can specify the number of records to be read in parentheses immediately after the statement name.

Example:

```
READ (10) VIEWXYZ BY NAME
```

This limit notation overrides any other limit in effect, but applies only to the statement in which it is specified.

Priority of Limit Settings

If the limit set with the LT parameter is smaller than a limit specified with a LIMIT statement or a limit notation, the LT limit has priority over any of these other limits.

Limiting Non-Database Loops - REPEAT Statement

Non-database processing loops begin and end based on logical condition criteria or some other specified limiting condition.

The REPEAT statement is discussed here as representative of a non-database loop statement.

With the REPEAT statement, you specify one or more statements which are to be executed repeatedly. Moreover, you can specify a logical condition, so that the statements are only executed either until or as long as that condition is met. For this purpose you use an UNTIL or WHILE clause.

If you specify the logical condition

- in an UNTIL clause, the REPEAT loop will continue *until* the logical condition is met;
- in a WHILE clause, the REPEAT loop will continue *as long as* the logical condition remains true.

If you specify *no* logical condition, the REPEAT loop must be exited with one of the following statements:

- ESCAPE terminates the execution of the processing loop and continues processing outside the loop (see **below**).
- STOP stops the execution of the entire Natural application.
- TERMINATE stops the execution of the Natural application and also ends the Natural session.

Example of 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
```

Output of Program `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
```

Terminating a Processing Loop - ESCAPE Statement

The ESCAPE statement is used to terminate the execution of a processing loop based on a logical condition.

You can place an ESCAPE statement within loops in conditional IF statement groups, in break processing statement groups (AT END OF DATA, AT END OF PAGE, AT BREAK), or as a stand-alone statement implementing the basic logical conditions of a non-database loop.

The ESCAPE statement offers the options TOP and BOTTOM, which determine where processing is to continue after the processing loop has been left via the ESCAPE statement:

- ESCAPE TOP is used to continue processing at the top of the processing loop.
- ESCAPE BOTTOM is used to continue processing with the first statement following the processing loop.

You can specify several ESCAPE statements within the same processing loop.

For further details and examples of the ESCAPE statement, see the *Statements* documentation.

Loops Within Loops

A database statement can be placed within a database processing loop initiated by another database statement. When database loop-initiating statements are embedded in this way, a "hierarchy" of loops is created, each of which is processed for each record which meets the selection criteria.

Multiple levels of loops can be embedded. For example, non-database loops can be nested one inside the other. Database loops can be nested inside non-database loops. Database and non-database loops can be nested within conditional statement groups.

Example of Nested FIND Statements

The following program illustrates a hierarchy of two loops, with one FIND loop nested or embedded within another FIND loop.

```
** 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
```

The above program selects data from multiple files. The outer `FIND` loop selects from the `EMPLOYEES` file all persons who live in New York or Beverley Hills. For each record selected in the outer loop, the inner `FIND` loop is entered, selecting the car data of those persons from the `VEHICLES` file.

Output of Program `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
```

Referencing Statements within a Program

Statement reference notation is used for the following purposes:

- Referring to previous statements in a program in order to specify processing over a particular range of data.
- Overriding Natural's **default referencing**.
- Documenting.

Any Natural statement which causes a processing loop to be initiated and/or causes data elements in a database to be accessed can be referenced, for example:

- READ
- FIND
- HISTOGRAM
- SORT
- REPEAT
- FOR

When multiple processing loops are used in a program, reference notation is used to uniquely identify the particular database field to be processed by referring back to the statement that originally accessed that field in the database.

If a field can be referenced in such a way, this is indicated in the Referencing Permitted column of the *Operand Definition Table* in the corresponding statement description (in the *Statements* documentation). See also ***User-Defined Variables***, ***Referencing of Database Fields Using (r) Notation***.

In addition, reference notation can be specified in some statements. For example:

- AT START OF DATA
- AT END OF DATA
- AT BREAK
- ESCAPE BOTTOM

Without reference notation, an AT START OF DATA, AT END OF DATA or AT BREAK statement will be related to the *outermost* active READ, FIND, HISTOGRAM, SORT or READ WORK FILE loop. With reference notation, you can relate it to another active processing loop.

If reference notation is specified with an ESCAPE BOTTOM statement, processing will continue with the first statement following the processing loop identified by the reference notation.

Statement reference notation may be specified in the form of a *statement reference label* or a *source-code line number*.

- **Statement reference label**
  A statement reference label consists of several characters, the last of which must be a period (.). The period serves to identify the entry as a label.

  A statement that is to be referenced is marked with a label by placing the label at the beginning of the line that contains the statement. For example:

```
0030 ...
0040 READ1. READ VIEWXYZ BY NAME
0050 ...
```

In the statement that references the marked statement, the label is placed in parentheses at the location indicated in the statement's syntax diagram (as described in the *Statements* documentation). For example:

```
AT BREAK (READ1.) OF NAME
```

- **Source-code line number**
  If source-code line numbers are used for referencing, they must be specified as 4-digit numbers (leading zeros must not be omitted) and in parentheses. For example:

```
AT BREAK (0040) OF NAME
```

  In a statement where the label/line number relates a particular field to a previous statement, the label/line number is placed in parentheses after the field name. For example:

```
DISPLAY NAME (READ1.) JOB-TITLE (READ1.) MAKE MODEL
```

Line numbers and labels can be used interchangeably.

See also ***User-Defined Variables***, ***Referencing of Database Fields Using (r) Notation***.

Example of Referencing with Line Numbers

The following program uses source code line numbers (4-digit numbers in parentheses) for referencing.

In this particular example, the line numbers refer to the statements that would be referenced in any case by default.

```
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
```

Example with Statement Reference Labels

The following example illustrates the use of statement reference labels.

It is identical to the previous program, except that labels are used for referencing instead of line numbers.

```
** 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 ↩
```

Both programs produce the following output:

```
        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***
```

42 Control Breaks

- Use of Control Breaks 416
- AT BREAK Statement 416
- Automatic Break Processing 421
- Example of System Functions with AT BREAK Statement 422
- Further Example of AT BREAK Statement 424
- BEFORE BREAK PROCESSING Statement 424
- Example of BEFORE BREAK PROCESSING Statement 424
- User-Initiated Break Processing - PERFORM BREAK PROCESSING Statement 425
- Example of PERFORM BREAK PROCESSING Statement 427

This chapter describes how the execution of a statement can be made dependent on a control break, and how control breaks can be used for the evaluation of Natural system functions.

Use of Control Breaks

A control break occurs when the value of a control field changes.

The execution of statements can be made dependent on a control break.

A control break can also be used for the evaluation of Natural system functions.

System functions are discussed in ***System Variables and System Functions***. For detailed descriptions of the system functions available, refer to the *System Functions* documentation.

AT BREAK Statement

With the statement `AT BREAK`, you specify the processing which is to be performed whenever a control break occurs, that is, whenever the value of a control field which you specify with the `AT BREAK` statement changes. As a control field, you can use a database field or a user-defined variable.

The following topics are covered below:

- Control Break Based on a Database Field
- Control Break Based on a User-Defined Variable
- Multiple Control Break Levels

Control Break Based on a Database Field

The field specified as control field in an `AT BREAK` statement is usually a database field.

Example:

```
...
AT BREAK OF DEPT
  statements
END-BREAK
...
```

In this example, the control field is the database field `DEPT`; if the value of the field changes, for example, `FROM SALE01` to `SALE02`, the *`statements`* specified in the `AT BREAK` statement would be executed.

Instead of an entire field, you can also use only part of a field as a control field. With the slash-*n*-slash notation */n/*, you can determine that only the first *n* positions of a field are to be checked for a change in value.

Example:

```
...
AT BREAK OF DEPT /4/
  statements
END-BREAK
...
```

In this example, the specified *statements* would only be executed if the value of the first 4 positions of the field DEPT changes, for example, FROM SALE to TECH; if, however, the field value changes from SALE01 to SALE02, this would be ignored and no AT BREAK processing performed.

Example:

```
** 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^)
          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
```

In the above program, the first WRITE statement is executed whenever the value of the field CITY changes.

In the AT BREAK statement, the Natural system functions OLD, AVER and COUNT are evaluated (and output in the WRITE statement).

In the AT END OF DATA statement, the Natural system function TOTAL is evaluated.

Output of Program ATBREX01:

```
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           ↩
```

Control Break Based on a User-Defined Variable

A **user-defined variable** can also be used as control field in an AT BREAK statement.

In the following program, the user-defined variable #LOCATION is used as control field.

```
** 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
```

Output of Program 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                    ↩
```

Multiple Control Break Levels

As explained **above**, the notation */n/* allows some portion of a field to be checked for a control break. It is possible to combine several AT BREAK statements, using an entire field as control field for one break and part of the same field as control field for another break.

In such a case, the break at the lower level (entire field) must be specified before the break at the higher level (part of field); that is, in the first AT BREAK statement the entire field must be specified as control field, and in the second one part of the field.

The following example program illustrates this, using the field DEPT as well as the first 4 positions of that field (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
```

Output of Program 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 ***    ↩
```

In the following program, one blank line is output whenever the value of the field DEPT changes; and whenever the value in the first 4 positions of DEPT changes, a record count is carried out by evaluating the system function COUNT.

```
** 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
```

Output of Program 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                                          ↩
```

Automatic Break Processing

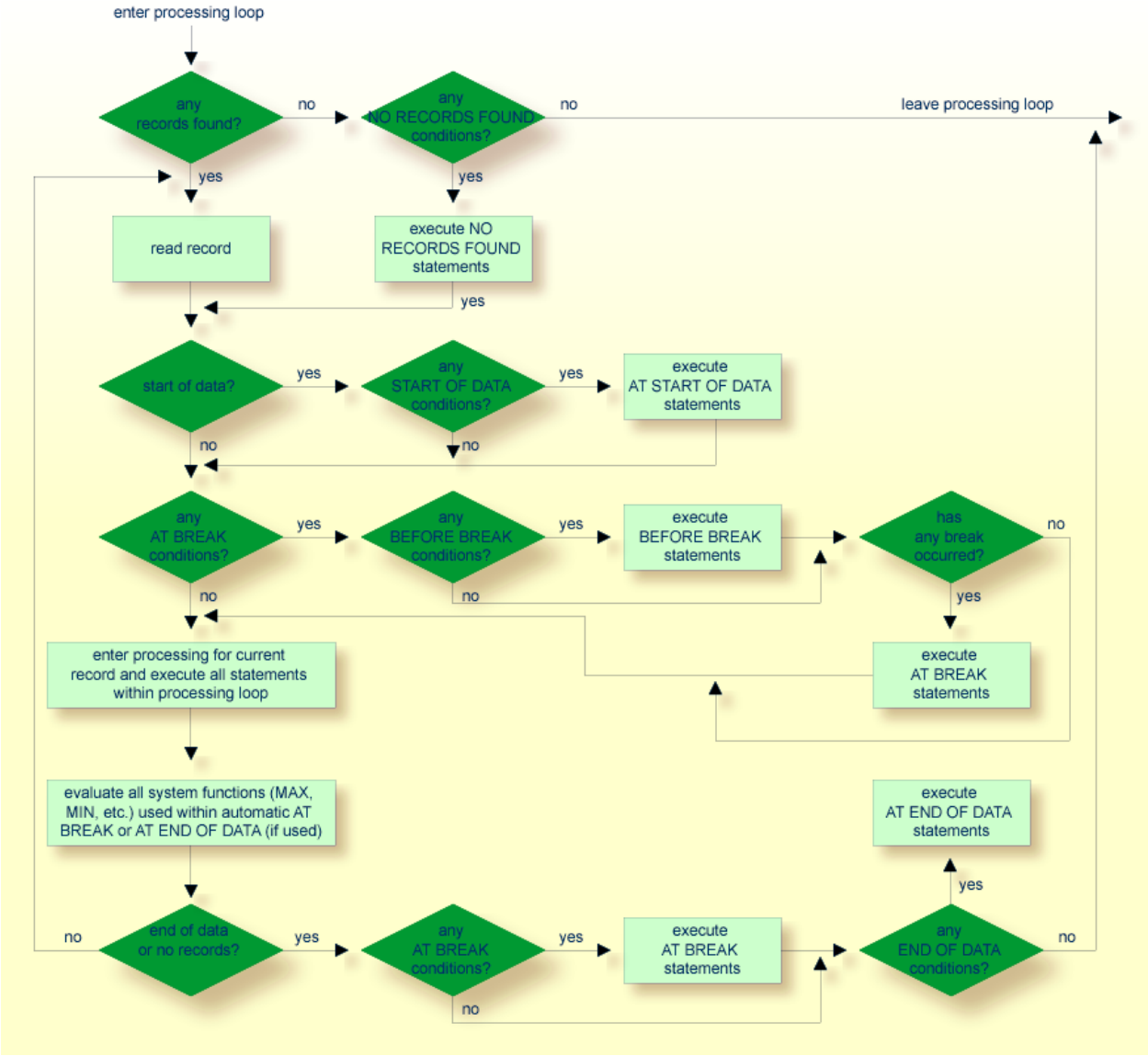
Automatic break processing is in effect for a processing loop which contains an AT BREAK statement. This applies to the following statements:

- FIND
- READ
- HISTOGRAM
- SORT
- READ WORK FILE

The value of the control field specified with the AT BREAK statement is checked only for records which satisfy the selection criteria of both the WITH clause and the WHERE clause.

Natural system functions (AVER, MAX, MIN, etc.) are evaluated for each record after all statements within the processing loop have been executed. System functions are not evaluated for any record which is rejected by WHERE criteria.

The figure below illustrates the flow logic of automatic break processing.



Example of System Functions with AT BREAK Statement

The following example shows the use of the Natural system functions `OLD`, `MIN`, `AVER`, `MAX`, `SUM` and `COUNT` in an `AT BREAK` statement (and of the system function `TOTAL` in an `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^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
```

Output of Program `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    ↩
```

Further Example of AT BREAK Statement

See the following example program:

- *ATBREX06 - AT BREAK OF (comparing NMIN, NAVER, NCOUNT with MIN, AVER, COUNT)*

BEFORE BREAK PROCESSING Statement

With the `BEFORE BREAK PROCESSING` statement, you can specify statements that are to be executed immediately before a control break; that is, before the value of the control field is checked, before the statements specified in the `AT BREAK` block are executed, and before any Natural system functions are evaluated.

Example of 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
```

Output of Program 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
                                 ------------------------
```

User-Initiated Break Processing - PERFORM BREAK PROCESSING Statement

With automatic break processing, the statements specified in an AT BREAK block are executed whenever the value of the specified control field changes - regardless of the position of the AT BREAK statement in the processing loop.

With a PERFORM BREAK PROCESSING statement, you can perform break processing at a specified position in a processing loop: the PERFORM BREAK PROCESSING statement is executed when it is encountered in the processing flow of the program.

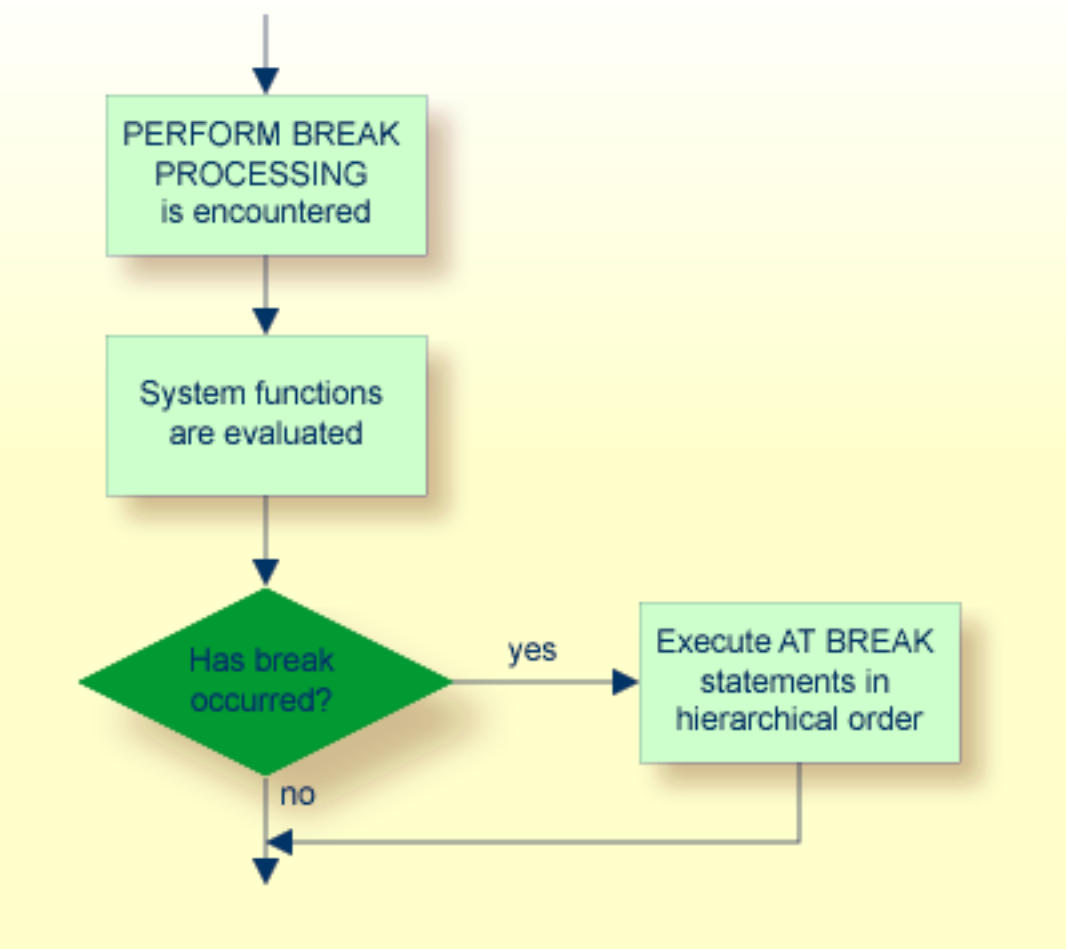
Immediately after the PERFORM BREAK PROCESSING, you specify one or more AT BREAK statement blocks:

```
...
PERFORM BREAK PROCESSING
  AT BREAK OF field1
    statements
  END-BREAK
  AT BREAK OF field2
    statements
  END-BREAK
...
```

When a PERFORM BREAK PROCESSING is executed, Natural checks if a break has occurred; that is, if the value of the specified control field has changed; and if it has, the specified statements are executed.

With PERFORM BREAK PROCESSING, system functions are evaluated *before* Natural checks if a break has occurred.

The following figure illustrates the flow logic of user-initiated break processing:



Example of 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
```

Output of Program 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          ↩
```

43 Data Computation

- COMPUTE Statement 430
- Statements MOVE and COMPUTE 431
- Statements ADD, SUBTRACT, MULTIPLY and DIVIDE 432
- Example of MOVE, SUBTRACT and COMPUTE Statements 432
- COMPRESS Statement 433
- Example of COMPRESS and MOVE Statements 434
- Example of COMPRESS Statement 435
- Mathematical Functions 436
- Further Examples of COMPUTE, MOVE and COMPRESS Statements 437

This chapter discusses arithmetic statements that are used for computing data:

- COMPUTE
- ADD
- SUBTRACT
- MULTIPLY
- DIVIDE

In addition, the following statements are discussed which are used to transfer the value of an operand into one or more fields:

- MOVE
- COMPRESS

Important: For optimum processing, **user-defined variables** used in arithmetic statements should be defined with format P (packed numeric).

COMPUTE Statement

The COMPUTE statement is used to perform arithmetic operations. The following connecting operators are available:

**	Exponentiation
*	Multiplication
/	Division
+	Addition
-	Subtraction
( )	Parentheses may be used to indicate logical grouping.

Example 1:

```
COMPUTE LEAVE-DUE = LEAVE-DUE * 1.1
```

In this example, the value of the field LEAVE-DUE is multiplied by 1.1, and the result is placed in the field LEAVE-DUE.

Example 2:

```
COMPUTE #A = SQRT (#B)
```

In this example, the square root of the value of the field #B is evaluated, and the result is assigned to the field #A.

SQRT is a mathematical function supported in the following arithmetic statements:

- COMPUTE
- ADD
- SUBTRACT
- MULTIPLY
- DIVIDE

For an overview of mathematical functions, see **Mathematical Functions** below.

Example 3:

```
COMPUTE #INCOME = BONUS (1,1) + SALARY (1)
```

In this example, the first bonus of the current year and the current salary amount are added and assigned to the field #INCOME.

Statements MOVE and COMPUTE

The statements MOVE and COMPUTE can be used to transfer the value of an operand into one or more fields. The operand may be a constant such as a text item or a number, a database field, a user-defined variable, a system variable, or, in certain cases, a system function.

The difference between the two statements is that in the MOVE statement the value to be moved is specified on the left; in the COMPUTE statement the value to be assigned is specified on the right, as shown in the following examples.

Examples:

```
MOVE NAME TO #LAST-NAME
COMPUTE #LAST-NAME = NAME
```

Statements ADD, SUBTRACT, MULTIPLY and DIVIDE

The `ADD`, `SUBTRACT`, `MULTIPLY` and `DIVIDE` statements are used to perform arithmetic operations.

Examples:

```
ADD +5 -2 -1 GIVING #A
SUBTRACT 6 FROM 11 GIVING #B
MULTIPLY 3 BY 4 GIVING #C
DIVIDE 3 INTO #D GIVING #E
```

All four statements have a `ROUNDED` option, which you can use if you wish the result of the operation to be rounded.

For rules on rounding, see ***Rules for Arithmetic Assignment***.

The *Statements* documentation provides more detailed information on these statements.

Example of MOVE, SUBTRACT and COMPUTE Statements

The following program demonstrates the use of **user-defined variables** in arithmetic statements. It calculates the ages and wages of three employees and outputs these.

```
** 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
```

Output of Program 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

The COMPRESS statement is used to transfer (combine) the contents of two or more operands into a single alphanumeric field.

Leading zeros in a numeric field and trailing blanks in an alphanumeric field are suppressed before the field value is moved to the receiving field.

By default, the transferred values are separated from one another by a single blank in the receiving field. For other separating possibilities, see the COMPRESS statement option LEAVING NO SPACE (in the *Statements* documentation).

Example:

```
COMPRESS 'NAME:' FIRST-NAME #LAST-NAME INTO #FULLNAME
```

In this example, a COMPRESS statement is used to combine a text constant ('NAME:'), a database field (FIRST-NAME) and a user-defined variable (#LAST-NAME) into one user-defined variable (#FULLNAME).

For further information on the COMPRESS statement, please refer to the COMPRESS statement description (in the *Statements* documentation).

Example of COMPRESS and MOVE Statements

The following program illustrates the use of the statements MOVE and 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
```

Output of Program COMPRX01:

Notice the output format of the compressed field.

```
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
```

In multiple-line displays, it may be useful to combine fields/text in a **user-defined variables** by using a COMPRESS statement.

Example of COMPRESS Statement

In the following program, three **user-defined variables** are used: #FULL-SALARY, #FULL-NAME, and #FULL-CITY. #FULL-SALARY, for example, contains the text 'SALARY:' and the database fields SALARY and CURR-CODE. The WRITE statement then references only the compressed variables.

```
** 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^X)
  WRITE 1/5  #FULL-NAME
        1/37 #FULL-SALARY
        2/5  ADDRESS-LINE (1)
        3/5  #FULL-CITY
  SKIP 1
END-READ
END
```

Output of Program 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
```

Mathematical Functions

The following Natural mathematical functions are supported in arithmetic processing statements (ADD, COMPUTE, DIVIDE, SUBTRACT, MULTIPLY).

Mathematical Function	Natural System Function
Absolute value of *field*.	ABS(*field*)
Arc tangent of *field*.	ATN(*field*)
Cosine of *field*.	COS(*field*)
Exponential of *field*.	EXP(*field*)
Fractional part of *field*.	FRAC(*field*)
Integer part of *field*.	INT(*field*)
Natural logarithm of *field*.	LOG(*field*)
Sign of *field*.	SGN(*field*)
Sine of *field*.	SIN(*field*)
Square root of *field*.	SQRT(*field*)
Tangent of *field*.	TAN(*field*)
Numeric value of an alphanumeric *field*.	VAL(*field*)

See also the *System Functions* documentation for a detailed explanation of each mathematical function.

Further Examples of COMPUTE, MOVE and COMPRESS Statements

See the following example programs:

- *WRITEX11 - WRITE (with nX, n/n and COMPRESS)*
- *IFX03 - IF statement*
- *COMPRX03 - COMPRESS (using parameters LC and TC)*

44 System Variables and System Functions

- System Variables 440
- System Functions 441
- Example of System Variables and System Functions 442
- Further Examples of System Variables 443
- Further Examples of System Functions 444

This chapter describes the purpose of Natural system variables and Natural system functions and how they are used in Natural programs.

System Variables

The following topics are covered below:

- Purpose
- Characteristics of System Variables
- System Variables Grouped by Function

Purpose

System variables are used to display system information. They may be referenced at any point within a Natural program.

Natural system variables provide variable information, for example, about the current Natural session:

- the current library;
- the user and terminal identification;
- the current status of a loop processing;
- the current report processing status;
- the current date and time.

The typical use of system variables is illustrated in the ***Example of System Variables and System Functions*** below and in the examples contained in library `SYSEXPG`.

The information contained in a system variable may be used in Natural programs by specifying the appropriate system variables. For example, date and time system variables may be specified in a `DISPLAY`, `WRITE`, `PRINT`, `MOVE` or `COMPUTE` statement.

Characteristics of System Variables

The names of all system variables begin with an asterisk (*).

Format/Length

Information on format and length is given in the detailed descriptions in the *System Variables* documentation. The following abbreviations are used:

Format	
A	Alphanumeric
B	Binary
D	Date
I	Integer
L	Logical
N	Numeric (unpacked)
P	Packed numeric
T	Time

Content Modifiable

In the individual descriptions, this indicates whether in a Natural program you can assign another value to the system variable, that is, overwrite its content as generated by Natural.

System Variables Grouped by Function

The Natural system variables are grouped as follows:

- Application Related System Variables
- Date and Time System Variables
- Input/Ouput Related System Variables
- Natural Environment Related System Variables
- System Environment Related System Variables
- XML Related System Variables

For detailed descriptions of all system variables, see the *System Variables* documentation.

System Functions

Natural system functions comprise a set of statistical and mathematical functions that can be applied to the data after a record has been processed, but before break processing occurs.

System functions may be specified in a `DISPLAY`, `WRITE`, `PRINT`, `MOVE` or `COMPUTE` statement that is used in conjunction with an `AT END OF PAGE`, `AT END OF DATA` or `AT BREAK` statement.

In the case of an `AT END OF PAGE` statement, the corresponding `DISPLAY` statement must include the `GIVE SYSTEM FUNCTIONS` clause (as shown in the example **below**).

The following functional groups of system functions exist:

- System Functions for Use in Processing Loops
- Mathematical Functions
- Miscellaneous Functions

For detailed information on all system functions available, see the *System Functions* documentation.

See also *Using System Functions in Processing Loops* (in the *System Functions* documentation).

The typical use of system functions is explained in the example programs given below and in the examples contained in library SYSEXPG.

Example of System Variables and System Functions

The following example program illustrates the use of system variables and system functions:

```
** 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
```

Explanation:

- The system variable *DATE is output with the WRITE TITLE statement.
- The system variable *TIME is output with the AT START OF DATA statement.
- The system function OLD is used in the AT END OF DATA statement.
- The system function AVER is used in the AT END OF PAGE statement.

Output of Program SYSVAX01:

Note how the system variables and system function are displayed.

```
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
```

Further Examples of System Variables

See the following example programs:

- *EDITMX05 - Edit mask (EM for date and time system variables)*
- *READX04 - READ (in combination with FIND and the system variables *NUMBER and *COUNTER)*
- *WTITLX01 - WRITE TITLE (with *PAGE-NUMBER)*

Further Examples of System Functions

See the following example programs:

- *ATBREX06 - AT BREAK OF (comparing NMIN, NAVER, NCOUNT with MIN, AVER, COUNT)*
- *ATENPX01 - AT END OF PAGE (with system function available via GIVE SYSTEM FUNCTIONS in DISPLAY)*

45 Stack

- Use of Natural Stack 446
- Stack Processing 446
- Placing Data on the Stack 447
- Clearing the Stack 448

The Natural stack is a kind of “intermediate storage” in which you can store Natural commands, user-defined commands, and input data to be used by an INPUT statement.

Use of Natural Stack

In the stack you can store a series of functions which are frequently executed one after the other, such as a series of logon commands.

The data/commands stored in the stack are “stacked” on top of one another. You can decide whether to put them on top or at the bottom of the stack. The data/command in the stack can only be processed in the order in which they are stacked, beginning from the top of the stack.

In a program, you may reference the system variable *DATA to determine the content of the stack (see the *System Variables* documentation for further information).

Stack Processing

The processing of the commands/data stored in the stack differs depending on the function being performed.

If a command is expected, that is, the NEXT prompt is about to be displayed, Natural first checks if a command is on the top of the stack. If there is, the NEXT prompt is suppressed and the command is read and deleted from the stack; the command is then executed as if it had been entered manually in response to the NEXT prompt.

If an INPUT statement containing input fields is being executed, Natural first checks if there are any input data on the top of the stack. If there are, these data are passed to the INPUT statement (in delimiter mode); the data read from the stack must be format-compatible with the variables in the INPUT statement; the data are then deleted from the stack. See also *Processing Data from the Natural Stack* in the INPUT statement description.

If an INPUT statement was executed using data from the stack, and this INPUT statement is re-executed via a REINPUT statement, the INPUT statement screen will be re-executed displaying the same data from the stack as when it was executed originally. With the REINPUT statement, no further data are read from the stack.

When a Natural program terminates normally, the stack is flushed beginning from the top until either a command is on the top of the stack or the stack is cleared. When a Natural program is terminated via the terminal command %% or with an error, the stack is cleared entirely.

Placing Data on the Stack

The following methods can be used to place data/commands on the stack:

- STACK Parameter
- STACK Statement
- FETCH and RUN Statements

STACK Parameter

The Natural profile parameter `STACK` may be used to place data/commands on the stack. The `STACK` parameter (described in the *Parameter Reference*) can be specified by the Natural administrator in the Natural parameter module at the installation of Natural; or you can specify it as a dynamic parameter when you invoke Natural.

When data/commands are to be placed on the stack via the `STACK` parameter, multiple commands must be separated from one another by a semicolon (;). If a command is to be passed within a sequence of data or command elements, it must be preceded by a semicolon.

Data for multiple `INPUT` statements must be separated from one another by a colon (:). Data that are to be read by a separate `INPUT` statement must be preceded by a colon. If a command is to be stacked which requires parameters, no colon is to be placed between the command and the parameters.

Semicolon and colon must not be used within the input data themselves as they will be interpreted as separation characters.

STACK Statement

The `STACK` statement can be used within a program to place data/commands in the stack. The data elements specified in one `STACK` statement will be used for one `INPUT` statement, which means that if data for multiple `INPUT` statements are to be placed on the stack, multiple `STACK` statements must be used.

Data may be placed on the stack either unformatted or formatted:

- If unformatted data are read from the stack, the data string is interpreted in delimiter mode and the characters specified with the session parameters `IA` (Input Assignment character) and `ID` (Input Delimiter character) are processed as control characters for **keyword** assignment and data separation.
- If formatted data are placed on the stack, each content of a field will be separated and passed to one input field in the corresponding `INPUT` statement. If the data to be placed on the stack contains delimiter, control or DBCS characters, it should be placed formatted on the stack to avoid unintentional interpretation of these characters.

See the *Statements* documentation for further information on the STACK statement.

FETCH and RUN Statements

The execution of a FETCH or RUN statement that contains parameters to be passed to the invoked program will result in these parameters being placed on top of the stack.

Clearing the Stack

The contents of the stack can be deleted with the RELEASE statement. See the *Statements* documentation for details on the RELEASE statement.

Note: When a Natural program is terminated via the terminal command %% or with an error, the stack is cleared entirely.

46 Processing of Date Information

- Edit Masks for Date Fields and Date System Variables 450
- Default Edit Mask for Date - DTFORM Parameter 450
- Date Format for Alphanumeric Representation - DF Parameter 451
- Date Format for Output - DFOUT Parameter 453
- Date Format for Stack - DFSTACK Parameter 454
- Year Sliding Window - YSLW Parameter 455
- Combinations of DFSTACK and YSLW 457
- Year Fixed Window 459
- Date Format for Default Page Title - DFTITLE Parameter 459

This chapter covers various aspects concerning the handling of date information in Natural applications.

Edit Masks for Date Fields and Date System Variables

If you wish the value of a date field to be output in a specific representation, you usually specify an **edit mask** for the field. With an edit mask, you determine character by character what the output is to look like.

If you wish to use the current date in a specific representation, you need not define a date field and specify an edit mask for it; instead you can simply use a *date system variable*. Natural provides various date system variables, which contain the current date in different representations. Some of these representations contain a 2-digit year component, some a 4-digit year component.

For more information and a list of all date system variables, see the *System Variables* documentation.

Default Edit Mask for Date - DTFORM Parameter

The profile parameter `DTFORM` determines the default format used for dates as part of the default title on Natural reports, for date constants and for date input.

This date format determines the sequence of the day, month and year components of a date, as well as the delimiter characters to be used between these components.

Possible `DTFORM` settings are:

Setting	Date Format*	Example
`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, *mm* = month, *yyyy* = year.

The `DTFORM` parameter can be set in the Natural parameter module/file or dynamically when Natural is invoked. By default, `DTFORM=I` applies.

Date Format for Alphanumeric Representation - DF Parameter

If an edit mask is specified, the representation of the field value is determined by the edit mask. If no edit mask is specified, the representation of the field value is determined by the session parameter DF in combination with the profile parameter DTFORM.

With the DF parameter, you can choose one of the following date representations:

DF=S	8-byte representation with 2-digit year component and delimiters (*yy-mm-dd*).
DF=I	8-byte representation with 4-digit year component without delimiters (*yyyymmdd*).
DF=L	10-byte representation with 4-digit year component and delimiters (*yyyy-mm-dd*).

For each representation, the sequence of the day, month and year components, and the delimiter characters used, are determined by the DTFORM parameter.

By default, DF=S applies (except for INPUT statements; see below).

The session parameter DF is evaluated at compilation.

It can be specified with the following statements:

- FORMAT,
- INPUT, DISPLAY, WRITE and PRINT at statement and element (field) level,
- MOVE, COMPRESS, STACK, RUN and FETCH at element (field) level.

When specified in one of these statements, the DF parameter applies to the following:

Statement	Effect of DF parameter
DISPLAY, WRITE, PRINT	When the value of a date variable is output with one of these statements, the value is converted to an alphanumeric representation before it is output. The DF parameter determines which representation is used.
MOVE, COMPRESS	When the value of a date variable is transferred to an alphanumeric field with a MOVE or COMPRESS statement, the value is converted to an alphanumeric representation before it is transferred. The DF parameter determines which representation is used.
STACK, RUN, FETCH	When the value of a date variable is placed on the stack, it is converted to alphanumeric representation before it is placed on the stack. The DF parameter determines which representation is used. The same applies when a date variable is specified as a parameter in a FETCH or RUN statement (as these parameters are also passed via the stack).

Statement	Effect of DF parameter
INPUT	When a data variable is used in an INPUT statement, the DF parameter determines how a value must be entered in the field. However, when a date variable for which *no* DF parameter is specified is used in an INPUT statement, the date can be entered either with a 2-digit year component and delimiters or with a 4-digit year component and no delimiters. In this case, too, the sequence of the day, month and year components, and the delimiter characters to be used, are determined by the DTFORM parameter.

Note: With DF=S, only 2 digits are provided for the year information; this means that if a date value contained the century, this information would be lost during the conversion. To retain the century information, you set DF=I or DF=L.

Examples of DF Parameter with WRITE Statements

These examples assume that DTFORM=G applies.

```
/* 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
```

Example of DF Parameter with MOVE Statement

This example assumes that DTFORM=E applies.

```
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
...
```

Example of DF Parameter with STACK Statement

This example assumes that DTFORM=I applies.

```
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
...
```

Example of DF Parameter with INPUT Statement

This example assumes that DTFORM=I applies.

```
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
...
```

Date Format for Output - DFOUT Parameter

The session/profile parameter DFOUT only applies to date fields in INPUT, DISPLAY, WRITE and PRINT statements for which no edit mask is specified, and for which no DF parameter applies.

For date fields which are displayed by INPUT, DISPLAY, PRINT and WRITE statements and for which neither an edit mask is specified nor a DF parameter applies, the profile/session parameter DFOUT determines the format in which the field values are displayed.

Possible DFOUT settings are:

DFOUT=S	Date variables are displayed with a 2-digit year component, and delimiters as determined by the DTFORM parameter (*yy-mm-dd*).
DFOUT=I	Date variables are displayed with a 4-digit year component and no delimiters (*yyyymmdd*).

By default, DFOUT=S applies. For either DFOUT setting, the sequence of the day, month and year components in the date values is determined by the DTFORM parameter.

The lengths of the date fields are not affected by the DFOUT setting, as either date value representation fits into an 8-byte field.

The DFOUT parameter can be set in the Natural parameter module/file, dynamically when Natural is invoked, or at session level with the system command GLOBALS. It is evaluated at runtime.

Example:

This example assumes that DTFORM=I applies.

```
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
...
```

Date Format for Stack - DFSTACK Parameter

The session/profile parameter DFSTACK only applies to date fields used in STACK, FETCH and RUN statements for which no DF parameter has been specified.

The DFSTACK parameter determines the format in which the values of date variables are placed on the stack via a STACK, RUN or FETCH statement.

Possible DFSTACK settings are:

DFSTACK=S	Date variables are placed on the stack with a 2-digit year component, and delimiters as determined by the profile parameter DTFORM (*yy-mm-dd*).
DFSTACK=C	Same as DFSTACK=S. However, a change in the century will be intercepted at runtime.
DFSTACK=I	Date variables are placed on the stack with a 4-digit year component and no delimiters (*yyyymmdd*).

By default, DFSTACK=S applies. DFSTACK=S means that when a date value is placed on the stack, it is placed there without the century information (which is lost). When the value is then read from the stack and placed into another date variable, the century is either assumed to be the current

one or determined by the setting of the YSLW parameter (see **below**). This might lead to the century being different from that of the original date value; however, Natural would not issue any error in this case.

DFSTACK=C works the same as DFSTACK=S in that a date value is placed on the stack without the century information. However, if the value is read from the stack and the resulting century is different from that of the original date value (either because of the YSLW parameter, or the original century not being the current one), Natural issues a runtime error.

Note: This runtime error is already issued at the time when the value is placed on the stack.

DFSTACK=I allows you to place a date value on the stack in a length of 8 bytes without losing the century information.

The DFSTACK parameter can be set in the Natural parameter module/file, dynamically when Natural is invoked, or at session level with the system command GLOBALS. It is evaluated at runtime.

Example:

This example assumes that DTFORM=I and YSLW=0 apply.

```
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 - YSLW Parameter

The profile parameter YSLW allows you determine the century of a 2-digit year value.

The YSLW parameter can be set in the Natural parameter module/file or dynamically when Natural is invoked. It is evaluated at runtime when an alphanumeric date value with a 2-digit year component is moved into a date variable. This applies to data values which are:

- used with the **mathematical function** VAL*(field)*,
- used with the IS(D) option in a logical condition,

- read from the **stack** as input data, or
- entered in an input field as input data.

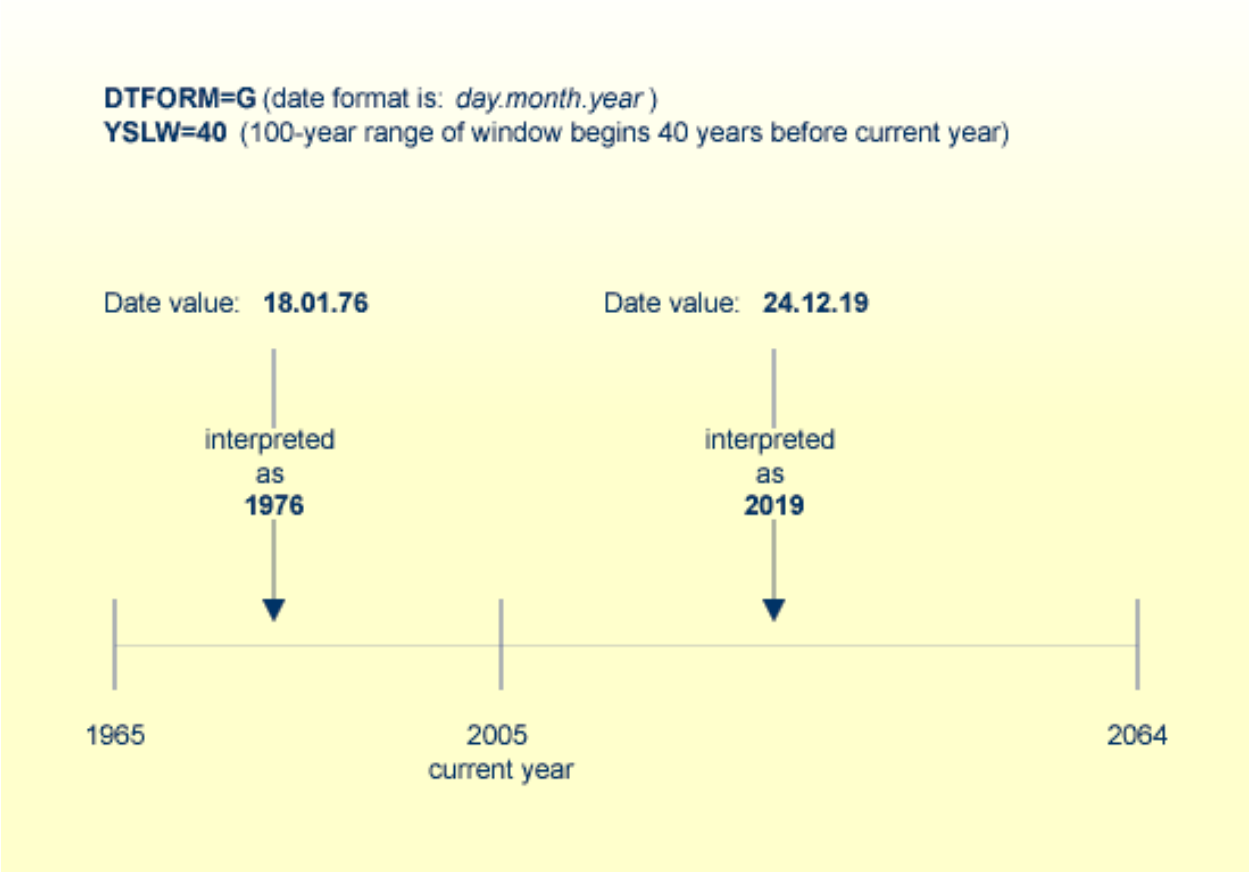
The YSLW parameter determines the range of years covered by a so-called “year sliding window”. The sliding-window mechanism assumes a date with a 2-digit year to be within a “window” of 100 years. Within these 100 years, every 2-digit year value can be uniquely related to a specific century.

With the YSLW parameter, you determine how many years in the past that 100-year range is to begin: The YSLW value is subtracted from the current year to determine the first year of the window range.

Possible values of the YSLW parameter are 0 to 99. The default value is YSLW=0, which means that no sliding-window mechanism is used; that is, a date with a 2-digit year is assumed to be in the current century.

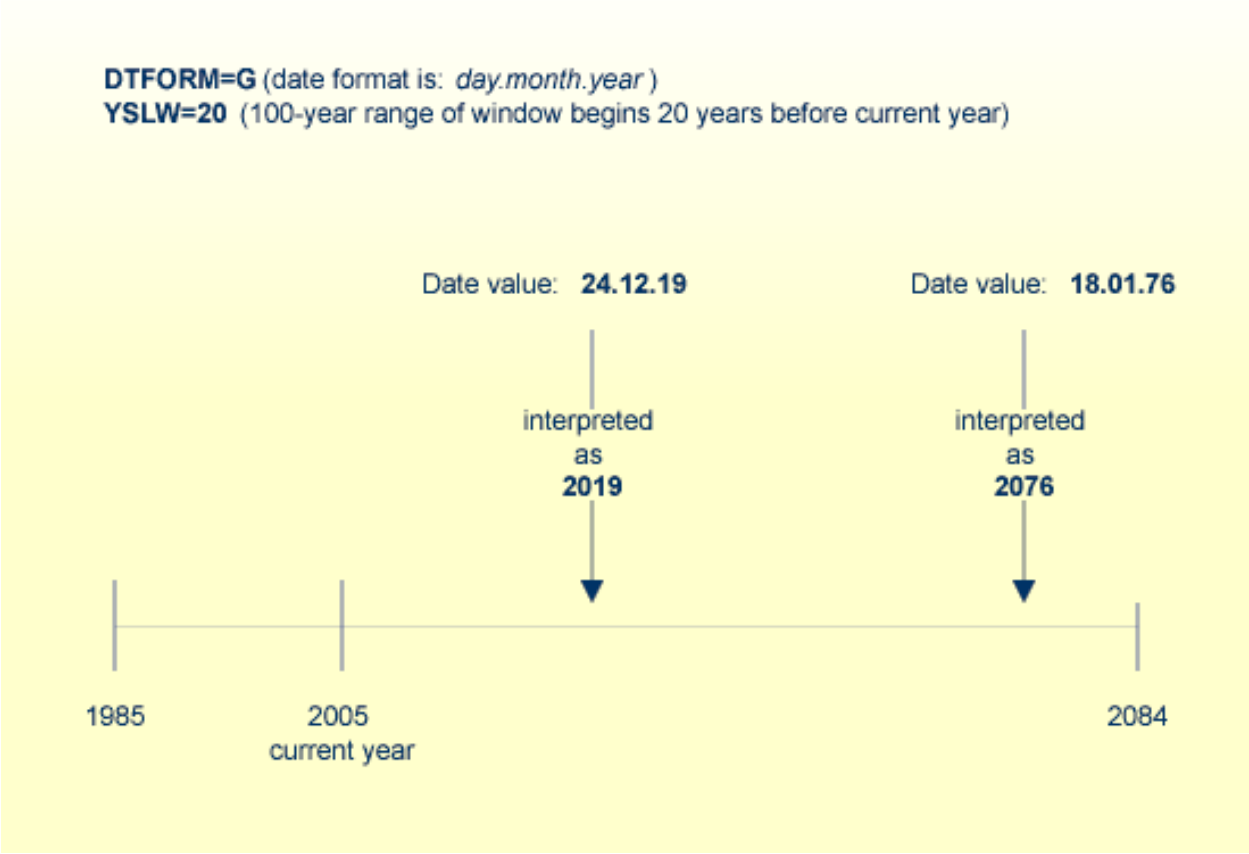
Example 1:

If the current year is 2005 and you specify YSLW=40, the sliding window will cover the years 1965 to 2064. A 2-digit year value *nn* from 65 to 99 is interpreted accordingly as 19*nn*, while a 2-digit year value *nn* from 00 to 64 is interpreted as 20*nn*.



Example 2:

If the current year is 2005 and you specify `YSLW=20`, the sliding window will cover the years 1985 to 2084. A 2-digit year value *nn* from 85 to 99 is interpreted accordingly as 19*nn*, while a 2-digit year value *nn* from 00 to 84 is interpreted as 20*nn*.



Combinations of DFSTACK and YSLW

The following examples illustrate the effects of using various combinations of the parameters `DFSTACK` and `YSLW`.

Note: All these examples assume that `DTFORM=I` applies.

Example 1:

This example assumes the current year to be 2005, and that the parameter settings `DFSTACK=S` (default) and `YSLW=20` apply.

```
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 this case, the year sliding window is not set appropriately, so that the century information is (inadvertently) changed.

Example 2:

This example assumes the current year to be 2005, and that the parameter settings `DFSTACK=S` (default) and `YSLW=60` apply.

```
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 this case, the year sliding window is set appropriately, so that the original century information is correctly restored.

Example 3:

This example assumes the current year to be 2005, and that the parameter settings `DFSTACK=C` and `YSLW=0` (default) apply.

```

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 this case, the century information is (inadvertently) changed. However, this change is intercepted by the DFSTACK=C setting.

Example 4:

This example assumes the current year to be 2005, and that the parameter settings DFSTACK=C and YSLW=60 (default) apply.

```
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 this case, the century information is changed due to the year sliding window. However, this change is intercepted by the DFSTACK=C setting.

Year Fixed Window

For information on this topic, see the description of the profile parameter YSLW.

Date Format for Default Page Title - DFTITLE Parameter

The session/profile parameter DFTITLE determines the format of the date in a default **page title** (as output with a DISPLAY, WRITE or PRINT statement).

DFTITLE=S	The date is output with a 2-digit year component and delimiters (*yy-mm-dd*).
DFTITLE=L	The date is output with a 4-digit year component and delimiters (*yyyy-mm-dd*).
DFTITLE=I	The date is output with a 4-digit year component and no delimiters (*yyyymmdd*).

For each of these output formats, the sequence of the day, month and year components, and the delimiter characters used, are determined by the DTFORM parameter.

The DFTITLE parameter can be set in the Natural parameter module/file, dynamically when Natural is invoked, or at session level with the system command GLOBALS. It is evaluated at runtime.

Example:

This example assumes that `DTFORM=I` applies.

```
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
```

Note: The `DFTITLE` parameter has no effect on a user-defined page title as specified with a `WRITE TITLE` statement.

47 Text Notation

- Defining a Text to Be Used with a Statement - the 'text' Notation 462
- Defining a Character to Be Displayed n Times before a Field Value - the 'c'(n) Notation 463

In an INPUT, DISPLAY, WRITE, WRITE TITLE or WRITE TRAILER statement, you can use text notation to define a text to be used in conjunction with such a statement.

Defining a Text to Be Used with a Statement - the 'text' Notation

The text to be used with the statement (for example, a prompting message) must be enclosed in either apostrophes (') or quotation marks ("). Do not confuse double apostrophes ('') with a quotation mark (").

Text enclosed in quotation marks can be converted automatically from lower-case letters to upper case. To switch off automatic conversion, change the settings in the editor profile.

For details, see *Program Editor Options*, **Ignore text constants** and **Uppercase translation** (in *Using Natural Studio*).

The text itself may be 1 to 72 characters and must not be continued from one line to the next.

Text elements may be concatenated by using a hyphen.

Examples:

```
DEFINE DATA LOCAL
1 #A(A10)
END-DEFINE

INPUT 'Input XYZ' (CD=BL) #A
WRITE '=' #A
WRITE 'Write1 ' - 'Write2 ' - 'Write3' (CD=RE)
END
```

Using Apostrophes as Part of a Text String

The following applies, if Natural profile parameter TQMARK (Translate Quotation Marks) is set to ON. This is the default setting.

If you want an apostrophe to be part of a text string that is enclosed in apostrophes, you must write this as double apostrophes ('') or as a quotation mark ("). Either notation will be output as a single apostrophe.

If you want an apostrophe to be part of a text string that is enclosed in quotation marks, you write this as a single apostrophe.

Examples of Apostrophe:

```
#FIELDA = 'O''CONNOR'
#FIELDA = 'O"CONNOR'
#FIELDA = "O'CONNOR"
```

In all three cases, the result will be:

```
O'CONNOR
```

Using Quotation Marks as Part of a Text String

The following applies, if the Natural profile parameter `TQ` (Translate Quotation Marks) is set to `OFF`. The default setting is `TQ=ON`.

If you want a quotation mark to be part of a text string that is enclosed in single apostrophes, write a quotation mark.

If you want a quotation mark to be part of a text string that is enclosed in quotation marks, write double quotation marks ("").

Example of Quotation Mark:

```
#FIELDA = 'O"CONNOR'
#FIELDA = "O""CONNOR"
```

In both cases, the result will be:

```
O"CONNOR
```

Defining a Character to Be Displayed n Times before a Field Value - the 'c'(n) Notation

If a single character is to be output several times as text, you use the following notation:

```
'c'(n)
```

As *c* you specify the character, and as *n* the number of times the character is to be generated. The maximum value for *n* is `249`.

Example:

```
WRITE '*'(3)
```

Instead of apostrophes before and after the character *c* you can also use quotation marks.

48 User Comments

- Using an Entire Source Code Line for Comments 466
- Using the Latter Part of a Source Code Line for Comments 467

User comments are descriptions or explanatory notes added to or interspersed among the statements of the source code. Such information may be particularly helpful in understanding and maintenaing source code that was written or edited by another programmer. Also, the characters marking the beginning of a comment can be used to temporarily disable the function of a statement or several source code lines for test purposes.

Using an Entire Source Code Line for Comments

If you wish to use an entire source-code line for a user comment, you enter one of the following at the beginning of the line:

- an asterisk and a blank (*),
- two asterisks (**), or
- a slash and an asterisk (/*).

```
*  USER COMMENT
** USER COMMENT
/* USER COMMENT
```

Example:

As can be seen from the following example, comment lines may also be used to provide for a clear source code structure.

```
** 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
```

Using the Latter Part of a Source Code Line for Comments

If you wish to use only the latter part of a source-code line for a user comment, you enter a blank, a slash and an asterisk (/*); the remainder of the line after this notation is thus marked as a comment:

```
ADD 5 TO #A          /* USER COMMENT
```

Example:

```
** 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
```

49 Logical Condition Criteria

- Introduction 470
- Relational Expression 471
- Extended Relational Expression 475
- Evaluation of a Logical Variable 476
- Fields Used within Logical Condition Criteria 477
- Logical Operators in Complex Logical Expressions 479
- BREAK Option - Compare Current Value with Value of Previous Loop Pass 480
- IS Option - Check whether Content of Alphanumeric or Unicode Field can be Converted 482
- MASK Option - Check Selected Positions of a Field for Specific Content 484
- MASK Option Compared with IS Option 491
- MODIFIED Option - Check whether Field Content has been Modified 493
- SCAN Option - Scan for a Value within a Field 494
- SPECIFIED Option - Check whether a Value is Passed for an Optional Parameter 496

This chapter describes purpose and use of logical condition criteria that can be used in the statements FIND, READ, HISTOGRAM, ACCEPT/REJECT, IF, DECIDE FOR, REPEAT.

Introduction

The basic criterion is a **relational expression**. Multiple relational expressions may be combined with logical operators (AND, OR) to form complex criteria.

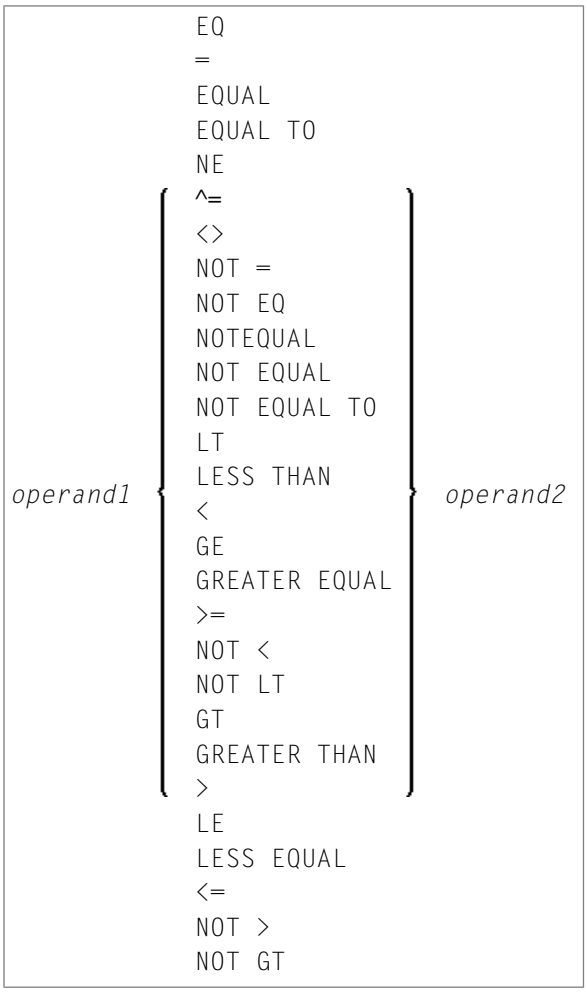
Arithmetic expressions may also be used to form a relational expression.

Logical condition criteria can be used in the following statements:

Statement	Usage
FIND	A WHERE clause containing logical condition criteria may be used to indicate criteria in addition to the basic selection criteria as specified in the WITH clause. The logical condition criteria specified with the WHERE clause are evaluated after the record has been selected and read. In a WITH clause, "basic search criteria" (as described with the FIND statement) are used, but not logical condition criteria.
READ	A WHERE clause containing logical condition criteria may be used to specify whether a record that has just been read is to be processed. The logical condition criteria are evaluated after the record has been read.
HISTOGRAM	A WHERE clause containing logical condition criteria may be used to specify whether the value that has just been read is to be processed. The logical condition criteria are evaluated after the value has been read.
ACCEPT/REJECT	An IF clause may be used with an ACCEPT or REJECT statement to specify logical condition criteria in addition to that specified when the record was selected/read with a FIND, READ, or HISTOGRAM statement. The logical condition criteria are evaluated after the record has been read and after record processing has started.
IF	Logical condition criteria are used to control statement execution.
DECIDE FOR	Logical condition criteria are used to control statement execution.
REPEAT	The UNTIL or WHILE clause of a REPEAT statement contain logical condition criteria which determine when a processing loop is to be terminated.

Relational Expression

Syntax:



Operand Definition Table:

Operand	Possible Structure						Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A		N	E	A	U	N	P	I	F	B	D	T	L		G	O	yes	yes
operand2	C	S	A		N	E	A	U	N	P	I	F	B	D	T	L		G	O	yes	no

For an explanation of the Operand Definition Table shown above, see *Syntax Symbols and Operand Definition Tables* in the *Statements* documentation.

In the "Possible Structure" column of the table above, "E" stands for arithmetic expressions; that is, any arithmetic expression may be specified as an operand within the relational expression. For

further information on arithmetic expressions, see *arithmetic-expression* in the COMPUTE statement description.

Explanation of the comparison operators:

Comparison Operator	Explanation
EQ = EQUAL EQUAL TO	equal to
NE ^= <> NOT = NOT EQ NOTEQUAL NOT EQUAL NOT EQUAL TO	not equal to
LT LESS THAN <	less than
GE GREATER EQUAL >=	greater than or equal to
NOT < NOT LT	not less than
GT GREATER THAN >	greater than
LE LESS EQUAL <=	less than or equal to
NOT > NOT GT	not greater than

Examples of Relational Expressions:

```
IF NAME = 'SMITH'
IF LEAVE-DUE GT 40
IF NAME = #NAME
```

For information on comparing arrays in a relational expression, see ***Processing of Arrays***.

Note: If a floating-point operand is used, comparison is performed in floating point. **Floating-point numbers** as such have only a limited precision; therefore, rounding/truncation

errors cannot be precluded when numbers are converted to/from floating-point representation.

Arithmetic Expressions in Logical Conditions

The following example shows how arithmetic expressions can be used in logical conditions:

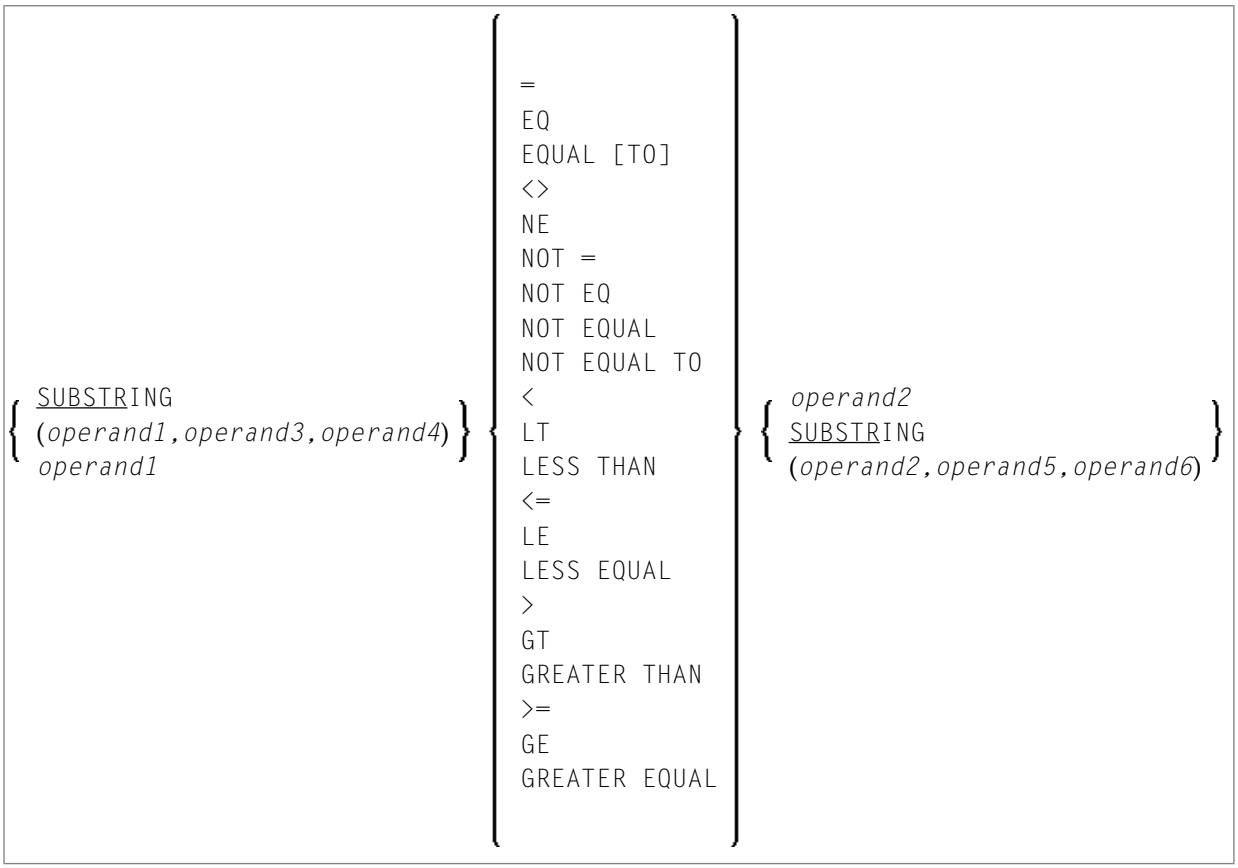
```
IF #A + 3 GT #B - 5 AND #C * 3 LE #A + #B
```

Handles in Logical Conditions

If the operands in a relation expression are handles, only `EQUAL` and `NOT EQUAL` operators may be used.

SUBSTRING Option in Relational Expression

Syntax:



Operand Definition Table:

Operand	Possible Structure						Possible Formats												Referencing Permitted	Dynamic Definition
operand1	C	S	A		N		A	U					B						yes	yes
operand2	C	S	A		N		A	U					B						yes	no
operand3	C	S							N	P	I		B						yes	no
operand4	C	S							N	P	I								yes	no
operand5	C	S							N	P	I								yes	no
operand6	C	S							N	P	I								yes	no

With the SUBSTRING option, you can compare a *part* of an alphanumeric, a binary or a Unicode field. After the field name (*operand1*) you specify first the starting position (*operand3*) and then the **length** (*operand4*) of the field portion to be compared.

Also, you can compare a field value with part of another field value. After the field name (*operand2*) you specify first the starting position (*operand5*) and then the length (*operand6*) of the field portion *operand1* is to be compared with.

You can also combine both forms, that is, you can specify a SUBSTRING for both *operand1* and *operand2*.

Examples:

The following expression compares the 5th to 12th position inclusive of the value in field #A with the value of field #B:

```
SUBSTRING(#A,5,8) = #B
```

- where 5 is the starting position and 8 is the length.

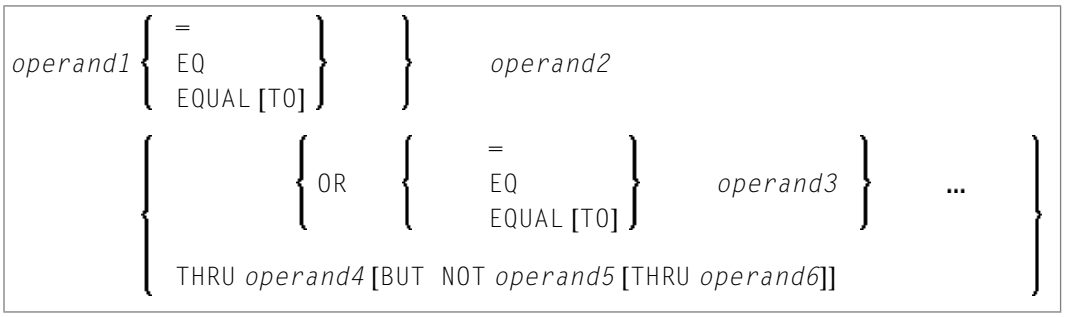
The following expression compares the value of field #A with the 3rd to 6th position inclusive of the value in field #B:

```
#A = SUBSTRING(#B,3,4)
```

Note: If you omit *operand3*/*operand5*, the starting position is assumed to be 1. If you omit *operand4*/*operand6*, the length is assumed to be from the starting position to the end of the field.

Extended Relational Expression

Syntax:



Operand Definition Table:

Operand	Possible Structure						Possible Formats												Referencing Permitted	Dynamic Definition
operand1	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	yes	no
operand2	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	yes	no
operand3	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	yes	no
operand4	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	yes	no
operand5	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	yes	no
operand6	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	yes	no

* Mathematical functions and system variables are permitted. Break functions are not permitted.

operand3 can also be specified using a MASK or SCAN option; that is, it can be specified as:

```
MASK (mask-definition) [operand]
MASK operand
SCAN operand
```

For details on these options, see the sections ***MASK Option*** and ***SCAN Option***.

Examples:

```
IF #A = 2 OR = 4 OR = 7
IF #A = 5 THRU 11 BUT NOT 7 THRU 8
```

Evaluation of a Logical Variable

Syntax:

operand1

This option is used in conjunction with a logical variable (format L). A logical variable may take the value TRUE or FALSE. As *operand1* you specify the name of the logical variable to be used.

Operand Definition Table:

Operand	Possible Structure						Possible Formats											Referencing Permitted	Dynamic Definition
operand1	C	S	A											L				no	no

Example of Logical 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
```

Output of Program 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
```

Fields Used within Logical Condition Criteria

Database fields and user-defined variables may be used to construct logical condition criteria. A database field which is a multiple-value field or is contained in a periodic group can also be used. If a range of values for a multiple-value field or a range of occurrences for a periodic group is specified, the condition is true if the search value is found in any value/occurrence within the specified range.

Each value used must be compatible with the field used on the opposite side of the expression. Decimal notation may be specified only for values used with numeric fields, and the number of decimal positions of the value must agree with the number of decimal positions defined for the field.

If the operands are not of the same format, the second operand is converted to the format of the first operand.

Note: A numeric constant without decimal point notation is stored with format I for the values `-2147483648` to `+2147483647`, see ***Numeric Constants***. Consequently the comparison with such an integer constant as *operand1* is performed by converting *operand2* to a integer value. This means that the digits after the decimal point of *operand2* are not considered due to truncation.

Example:

```
 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 ↩
```

The following table shows which operand formats can be used together in a logical condition:

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									
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

Notes:

1. [1] where GH = GUI handle, OH = object handle.
2. [2] The binary value will be assumed to contain Unicode code points, and the comparison is performed as for a comparison of two Unicode values. The length of the binary field must be even.

If two values are compared as alphanumeric values, the shorter value is assumed to be extended with trailing blanks in order to get the same length as the longer value.

If two values are compared as binary values, the shorter value is assumed to be extended with leading binary zeroes in order to get the same length as the longer value.

If two values are compared as Unicode values, trailing blanks are removed from both values before the ICU collation algorithm is used to compare the two resulting values. See also *Logical Condition Criteria* in the *Unicode and Code Page Support* documentation.

Comparison Examples:

```
A1(A1) := 'A'
A5(A5) := 'A    '
B1(B1) := H'FF'
B5(B5) := H'00000000FF'
U1(U1) := UH'00E4'
U2(U2) := UH'00610308'
IF A1 = A5 THEN ...                 /* TRUE
IF B1 = B5 THEN ...                 /* TRUE
IF U1 = U2 THEN ...                 /* TRUE ↩
```

If an array is compared with a scalar value, each element of the array will be compared with the scalar value. The condition will be true if at least one of the array elements meets the condition (`OR` operation).

If an array is compared with an array, each element in the array is compared with the corresponding element of the other array. The result is true only if all element comparisons meet the condition (`AND` operation).

See also ***Processing of Arrays***.

Note: An Adabas phonetic descriptor cannot be used within a logical condition.

Examples of Logical Condition Criteria:

```
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
```

Logical Operators in Complex Logical Expressions

Logical condition criteria may be combined using the Boolean operators `AND`, `OR`, and `NOT`. Parentheses may also be used to indicate logical grouping.

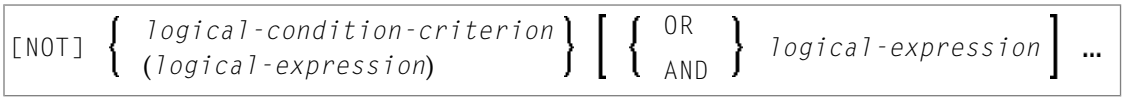
The operators are evaluated in the following order:

Priority	Operator	Meaning
1	( )	Parentheses
2	NOT	Negation
3	AND	AND operation
4	OR	OR operation

The following *logical-condition-criteria* may be combined by logical operators to form a complex *logical-expression*:

- **Relational expressions**
- **Extended relational expressions**
- **MASK option**
- **SCAN option**
- **BREAK option**

The syntax for a *logical-expression* is as follows:



Examples of Logical Expressions:

```
FIND STAFF-VIEW WITH CITY = 'TOKYO'
     WHERE BIRTH GT 19610101 AND SEX = 'F'
IF NOT (#CITY = 'A' THRU 'E')
```

For information on comparing arrays in a logical expression, see *Processing of Arrays*.

Note: If multiple logical-condition-criteria are connected with AND, the evaluation terminates as soon as the first of these criteria is not true.

BREAK Option - Compare Current Value with Value of Previous Loop Pass

The BREAK option allows the current value or a portion of a value of a field to be compared with the value contained in the same field in the previous pass through the processing loop.

Syntax:

```
BREAK [OF] operand1 [/n/]
```

Operand Definition Table:

Operand	Possible Structure						Possible Formats												Referencing Permitted	Dynamic Definition
operand1		S					A	U	N	P	I	F	B	D	T	L			yes	no

Syntax Element Description:

operand1	Specifies the control field which is to be checked. A specific occurrence of an array can also be used as a control field.
/n/	The notation */n/* may be used to indicate that only the first *n* positions (counting from left to right) of the control field are to be checked for a change in value. This notation can only be used with operands of format A, B, N, or P. The result of the BREAK operation is true when a change in the specified positions of the field occurs. The result of the BREAK operation is not true if an AT END OF DATA condition occurs. Example: In this example, a check is made for a different value in the first position of the field FIRST-NAME. `BREAK FIRST-NAME /1/` Natural system functions (which are available with the AT BREAK statement) are not available with this option.

Example of 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
```

Output of Program `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 - Check whether Content of Alphanumeric or Unicode Field can be Converted

Syntax:

operand1 IS (*format*)

This option is used to check whether the content of an alphanumeric or Unicode field (*operand1*) can be converted to a specific other format.

Operand Definition Table:

Operand	Possible Structure						Possible Formats												Referencing Permitted	Dynamic Definition
operand1	C	S	A		N		A	U											yes	no

The *format* for which the check is performed can be:

N*ll.ll*	Numeric with length *ll.ll*.
F*ll*	Floating point with length *ll*.
D	Date. The following date formats are possible: *dd-mm-yy*, *dd-mm-yyyy*, *ddmmyyyy* (*dd* = day, *mm* = month, *yy* or *yyyy* = year). The sequence of the day, month and year components as well as the characters between the components are determined by the profile parameter DTFORM (which is described in the *Parameter Reference*).
T	Time (according to the default time display format).
P*ll.ll*	Packed numeric with length *ll.ll*.
I*ll*	Integer with length *ll*.

When the check is performed, leading and trailing blanks in *operand1* will be ignored.

The IS option may, for example, be used to check the content of a field before the mathematical function VAL (extract numeric value from an alphanumeric field) is used to ensure that it will not result in a runtime error.

Note: The IS option cannot be used to check if the value of an alphanumeric field is in the specified "format", but if it can be *converted* to that "format". To check if a value is in a specific format, you can use the MASK option. For further information, see ***MASK Option Compared with IS Option*** and ***Checking Packed or Unpacked Numeric Data***.

Example of 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
```

Output of Program LOGICX04:

```
#DATE 150487    #FIELDA









INPUT IS NOT IN DATE FORMAT (YY-MM-DD)
```

MASK Option - Check Selected Positions of a Field for Specific Content

With the MASK option, you can check selected positions of a field for specific content.

The following topics are covered below:

- Constant Mask
- Variable Mask
- Characters in a Mask
- Mask Length
- Checking Dates
- Checking Against the Content of Constants or Variables
- Range Checks
- Checking Packed or Unpacked Numeric Data

Constant Mask

Syntax:

```
operand1 { =            } MASK (mask-definition) [operand2]
         { EQ           }
         { EQUAL TO     }
         { NE           }
         { NOT EQUAL    }
```

Operand Definition Table:

Operand	Possible Structure						Possible Formats												Referencing Permitted	Dynamic Definition
operand1	C	S	A		N		A	U	N	P									yes	no
operand2	C	S					A	U	N	P			B						yes	no

operand2 can only be used if the *mask-definition* contains at least one X. *operand1* and *operand2* must be format-compatible:

- If *operand1* is of format A, *operand2* must be of format A, B, N or U.
- If *operand1* is of format U, *operand2* must be of format A, B, N or U.
- If *operand1* is of format N or P, *operand2* must be of format N or P.

An X in the *mask-definition* selects the corresponding positions of the content of *operand1* and *operand2* for comparison.

Variable Mask

Apart from a constant *mask-definition* (see above), you may also specify a variable mask definition.

Syntax:

```
operand1 { =           } MASK operand2
         { EQ          }
         { EQUAL TO    }
         { NE          }
         { NOT EQUAL   }
```

Operand Definition Table:

Operand	Possible Structure						Possible Formats												Referencing Permitted	Dynamic Definition
operand1	C	S	A		N		A	U	N	P									yes	no
operand2		S					A	U											yes	no

The content of *operand2* will be taken as the mask definition. Trailing blanks in *operand2* will be ignored.

- If *operand1* is of format A, N or P, *operand2* must be of format A.
- If *operand1* is of format U, *operand2* must be of format U.

Characters in a Mask

The following characters may be used within a mask definition (the mask definition is contained in *mask-definition* for a constant mask and *operand2* for a variable mask):

Character	Meaning
. or ? or _	A period, question mark or underscore indicates a single position that is not to be checked.
* or %	An asterisk or percent mark is used to indicate any number of positions not to be checked.
/	A slash (/) is used to check if a value ends with a specific character (or string of characters). For example, the following condition will be true if there is either an E in the last position of the field, or the last E in the field is followed by nothing but blanks: `IF #FIELD = MASK (*'E'/)`
A	The position is to be checked for an alphabetical character (upper or lower case).
'*c*'	One or more positions are to be checked for the characters bounded by apostrophes (a double apostrophe indicates that a single apostrophe is the character to be checked for). If *operand1* is in Unicode format, '*c*' must contain Unicode characters.
C	The position is to be checked for an alphabetical character (upper or lower case), a numeric character, or a blank.
DD	The two positions are to be checked for a valid day notation (01 - 31; dependent on the values of MM and YY/YYYY, if specified; see also ***Checking Dates***).
H	The position is to be checked for hexadecimal content (A - F, 0 - 9).
JJJ	The positions are to be checked for a valid Julian Day; that is, the day number in the year (001-366, dependent on the value of YY/YYYY, if specified. See also ***Checking Dates***.)
L	The position is to be checked for a lower-case alphabetical character (a - z).
MM	The positions are to be checked for a valid month (01 - 12); see also ***Checking Dates***.
N	The position is to be checked for a numeric digit.
n...	One (or more) positions are to be checked for a numeric value in the range 0 - *n*.
n1-*n2* or *n1*:*n2*	The positions are checked for a numeric value in the range *n*1-*n*2. *n*1 and *n*2 must be of the same length.
P	The position is to be checked for a displayable character (U, L, N or S).
S	The position is to be checked for special characters. See also *Support of Different Character Sets with NATCONV.INI* in the *Operations* documentation.
U	The position is to be checked for an upper-case alphabetical character (A - Z).
X	The position is to be checked against the equivalent position in the value (*operand2*) following the mask-definition. X is not allowed in a variable mask definition, as it makes no sense.
YY	The two positions are to be checked for a valid year (00 - 99). See also ***Checking Dates***.
YYYY	The four positions are checked for a valid year (0000 - 2699).

Character	Meaning
Z	The position is to be checked for a character whose left half-byte is hexadecimally 3 or 7, and whose right half-byte is hexadecimally 0 - 9. This may be used to correctly check for numeric digits in negative numbers. With N (which indicates a position to be checked for a numeric digit), a check for numeric digits in negative numbers leads to incorrect results, because the sign of the number is stored in the last digit of the number, causing that digit to be hexadecimally represented as non-numeric. Within a mask, use only one Z for each sequence of numeric digits that is checked.

Mask Length

The length of the mask determines how many positions are to be checked.

Example:

```
DEFINE DATA LOCAL
1 #CODE (A15)
END-DEFINE
...
IF #CODE = MASK (NN'ABC'....NN)
...
```

In the above example, the first two positions of #CODE are to be checked for numeric content. The three following positions are checked for the contents ABC. The next four positions are not to be checked. Positions ten and eleven are to be checked for numeric content. Positions twelve to fifteen are not to be checked.

Checking Dates

Only one date may be checked within a given mask. When the same date component (JJJ, DD, MM, YY or YYYY) is specified more than once in the mask, only the value of the last occurrence is checked for consistency with other date components.

When dates are checked for a day (DD) and no month (MM) is specified in the mask, the current month will be assumed.

When dates are checked for a day (DD) or a Julian day (JJJ) and no year (YY or YYYY) is specified in the mask, the current year will be assumed.

When dates are checked for a 2-digit year (YY), the current century will be assumed if no Sliding or Fixed Window is set. For more details about Sliding or Fixed Windows, refer to profile parameter YSLW in the *Parameter Reference*.

Example 1:

```
MOVE 1131 TO #DATE (N4)
IF #DATE = MASK (MMDD)
```

In this example, month and day are checked for validity. The value for month (11) will be considered valid, whereas the value for day (31) will be invalid since the 11th month has only 30 days.

Example 2:

```
IF #DATE(A8) = MASK (MM'/'DD'/'YY)
```

In this example, the content of the field `#DATE` is be checked for a valid date with the format `MM/DD/YY` (month/day/year).

Example 3:

```
IF #DATE (A8) = MASK (1950-2020MMDD)
```

In this example, the content of field `#DATE` is checked for a four-digit number in the range 1950 to 2020 followed by a valid month and day in the current year.

Note: Although apparent, the above mask does not allow to check for a valid date in the years 1950 through 2020, because the numeric value range 1950-2020 is checked independent of the validation of month and day. The check will deliver the intended results except for February, 29, where the result depends on whether the current year is a leap year or not. To check for a specific year range in addition to the date validation, code one check for the date validation and another for the range validation:

```
IF #DATE (A8) = MASK (YYYYMMDD) AND #DATE = MASK (1950-2020)
```

Example 4:

```
IF #DATE (A4) = MASK (19-20YY)
```

In this example, the content of field `#DATE` is checked for a two-digit number in the range 19 to 20 followed by a valid two-digit year (00 through 99). The century is supplied by Natural as described above.

Note: Although apparent, the above mask does not allow to check for a valid year in the range 1900 through 2099, because the numeric value range 19-20 is checked independent of the year validation. To check for year ranges, code one check for the date validation and another for the range validation:

```
IF #DATE (A10) = MASK (YYYY'-'MM'-'DD) AND #DATE = MASK (19-20)
```

Checking Against the Content of Constants or Variables

If the value for the mask check is to be taken from either a constant or a variable, this value (*operand2*) must be specified immediately following the *mask-definition*.

operand2 must be at least as long as the mask.

In the mask, you indicate each position to be checked with an X, and each position not to be checked with a period (.) or a question mark (?) or an underscore (_).

Example:

```
DEFINE DATA LOCAL
1 #NAME (A15)
END-DEFINE
...
IF #NAME = MASK (..XX) 'ABCD'
...
```

In the above example, it is checked whether the field #NAME contains CD in the third and fourth positions. Positions one and two are not checked.

The length of the mask determines how many positions are to be checked. The mask is left-justified against any field or constant used in the mask operation. The format of the field (or constant) on the right side of the expression must be the same as the format of the field on the left side of the expression.

If the field to be checked (*operand1*) is of format A, any constant used (*operand2*) must be enclosed in apostrophes. If the field is numeric, the value used must be a numeric constant or the content of a numeric database field or user-defined variable.

In either case, any characters/digits within the value specified whose positions do not match the X indicator within the mask are ignored.

The result of the MASK operation is true when the indicated positions in both values are identical.

Example:

```
** 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 the above example, the record will be accepted if the fifth and sixth positions of the field CITY each contain the character N.

Range Checks

When performing range checks, the number of positions verified in the supplied variable is defined by the precision of the value supplied in the mask specification. For example, a mask of (...193...) will verify positions 4 to 6 for a three-digit number in the range 000 to 193.

Additional Examples of Mask Definitions:

- In this example, each character of #NAME is checked for an alphabetical character:

```
IF #NAME (A10) = MASK (AAAAAAAAAA)
```

- In this example, positions 4 to 6 of #NUMBER are checked for a numeric value:

```
IF #NUMBER (A6) = MASK (...NNN)
```

- In this example, positions 4 to 6 of #VALUE are to be checked for the value 123:

```
IF #VALUE(A10) = MASK (...'123')
```

- This example will check if #LICENSE contains a license number which begins with NY- and whose last five characters are identical to the last five positions of #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
```

- The following condition would be met by any value which contains NAT and AL no matter which and how many other characters are between NAT and AL (this would include the values NATURAL and NATIONALITY as well as NATAL):

```
MASK('NAT'*'AL')
```

Checking Packed or Unpacked Numeric Data

Legacy applications often have packed or unpacked numeric variables redefined with alphanumeric or binary fields. Such redefinitions are not recommended, because using the packed or unpacked variable in an assignment or computation may lead to errors or unpredictable results. To validate the contents of such a redefined variable before the variable is used, use the N option (see *Characters in a Mask*) as many as number of digits - 1 times followed by a single Z option.

Examples :

```
IF #P1 (P1) = MASK (Z)
IF #N4 (N4) = MASK (NNNZ)
IF #P5 (P5) = MASK (NNNNZ)
```

For further information about checking field contents, see *MASK Option Compared with IS Option*.

MASK Option Compared with IS Option

This section points out the difference between `MASK` option and `IS` option and contains a sample program to illustrate the difference.

The `IS` option can be used to check whether the content of an alphanumeric or Unicode field can be converted to a specific other format, but it cannot be used to check if the value of an alphanumeric field is in the specified format.

The `MASK` option can be used to validate the contents of a redefined packed or unpacked numeric variable.

Example Illustrating the Difference:

```
** 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                                                                          ↩
```

Output of Program `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 - Check whether Field Content has been Modified

Syntax:

```
operand1 [NOT] MODIFIED
```

This option is used to determine whether the content of a field has been modified during the execution of an `INPUT` or `PROCESS PAGE` statement. As a precondition, a control variable must have been assigned using the parameter `CV`.

Operand Definition Table:

Operand	Possible Structure						Possible Formats										Referencing Permitted	Dynamic Definition
operand1		S	A												C		no	no

Attribute control variables referenced in an `INPUT` or `PROCESS PAGE` statement are always assigned the status “not modified” when the map is transmitted to the terminal.

Whenever the content of a field referencing an attribute control variable is modified, the attribute control variable has been assigned the status “modified”. When multiple fields reference the same attribute control variable, the variable is marked “modified” if any of these fields is modified.

If *operand1* is an array, the result will be true if at least one of the array elements has been assigned the status “modified” (`OR` operation).

Example of 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
```

Output of Program LOGICX06:

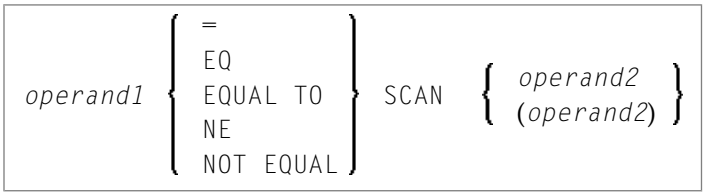
```
#A   #B
```

After entering any value and pressing ENTER, the following output is displayed:

```
FIELD #A OR #B HAS BEEN MODIFIED
```

SCAN Option - Scan for a Value within a Field

Syntax:



Operand Definition Table:

Operand	Possible Structure						Possible Formats												Referencing Permitted	Dynamic Definition
operand1	C	S	A		N		A	U	N	P									yes	no
operand2	C	S					A	U					B*						yes	no

* *operand2* may only be binary if *operand1* is of format A or U. If *operand1* is of format U and *operand2* is of format B, then the length of *operand2* must be even.

The SCAN option is used to scan for a specific value within a field.

The characters used in the SCAN option (*operand2*) may be specified as an alphanumeric or Unicode constant (a character string bounded by apostrophes) or the contents of an alphanumeric or Unicode database field or user-defined variable.

Caution: Trailing blanks are automatically eliminated from *operand1* and *operand2*. Therefore, the SCAN option cannot be used to scan for values containing trailing blanks. *operand1* and *operand2* may contain leading or embedded blanks. If *operand2* consists of blanks only, scanning will be assumed to be successful, regardless of the value of *operand1*; confer EXAMINE FULL statement if trailing blanks are not to be ignored in the scan operation.

The field to be scanned (*operand1*) may be of format A, N, P or U. The SCAN operation may be specified with the equal (EQ) or not equal (NE) operators.

The length of the character string for the SCAN operation should be less than the length of the field to be scanned. If the length of the character string specified is identical to the length of the field to be scanned, then an EQUAL operator should be used instead of SCAN.

Example of 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
```

Output of Program LOGICX02:

```
ENTER SCAN VALUE:  ↩
```

A scan for example for `LL` delivers three matches in 15 names:

```
        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 - Check whether a Value is Passed for an Optional Parameter

Syntax:

parameter-name [NOT] SPECIFIED

This option is used to check whether an optional parameter in an invoked object (subprogram, external subroutine, dialog or ActiveX control) has received a value from the invoking object or not.

An optional parameter is a field defined with the keyword `OPTIONAL` in the `DEFINE DATA PARAMETER` statement of the invoked object. If a field is defined as `OPTIONAL`, a value can - but need not - be passed from an invoking object to this field.

In the invoking statement, the notation *n*`X` is used to indicate parameters for which no values are passed.

If you process an optional parameter which has not received a value, this will cause a runtime error. To avoid such an error, you use the `SPECIFIED` option in the invoked object to check whether an optional parameter has received a value or not, and then only process it if it has.

parameter-name is the name of the parameter as specified in the DEFINE DATA PARAMETER statement of the invoked object.

For a field not defined as OPTIONAL, the SPECIFIED condition is always TRUE.

Example of SPECIFIED Option:

Calling Programming:

```
** 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  ↩
```

Subprogram Called:

```
** 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 ↩
```

Output of Program LOGICX07:

```
Page      1                                                    04-12-15  11:25:41

#PARM1: ABC
#PARM2 is not specified
#PARM3 is specified
#PARM3:  20
```

50 Rules for Arithmetic Assignment

- Field Initialization 500
- Data Transfer 500
- Field Truncation and Field Rounding 503
- Result Format and Length in Arithmetic Operations 503
- Arithmetic Operations with Floating-Point Numbers 504
- Arithmetic Operations with Date and Time 506
- Performance Considerations for Mixed Format Expressions 510
- Precision of Results of Arithmetic Operations 510
- Error Conditions in Arithmetic Operations 511
- Processing of Arrays 512

Field Initialization

A field (user-defined variable or database field) which is to be used as an operand in an arithmetic operation must be defined with one of the following formats:

Format	
N	Numeric unpacked
P	Packed numeric
I	Integer
F	Floating point
D	Date
T	Time

Note: For reporting mode: A field which is to be used as an operand in an arithmetic operation must have been previously defined. A user-defined variable or database field used as a result field in an arithmetic operation need not have been previously defined.

All user-defined variables and all database fields defined in a `DEFINE DATA` statement are initialized to the appropriate zero or blank value when the program is invoked for execution.

Data Transfer

Data transfer is performed with a `MOVE` or `COMPUTE` statement. The following table summarizes the data transfer compatibility of the formats an operand may take.

Sending Field Format	Receiving Field Format													
	N or P	A	U	B*n* (*n*<5)	B*n* (*n*>4)	I	L	C	D	T	F	G	O
N or 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	

Where:

Y	Indicates data transfer compatibility.
-	Indicates data transfer incompatibility.
[ ]	Numbers in brackets [ ] refer to the corresponding rule for data transfer given below.

Data Conversion

The following rules apply to converting data values:

1. **Alphanumeric to binary:**
   The value will be moved byte by byte from left to right. The result may be truncated or padded with trailing blank characters depending on the length defined and the number of bytes specified.
2. **(N,P,I) and binary (length 1-4) to alphanumeric:**
   The value will be converted to unpacked form and moved into the alphanumeric field left justified, that is, leading zeros will be suppressed and the field will be filled with trailing blank characters. For negative numeric values, the sign will be converted to the hexadecimal notation `Dx`. Any decimal point in the numeric value will be ignored. All digits before and after the decimal point will be treated as one integer value.
3. **(N,P,I,F) to binary (1-4 bytes):**
   The numeric value will be converted to binary (4 bytes). Any decimal point in the numeric value will be ignored (the digits of the value before and after the decimal point will be treated as an integer value). The resulting binary number will be positive or a two's complement of the number depending on the sign of the value.
4. **Binary (1-4 bytes) to numeric:**
   The value will be converted and assigned to the numeric value right justified, that is, with leading zeros. (Binary values of the length 1-3 bytes are always assumed to have a positive sign. For binary values of 4 bytes, the leftmost bit determines the sign of the number: 1=negative, 0=positive.) Any decimal point in the receiving numeric value will be ignored. All digits before and after the decimal point will be treated as one integer value.
5. **Binary to binary:**
   The value will be moved from right to left byte by byte. Leading binary zeros will be inserted into the receiving field.

6. **Binary (>4 bytes) to alphanumeric:**
   The value will be moved byte by byte from left to right. The result may be truncated or padded with trailing blanks depending on the length defined and the number of bytes specified.
7. **Date (D) to time (T):**
   If date is moved to time, it is converted to time assuming time 00:00:00:0.
8. **Time (T) to date (D):**
   If time is moved to date, the time information is truncated, leaving only the date information.
9. **L,D,T,F to A:**
   The values are converted to display form and are assigned left justified.
10. **F:**
    If F is assigned to an alphanumeric or Unicode field which is too short, the mantissa is reduced accordingly.
11. **Unicode to alphanumeric:**
    The Unicode value will be converted to alphanumeric character codes according to the default code page (value of the system variable `*CODEPAGE`) using the International Components for Unicode (ICU) library. The result may be truncated or padded with trailing blank characters, depending on the length defined and the number of bytes specified.
12. **Unicode to binary:**
    The value will be moved code unit by code unit from left to right. The result may be truncated or padded with trailing blank characters, depending on the length defined and the number of bytes specified. The length of the receiving binary field must be even.
13. **Alphanumeric to Unicode:**
    The alphanumeric value will be converted from the default code page to a Unicode value using the International Components for Unicode (ICU) library. The result may be truncated or padded with trailing blank characters, depending on the length defined and the number of code units specified.
14. **(N,P,I) and binary (length 1-4) to Unicode:**
    The value will be converted to unpacked form from which an alphanumeric value will be obtained by suppression of leading zeros. For negative numeric values, the sign will be converted to the hexadecimal notation `Dx`. Any decimal point in the numeric value will be ignored. All digits before and after the decimal point will be treated as one integer value. The resulting value will be converted from alphanumeric to Unicode. The result may be truncated or padded with trailing blank characters, depending on the length defined and the number of code units specified.
15. **Binary (>4 bytes) to Unicode:**
    The value will be moved byte by byte from left to right. The result may be truncated or padded with trailing blanks, depending on the length defined and the number of bytes specified. The length of the sending binary field must be even.

16. **L,D,T,F to U:**
    The values are converted to an alphanumeric display form. The resulting value will be converted from alphanumeric to Unicode and assigned left justified.

If source and target format are identical, the result may be truncated or padded with trailing blank characters (format A and U) or leading binary zeros (format B) depending on the length defined and the number of bytes (format A and B) or code units (format U) specified.

See also *Using Dynamic Variables*.

Field Truncation and Field Rounding

The following rules apply to field truncation and rounding:

- High-order numeric field truncation is allowed only when the digits to be truncated are leading zeros. Digits following an expressed or implied decimal point may be truncated.
- Trailing positions of an alphanumeric field may be truncated.
- If the option `ROUNDED` is specified, the last position of the result will be rounded up if the first truncated decimal position of the value being assigned contains a value greater than or equal to 5. For the result precision of a division, see also *Precision of Results of Arithmetic Operations*.

Result Format and Length in Arithmetic Operations

The following table shows the format and length of the result of an arithmetic operation:

	I1	I2	I4	N or 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 or P	P*	P*	P*	P*	F4	F8
F4	F4	F4	F4	F4	F4	F8
F8	F8	F8	F8	F8	F8	F8

On a mainframe computer, format/length F8 is used instead of F4 for improved precision of the results of an arithmetic operation.

P* is determined from the integer length and precision of the operands individually for each operation, as shown under *Precision of Results of Arithmetic Operations*.

The following decimal integer lengths and possible values are applicable for format I:

Format/Length	Decimal Integer Length	Possible Values
I1	3	-128 to 127
I2	5	-32768 to 32767
I4	10	-2147483648 to 2147483647

Arithmetic Operations with Floating-Point Numbers

The following topics are covered below:

- General Considerations
- Precision of Floating-Point Numbers
- Conversion to Floating-Point Representation
- Platform Dependency

General Considerations

Floating-point numbers (format F) are represented as a sum of powers of two (as are integer numbers (format I)), whereas unpacked and packed numbers (formats N and P) are represented as a sum of powers of ten.

In unpacked or packed numbers, the position of the decimal point is fixed. In floating-point numbers, however, the position of the decimal point (as the name indicates) is "floating", that is, its position is not fixed, but depends on the actual value.

Floating-point numbers are essential for the computing of trigonometric functions or mathematical functions such as sinus or logarithm.

Precision of Floating-Point Numbers

Due to the nature of floating-point numbers, their precision is limited:

- For a variable of format/length F4, the precision is limited to approximately 7 digits.
- For a variable of format/length F8, the precision is limited to approximately 15 digits.

Values which have more significant digits cannot be represented exactly as a floating-point number. No matter how many additional digits there are before or after the decimal point, a floating-point number can cover only the leading 7 or 15 digits respectively.

An integer value can only be represented exactly in a variable of format/length F4 if its absolute value does not exceed $2^{23}-1$.

Conversion to Floating-Point Representation

When an alphanumeric, unpacked numeric or packed numeric value is converted to floating-point format (for example, in an assignment operation), the representation has to be changed, that is, a sum of powers of ten has to be converted to a sum of powers of two.

Consequently, only numbers that are representable as a finite sum of powers of two can be represented exactly; all other numbers can only be represented approximately.

Examples:

This number has an exact floating-point representation:

$$1.25 = 2^{0} + 2^{-2}$$

This number is a periodic floating-point number without an exact representation:

$$1.2 = 2^{0} + 2^{-3} + 2^{-4} + 2^{-7} + 2^{-8} + 2^{-11} + 2^{-12} + \ldots$$

Thus, the conversion of alphanumeric, unpacked numeric or packed numeric values to floating-point values, and vice versa, can introduce small errors.

Platform Dependency

Because of different hardware architecture, the representation of floating-point numbers varies according to platforms. This explains why the same application, when run on different platforms, may return slightly different results when floating-point arithmetics are involved. The respective representation also determines the range of possible values for floating-point variables, which is (approximately)

- $\pm 1.17 * 10^{-38}$ to $\pm 3.40 * 10^{38}$ for F4 variables,
- $\pm 2.22 * 10^{-308}$ to $\pm 1.79 * 10^{308}$ for F8 variables.

Note: The representation used by your pocket calculator may also be different from the one used by your computer - which explains why results for the same computation may differ.

Arithmetic Operations with Date and Time

With formats D (date) and T (time), only addition, subtraction, multiplication and division are allowed. Multiplication and division are allowed on intermediate results of additions and subtractions only.

Date/time values can be added to/subtracted from one another; or integer values (no decimal digits) can be added to/subtracted from date/time values. Such integer values can be contained in fields of formats N, P, I, D, or T.

The intermediate results of such an addition or subtraction may be used as a multiplicand or dividend in a subsequent operation.

An integer value added to/subtracted from a date value is assumed to be in days. An integer value added to/subtracted from a time value is assumed to be in tenths of seconds.

For arithmetic operations with date and time, certain restrictions apply, which are due to the Natural's internal handling of arithmetic operations with date and time, as explained below.

Internally, Natural handles an arithmetic operation with date/time variables as follows:

```
COMPUTE result-field = operand1 +/- operand2
```

The above statement is resolved as:

1. *intermediate-result = operand1 +/- operand2*
2. *result-field = intermediate-result*

That is, in a first step Natural computes the result of the addition/subtraction, and in a second step assigns this result to the result field.

More complex arithmetic operations are resolved following the same pattern:

```
COMPUTE result-field = operand1 +/- operand2 +/- operand3 +/- operand4
```

The above statement is resolved as:

1. *intermediate-result1 = operand1 +/- operand2*
2. *intermediate-result2 = intermediate-result1 +/- operand3*
3. *intermediate-result3 = intermediate-result2 +/- operand4*
4. *result-field = intermediate-result3*

The resolution of multiplication and division operations is similar to the resolution for addition and subtraction.

The internal format of such an intermediate result depends on the formats of the operands, as shown in the tables below.

Addition

The following table shows the format of the intermediate result of an addition (*intermediate-result = operand1 + operand2*):

Format of *operand1*	Format of *operand2*	Format of *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
Ti, N, P, I	Ti	Ti
Di	Ti, N, P, I	Di
Ti	Di, N, P, I	Ti

Subtraction

The following table shows the format of the intermediate result of a subtraction (*intermediate-result = operand1 - operand2*):

Format of *operand1*	Format of *operand2*	Format of *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

Multiplication or Division

The following table shows the format of the intermediate result of a multiplication (*intermediate-result = operand1 * operand2*) or division (*intermediate-result = operand1 / operand2*):

Format of *operand1*	Format of *operand2*	Format of *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

Internal Assignments

Di is a value in internal date format; Ti is a value in internal time format; such values can be used in further arithmetic date/time operations, but they cannot be assigned to a result field of format D (see the assignment table below).

In complex arithmetic operations in which an intermediate result of internal format Di or Ti is used as operand in a further addition/subtraction/multiplication/division, its format is assumed to be D or T respectively.

The following table shows which intermediate results can internally be assigned to which result fields (*result-field = intermediate-result*).

Format of *result-field*	Format of *intermediate-result*	Assignment possible
D	D, T	yes
D	Di, Ti, N, P, I	no
T	D, T, Di, Ti, N, P, I	yes
N, P, I	D, T, Di, Ti, N, P, I	yes

A result field of format D or T must not contain a negative value.

Examples 1 and 2 (invalid):

```
COMPUTE DATE1 (D) = DATE2 (D) + DATE3 (D)
COMPUTE DATE1 (D) = DATE2 (D) - DATE3 (D)
```

These operations are not possible, because the intermediate result of the addition/subtraction would be format Di, and a value of format D*i* cannot be assigned to a result field of format D.

Examples 3 and 4 (invalid):

```
COMPUTE DATE1 (D) = TIME2 (T) - TIME3 (T)
COMPUTE DATE1 (D) = DATE2 (D) - TIME3 (T)
```

These operations are not possible, because the intermediate result of the addition/subtraction would be format Ti, and a value of format Ti cannot be assigned to a result field of format D.

Example 5 (valid):

```
COMPUTE DATE1 (D) = DATE2 (D) - DATE3 (D) + TIME3 (T)
```

This operation is possible. First, `DATE3` is subtracted from `DATE2`, giving an intermediate result of format Di; then, this intermediate result is added to `TIME3`, giving an intermediate result of format T; finally, this second intermediate result is assigned to the result field `DATE1`.

Examples 6 and 7 (invalid):

```
COMPUTE DATE1 (D) = DATE2 (D) + DATE3 (D) * 2
COMPUTE TIME1 (T) = TIME2 (T) - TIME3 (T) / 3
```

These operations are not possible, because the attempted multiplication/division is performed with date/time fields and not with intermediate results.

Example 8 (valid):

```
COMPUTE DATE1 (D) = DATE2 (D) + (DATE3(D) - DATE4 (D)) * 2
```

This operation is possible. First, `DATE4` is subtracted from `DATE3` giving an intermediate result of format Di; then, this intermediate result is multiplied by two giving an intermediate result of format Di; this intermediate result is added to `DATE2` giving an intermediate result of format D; finally, this third intermediate result is assigned to the result field `DATE1`.

If a format T value is assigned to a format D field, you must ensure that the time value contains a valid date component.

Performance Considerations for Mixed Format Expressions

When doing arithmetic operations, the choice of field formats has considerable impact on performance:

For business arithmetic, only fields of format I (integer) should be used, if possible.

For scientific arithmetic, fields of format F (floating point) should be used, if possible.

In expressions where formats are mixed between numeric (N, P) and floating point (F), a conversion to floating point format is performed. This conversion results in considerable CPU load. Therefore it is recommended to avoid mixed format expressions in arithmetic operations.

Precision of Results of Arithmetic Operations

Operation	Digits Before Decimal Point	Digits After Decimal Point
Addition/Subtraction	Fi + 1 or Si + 1 (whichever is greater)	Fd or Sd (whichever is greater)
Multiplication	Fi + Si + 2	Fd + Sd (maximum 7)
Division	Fi + Sd	(see below)
Exponentiation	15 - Fd (See *Exception* below)	Fd
Square Root	Fi	Fd

- where:

F	First operand
S	Second operand
R	Result
i	Digits before decimal point
d	Digits after decimal point

Exception:

If the exponent has one or more digits after the decimal point, the exponentiation is internally carried out in floating point format and the result will also have floating point format. See *Arithmetic Operations with Floating-Point Numbers* for further information.

Digits after Decimal Point for Division Results

The precision of the result of a division depends whether a result field is available or not:

- If a result field is available, the precision is: Fd or Rd (whichever is greater) [*].
- If no result field is available, the precision is: Fd or Sd (whichever is greater) [*].

[*] If the `ROUNDED` option is used, the precision of the result is internally increased by one digit before the result is actually rounded.

A result field is available (or assumed to be available) in a `COMPUTE` and `DIVIDE` statement, and in a logical condition in which the division is placed after the comparison operator (for example: `IF #A = #B / #C THEN ...`).

A result field is not (or assumed to be not) available in a logical condition in which the division is placed before the comparison operator (for example: `IF #B / #C = #A THEN ...`).

Exception:

If both dividend and divisor are of integer format and at least one of them is a variable, the division result is always of integer format (regardless of the precision of the result field and of whether the `ROUNDED` option is used or not).

Precision of Results for Arithmetic Expressions

The precision of arithmetic expressions, for example: `#A / (#B * #C) + #D * (#E - #F + #G)`, is derived by evaluating the results of the arithmetic operations in their processing order. For further information on arithmetic expressions, see *`arithmetic-expression`* in the `COMPUTE` statement description.

Error Conditions in Arithmetic Operations

In an addition, subtraction, multiplication or division, an error occurs if the total number of digits (before and after the decimal point) of the result is greater than 31.

In an exponentiation, an error occurs in any of the following situations:

- if the base is of packed format and either the result has over 16 digits or any intermediate result has over 15 digits;
- if the base is of floating-point format and the result is greater than approximately $7 * 10^{75}$.

Processing of Arrays

Generally, the following rules apply:

- All scalar operations may be applied to array elements which consist of a single occurrence.
- If a variable is defined with a constant value (for example, `#FIELD (I2) CONSTANT <8>`), the value will be assigned to the variable at compilation, and the variable will be treated as a constant. This means that if such a variable is used in an array index, the dimension concerned has a *definite* number of occurrences.
- If an assignment/comparison operation involves two arrays with a different number of dimensions, the "missing" dimension in the array with fewer dimensions is assumed to be (1:1).

  Example: If `#ARRAY1 (1:2)` is assigned to `#ARRAY2 (1:2,1:2)`, `#ARRAY1` is assumed to be `#ARRAY1 (1:1,1:2)`.

The following topics are covered below:

- Definitions of Array Dimensions
- Assignment Operations with Arrays
- Comparison Operations with Arrays
- Arithmetic Operations with Arrays

Definitions of Array Dimensions

The first, second and third dimensions of an array are defined as follows:

Number of Dimensions	Properties
3	#a3(3rd dim., 2nd dim., 1st dim.)
2	#a2(2nd dim., 1st dim.)
1	#a1(1st dim.)

Assignment Operations with Arrays

If an array range is assigned to another array range, the assignment is performed element by element.

Example:

```
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
```

If a single occurrence is assigned to an array range, each element of the range is filled with the value of the single occurrence. (For a mathematical function, each element of the range is filled with the result of the function.)

Before an assignment operation is executed, the individual dimensions of the arrays involved are compared with one another to check if they meet one of the conditions listed below. The dimensions are compared independently of one another; that is, the 1st dimension of the one array is compared with the 1st dimension of the other array, the 2nd dimension of the one array is compared with the 2nd dimension of the other array, and the 3rd dimension of the one array is compared with the 3rd dimension of the other array.

The assignment of values from one array to another is only allowed under one of the following conditions:

- The number of occurrences is the same for both dimensions compared.
- The number of occurrences is indefinite for both dimensions compared.
- The dimension that is assigned to another dimension consists of a single occurrence.

Example - Array Assignments:

The following program shows which array assignment operations are possible.

```
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
```

Comparison Operations with Arrays

Generally, the following applies: if arrays with multiple dimensions are compared, the individual dimensions are handled independently of one another; that is, the 1st dimension of the one array is compared with the 1st dimension of the other array, the 2nd dimension of the one array is compared with the 2nd dimension of the other array, and the 3rd dimension of the one array is compared with the 3rd dimension of the other array.

The comparison of two array dimensions is only allowed under one of the following conditions:

- The array dimensions compared with one another have the same number of occurrences.

- The array dimensions compared with one another have an indefinite number of occurrences.
- All array dimensions of one of the arrays involved are single occurrences.

Example - Array Comparisons:

The following program shows which array comparison operations are possible:

```
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
```

When you compare two array ranges, note that the following two expressions lead to different results:

```
#ARRAY1(*) NOT EQUAL #ARRAY2(*)
NOT #ARRAY1(*) = #ARRAY2(*)
```

Example:

- **Condition A:**

  ```
  IF #ARRAY1(1:2) NOT EQUAL #ARRAY2(1:2)
  ```

  This is equivalent to:

  ```
  IF (#ARRAY1(1) NOT EQUAL #ARRAY2(1)) AND (#ARRAY1(2) NOT EQUAL #ARRAY2(2))
  ```

  Condition A is therefore true if the first occurrence of #ARRAY1 does not equal the first occurrence of #ARRAY2 *and* the second occurrence of #ARRAY1 does not equal the second occurrence of #ARRAY2.

- **Condition B:**

  ```
  IF NOT #ARRAY1(1:2) = #ARRAY2(1:2)
  ```

  This is equivalent to:

  ```
  IF NOT (#ARRAY1(1)= #ARRAY2(1) AND #ARRAY1(2) = #ARRAY2(2))
  ```

  This in turn is equivalent to:

  ```
  IF (#ARRAY1(1) NOT EQUAL #ARRAY2(1)) OR (#ARRAY1(2) NOT EQUAL #ARRAY2(2))
  ```

  Condition B is therefore true if *either* the first occurrence of #ARRAY1 does not equal the first occurrence of #ARRAY2 *or* the second occurrence of #ARRAY1 does not equal the second occurrence of #ARRAY2.

Arithmetic Operations with Arrays

A general rule about arithmetic operations with arrays is that the number of occurrences of the corresponding dimensions must be equal.

The following illustrates this rule:

```
#c(2:3,2:4) := #a(3:4,1:3) + #b(3:5)
```

In other words:

Array	Dimension Number	Number of Occurrences	Range
#c	2nd	2	2:3
#c	1st	3	2:4
#a	2nd	2	3:4
#a	1st	3	1:3
#b	1st	3	3:5

The operation is performed element by element.

Note: An arithmetic operation of a different number of dimensions is allowed.

For the example above, the following operations are executed:

```
#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)
```

Below is a list of examples of how array ranges may be used in the following ways in arithmetic operations (in `COMPUTE`, `ADD` or `MULTIPLY` statements). In examples 1-4, the number of occurrences of the corresponding dimensions must be equal.

1. *range* + *range* = *range*.

   The addition is performed element by element.

2. *range* * *range* = *range*.

   The multiplication is performed element by element.

3. *scalar* + *range* = *range*.

   The scalar is added to each element of the range.

4. *range* * *scalar* = *range*.

   Each element of the range is multiplied by the scalar.

5. *range* + *scalar* = *scalar*.

   Each element of the range is added to the scalar and the result is assigned to the scalar.

6. *scalar* * *range* = *scalar2*.

The scalar is multiplied by each element of the array and the result is assigned to *scalar2*.

Since intermediate results will be generated for arithmetic operations as shown in the above examples, the result of overlapping index ranges is computed element by element in an intermediate result array and finally the intermediate result array is assigned to the result field.

Example:

```
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
```

51 Invoking Natural Subprograms from 3GL Programs

- Passing Parameters from the 3GL Program to the Subprogram 520
- Example of Invoking a Natural Subprogram from a 3GL Program 521

Natural subprograms can be invoked from a programming object written in a 3rd generation programming language (3GL). The invoking program can be written in any programming language that supports a standard `CALL` interface.

For this purpose, Natural provides the interface `ncxr_callnat`. The 3GL program invokes this interface with a specification of the name of the desired subprogram.

Note: Natural must have been activated beforehand; that is, the invoking 3GL program must in turn have been invoked by a Natural object with a `CALL` statement.

The subprogram is executed as if it had been invoked from another Natural object with a `CALLNAT` statement.

When the processing of the subprogram stops (either with the `END` statement or with an `ESCAPE ROUTINE` statement), control is returned to the 3GL program.

Passing Parameters from the 3GL Program to the Subprogram

Parameters can be passed from the invoking 3GL program to the Natural subprogram. For passing parameters, the same rules apply as for passing parameters with a `CALL` statement.

The 3GL program invokes the Natural interface `ncxr_callnat` with four parameters:

- The 1st parameter is the name of the Natural subprogram to be invoked.
- The 2nd parameter contains the number of parameters to be passed to the subprogram.
- The 3rd parameter contains the address of the table that contains the addresses of the parameters to be passed to the subprogram.
- The 4th parameter contains the address of the table that contains the format/length specifications of the parameters to be passed to the subprogram.

The sequence, format and length of the parameters in the invoking program must match exactly the sequence, format and length of the fields in the `DEFINE DATA PARAMETER` statement of the subprogram. The names of the fields in the invoking program and the invoked subprogram can be different.

Example of Invoking a Natural Subprogram from a 3GL Program

For an example of how to invoke a Natural subprogram from a 3GL program, refer to the following samples in the Natural root directory's subdirectory *samples\sysexuex*.

- `MY3GL.NSP` (for the main program),
- `MY3GLSUB.NSN` (for the subprogram),
- `MYC3GL.C` (for the `C` function).

52 Issuing Operating System Commands from within a Natural Program

- Syntax 524
- Parameters 524
- Parameter Options 524
- Return Codes 525
- Examples 525

The Natural user exit SHCMD can be used to issue an operating system command from within a Natural program.

Syntax

```
CALL 'SHCMD' 'command' ['option']
```

Parameters

command	*command* is to be executed by the operating system. To execute commands (such as DIR for directory or DEL for delete) you have to specify the system command interpreter as well. For more information, see ***Examples*** below.
option	*option* describes how the command should be executed. This parameter is optional. The following options are available: ■ ASYNCH ■ NOSCREENIO ■ SYNCH (default) See *Parameter Options* below.

Parameter Options

The following parameter options are available:

Option	Description
ASYNCH	Natural does not wait until the command is completely executed. This kind of processing is named asynchronous processing.
NOSCREENIO	This option is used to hide the output generated by the command. The hidden output is redirected to the null device.
SYNCH	Natural waits until the command is completed. This kind of processing is named synchronous processing and is set by default.

Note: The options ASYNCH and SYNCH may be not set at the same time.

Return Codes

The following return code values are available:

Return Code	Description
0	Command successfully executed.
4	Illegal SHCMD parameter specified.
All other codes	Operating-system-dependent error code.

Examples

Execute operating system command DIR to view a directory:

```
CALL 'SHCMD''CMD.EXE /C DIR'
```

Retrieve the return code by using the Natural function RET:

```
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
```

Execute a command which includes blanks within the command by enclosing the command with quotation marks.

The following example executes Microsoft Excel.

```
RESET #cmd (A253)
MOVE '"C:\Program Files\Microsoft Office\Office\EXCEL.EXE"' to #cmd
CALL "SHCMD" #cmd "ASYNCH"
```

In this case, parameter TQ (translate quotation marks) has to be OFF, otherwise the quotation marks have been removed.

To be independent of the TQ parameter, use the hexadecimal ASCII code of quotation marks (H'22') and append it at the beginning and the end of the command. The following example demonstrates this:

```
RESET #cmd (A253)
MOVE H'22' - "C:\Program Files\Microsoft Office\Office\EXCEL.EXE" - H'22' to #cmd
CALL "SHCMD" #cmd "ASYNCH"
```

53 Statements for Internet and XML Access

- Statements Available 528
- Further References 529

This chapter gives an overview of the Natural statements for internet and XML access and contains a list of further references. To take full advantage of these statements, a thorough knowledge of the underlying communication standards is required.

Statements Available

The following statements are available for internet and XML access:

- REQUEST DOCUMENT
- PARSE XML

REQUEST DOCUMENT

This statement enables you to use the HTTP/HTTPS protocol.

The following is an example of how this statement can be used to access an externally-located document:

```
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 ↩
```

For further information, see `REQUEST DOCUMENT` in the *Statements* documentation.

PARSE XML

The `PARSE XML` statement allows you to parse XML documents from a Natural program.

For further information, see `PARSE XML` in the *Statements* documentation.

Further References

Below is a list of resources that you may find useful.

- Sample Programs
- Training Courses
- Useful Links

Sample Programs

In addition to the sample programs provided at the end of the description of each statement, some sample programs are included in the Natural library `SYSEXV`.

Training Courses

Software AG's Corporate University offers special training courses on this subject. See the Corporate University offerings on Empower at ***https://empower.softwareag.com/***.

Or, ask your local Software AG representative for the availability of special on-site training courses at your location.

Useful Links

Below is a collection of links that may be of interest.

- 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/***

VIII Portable Natural Generated Programs

54 Portable Natural Generated Programs

- Compatibility 534
- Endian Mode Considerations 534
- ENDIAN Parameter 535
- Transferring Natural Generated Programs 535
- Portable FILEDIR.SAG and Error Message Files 537

As of Natural Version 5, Natural generated programs (GPs) are portable across UNIX, OpenVMS and Windows platforms.

Compatibility

As of Natural Version 5, a source which was cataloged on any Natural-supported UNIX, OpenVMS and Windows platform is executable with all of these Open Systems platforms without recompilation. This feature simplifies the deployment of applications across Open Systems platforms.

Natural applications generated with Natural Version 4 or Natural Version 3 can be executed with Natural Version 5 or above without cataloging the applications again (upward compatibility). In this case, the portable GP functionality is not available. To make use of the portable GP and other improvements, cataloging with Natural Version 5 or above is required.

Command processor GPs are not portable. The portable GP feature is not available for mainframe platforms. This means that Natural GPs which are generated on mainframe computers are not executable on UNIX, OpenVMS and Windows platforms without recompilation of the application and vice versa.

Endian Mode Considerations

As of Natural Version 5, Natural acts as follows: Depending on which UNIX, OpenVMS or Windows platform it is running, Natural will consider the byte order in which multi-byte numbers are stored in the GP. The two byte order modes are called “Little Endian” and “Big Endian”.

- “Little Endian” means that the low-order byte of the number is stored in memory at the lowest address, and the high-order byte at the highest address (the little end comes first).
- “Big Endian” means that the high-order byte of the number is stored in memory at the lowest address, and the low-order byte at the highest address (the big end comes first).

The UNIX, OpenVMS and Windows platforms use both endian modes: Intel processors and AXP computers have “Little Endian” byte order, other processors such as HP-UX, Sun Solaris, or RS6000 use “Big Endian” mode.

Natural converts a portable GP automatically into the endian mode of the execution platform, if necessary. This endian conversion is not performed if the GP has been generated in the endian mode of the platform.

ENDIAN Parameter

In order to increase execution performance of portable GPs, the profile parameter ENDIAN has been introduced. ENDIAN determines the endian mode in which a GP is generated during compilation:

DEFAULT	The endian mode of the machine on which the GP is generated.
BIG	Big endian mode (high order byte first).
LITTLE	Little endian mode (low order byte first).

The values DEFAULT, BIG and LITTLE are alternatives whereby the default value is DEFAULT.

The ENDIAN mode parameter may be set

- as a profile parameter with the Natural Configuration Utility,
- as a start-up parameter,
- as a session parameter or with the GLOBALS command.

Transferring Natural Generated Programs

To make use of the portable GP on different platforms (UNIX, OpenVMS and Windows), the generated Natural objects must be transferred to the target platform or must be accessible from the target platform, for example, via NFS.

Using the Natural Object Handler is the recommended way to distribute Natural generated objects or even entire Natural applications. This is done by unloading the objects in the source environment into a work file, transferring the work file to the target environment and loading the objects from the work file.

To deploy your Natural generated objects across Open Systems platforms

1 Start the Natural Object Handler. Unload all necessary cataloged objects into a work file of type PORTABLE.

Error messages, if needed, can also be unloaded to the work file.

Important: The specified work file type must be of type PORTABLE. PORTABLE performs an automatic endian conversion of a work file when it is transferred to a different machine. See also the information on the work file type in the description of the DEFINE WORK FILE statement in the *Statements* documentation.

2 Transfer the work file to the target environment. Depending on the transfer mechanism (network, CD, diskette, tape, email, download, etc.), the use of a compressed archive such as a ZIP file or encoding with UUENCODE/UUDECODE or similar may make sense. Copying via FTP requires binary transfer type.

Note: According to the transfer method used, it may be necessary to adjust the record format and attributes or block size of the transferred work file depending on the specific target platform, before continuing with the load function. The work file should have the same format and attributes on the target platform as a work file of the same type that was generated on the target platform itself. Use operating system tools if an adaptation is necessary.

3 Start the Natural Object Handler in the target environment. Select `PORTABLE` as work file type. Load the Natural objects and error messages from the work file.

For more details on how to use the Natural Object Handler, refer to *Object Handler* in the *Utilities* documentation.

You can find more information about how to port an application from a Natural development workstation to a Natural Runtime workstation in the section *Porting Procedure Overview* in the *Operations* documentation.

Beside the aforementioned preferred method, there are various other ways of "moving" or copying single Natural generated objects or even entire libraries or parts thereof, using operating system tools and different transfer methods. In all of these cases, to make the objects executable by Natural, they have to be imported into the Natural system file `FUSER` so that the *FILEDIR.SAG* structure is adapted. For information on the `FNAT` or `FUSER` directory, see *System Files FNAT and FUSER* in the *Operations* documentation.

This can be done with either of the following methods:

- Using the Import function of the `SYSMAIN` utility.
- Using the `FTOUCH` utility. This utility can be used without entering Natural.
- It is also possible to import files from the Windows Explorer to the Natural environment, using drag-and-drop or the menu commands **Cut**, **Copy** and **Paste**. This means, if you have access to the Natural objects you want to import via the Windows Explorer, you can use drag-and-drop or **Cut**, **Copy** and **Paste**, and the *FILEDIR.SAG* file will be updated automatically. For more details on copying, moving and importing objects, see *Managing Natural Objects* in the *Using Natural Studio* documentation.

The same applies when direct access is possible from a target platform to the generated objects in the source environment, for example, via NSF, network file server, etc. In this case, the objects have to be imported, too.

Portable FILEDIR.SAG and Error Message Files

As of Natural Version 6.2, the file *FILEDIR.SAG* and the error message files are platform independent. Hence, it is possible to share common `FUSER` system files among different Open Systems platforms. For example, it is possible to copy sets of Natural libraries from one Open Systems platform to another with operating system copy procedures. However, it is not recommended to share `FNAT` system files. For more information about the portable *FILEDIR.SAG*, refer to *Portable Natural System Files* in the *Operations* documentation.

IX 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*.

55 What is an Event-Driven Application?

- Introduction 542
- Program-Driven Applications 543
- Event Driven Applications 544
- What is Happening Here? 545
- Writing Event-Driven Code 545
- Components of an Event Driven Application 546

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.

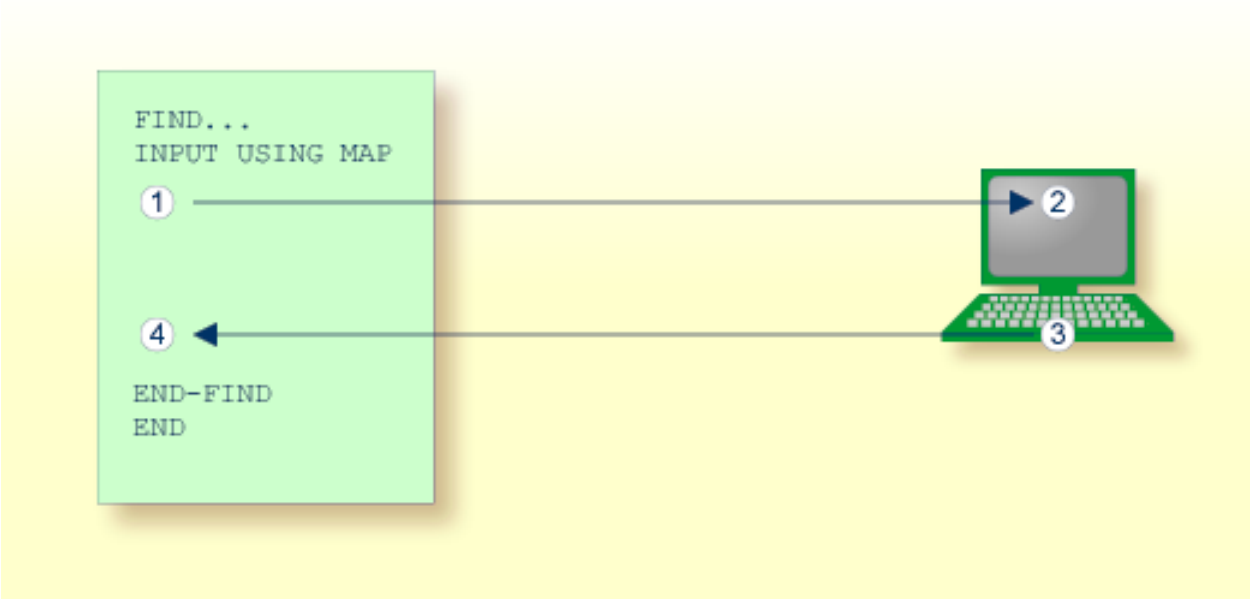
Because you cannot predict what the user will do, your code must make a few assumptions when it executes. For example, the application might assume that the user added text to an edit-area control before pressing the **OK** button.

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

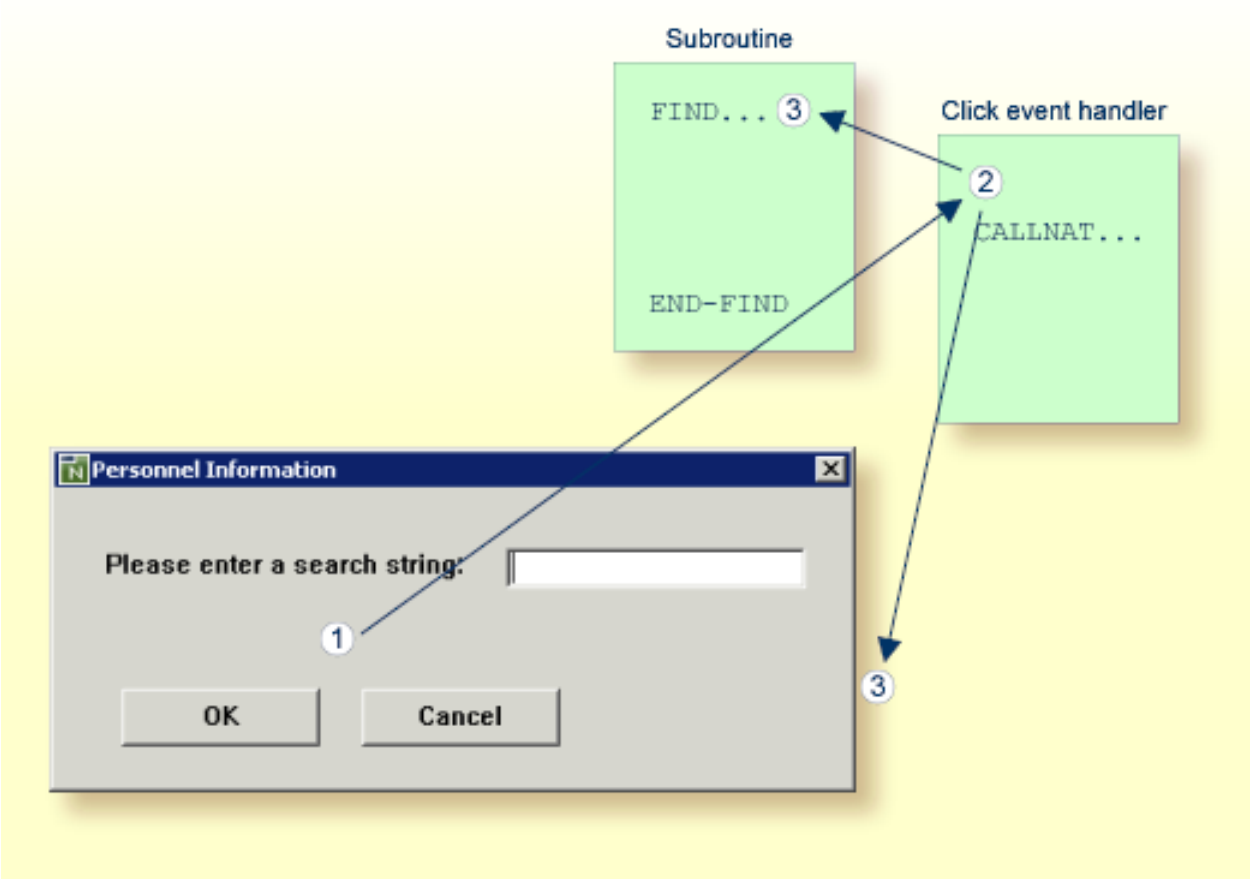


In typical program-driven applications, the following sequence of steps applies:

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.

Caution: 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.

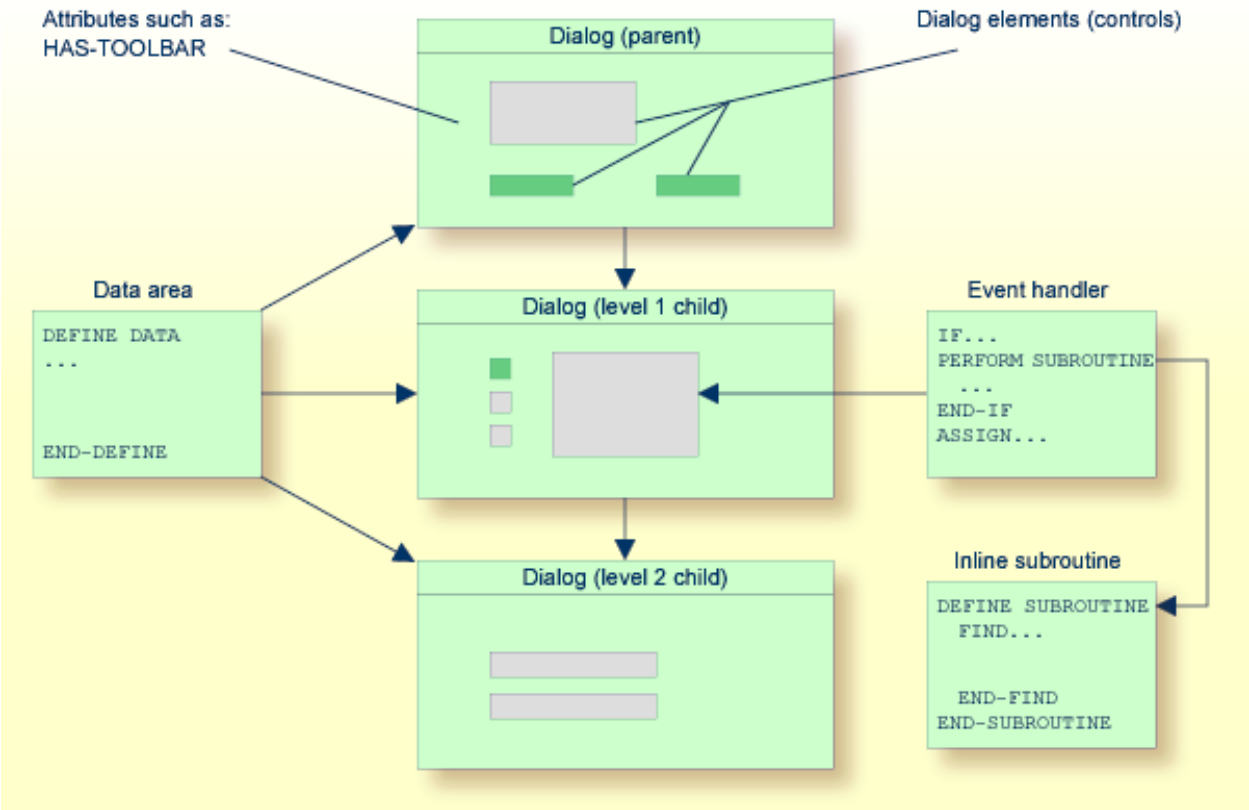
At runtime of the dialog, there is a difference between the runtime instance identified by the system variable `*DIALOG-ID` and the GUI instance (handle) of the dialog window (the default handle name is `#DLG$WINDOW`).

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.

56 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.

57 GUI Design Tips

- Introduction 552
- Do Your Research 552
- Screen Design 553
- Menu Design 554
- Color Usage 555
- Consistency Check 555

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.

58 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.

59 Tutorial

- Creating a Dialog 560
- Assigning Attributes to the Dialog 561
- Creating Dialog Elements Inside the Dialog 563
- Assigning Attributes to the Dialog Elements 565
- Creating the Application's Local Data Area 566
- Attaching Event Handler Code to the Dialog Element 567
- Checking, Stowing and Running the Application 568

This chapter 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:

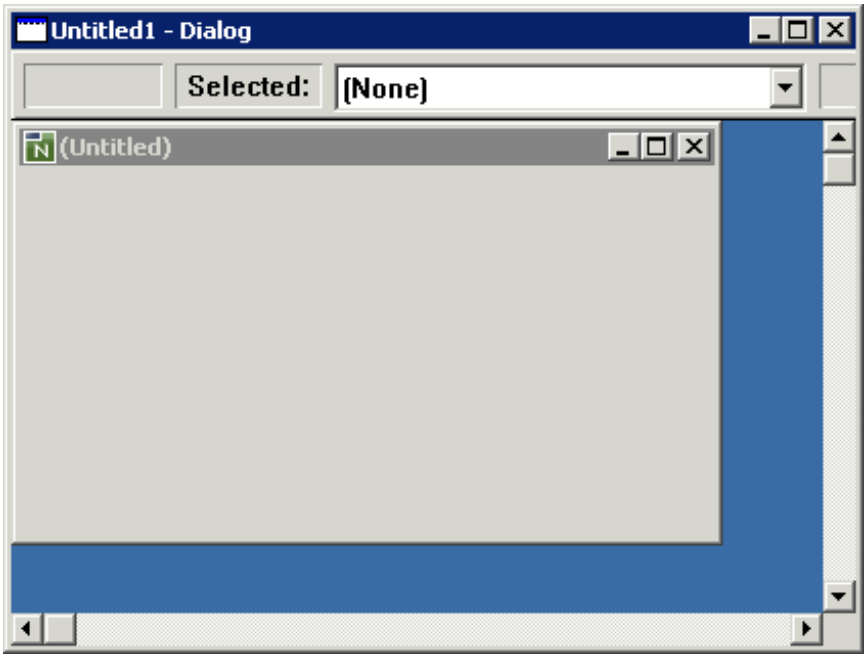
Creating a Dialog

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

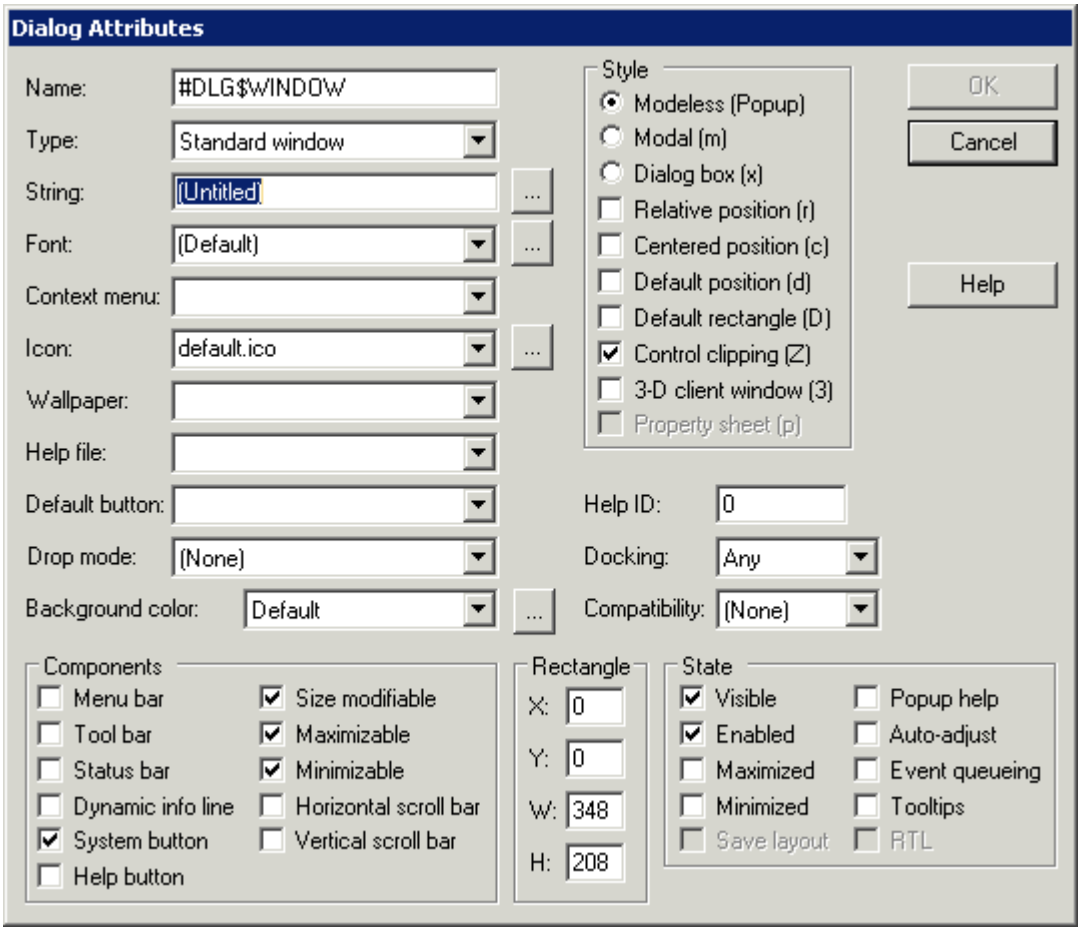
To assign attributes to the dialog

1 From the **Dialog** menu, choose **Attributes**.

Or:

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.

Note: 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**.

Or:

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**.

Or:

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**.

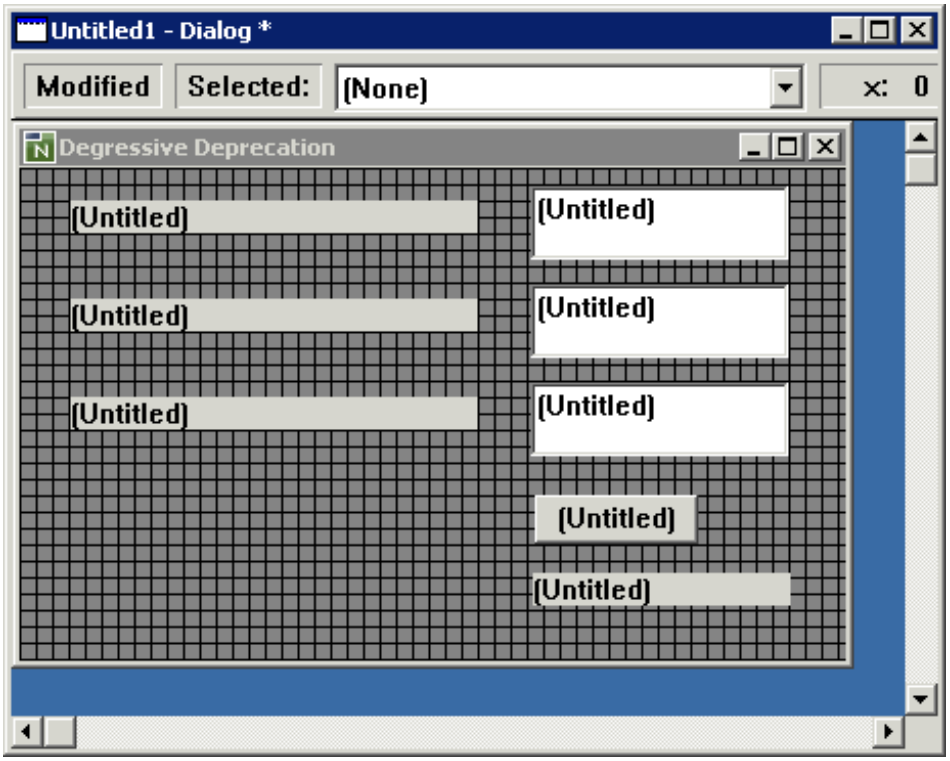
Or:

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**.

Or:

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.

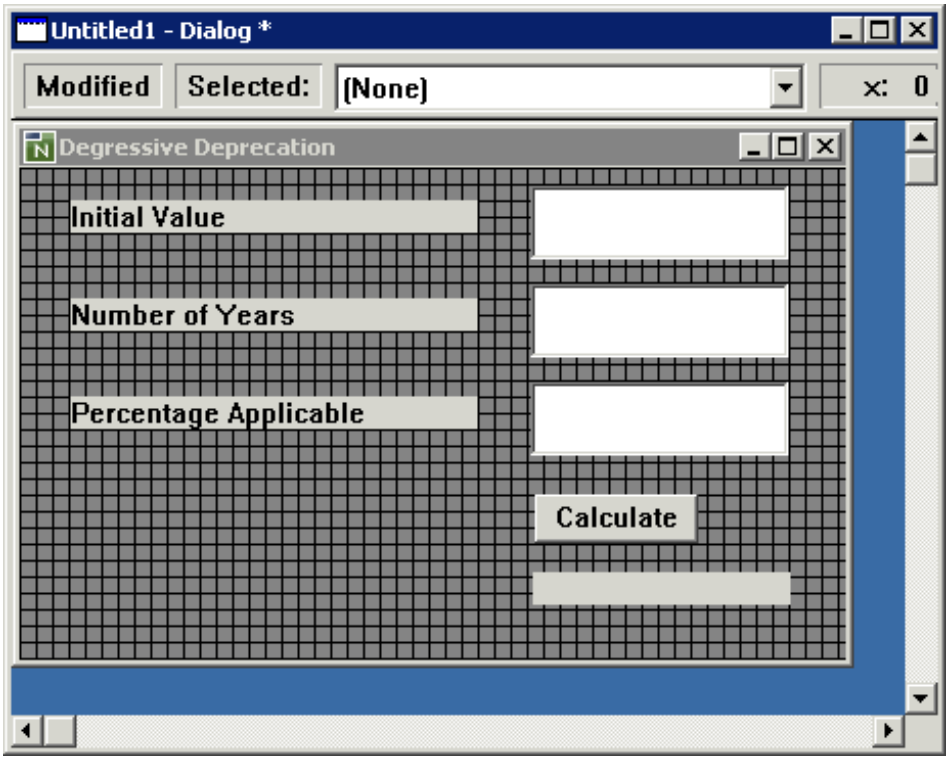
The attributes dialog box closes.

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**.

Or:

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**.

60 Basic Terminology

- Attribute 570
- Base Dialog 570
- Control 571
- Dialog 571
- Dialog Box 571
- Dialog Editor 571
- Dialog Element 571
- Event 572
- Event Handler 572
- Handle 572
- Item 572
- MDI - Multiple Document Interface 572
- MDI Child Window 573
- MDI Frame Window 573
- Modal Window 573
- SDI - Single Document Interface 573
- Popup 573
- Window 573

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.

X 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
Using the TERMINATE or STOP Statements within Dialog-based Applications
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

61 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.

Note: 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*.

62 How To Open and Close Dialogs

- Opening a Dialog 580
- Operands 580
- Passing Parameters to the Dialog 581
- Permanence in Creating, Passing and Checking Data 582
- Processing Steps When Opening a Dialog 583
- Closing Dialogs 584
- Initializing Attribute Values 584

This chapter covers the following topics:

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  ]
                      [ WITH      { operand4...           } ]
                                  { 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
```

Note: 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.

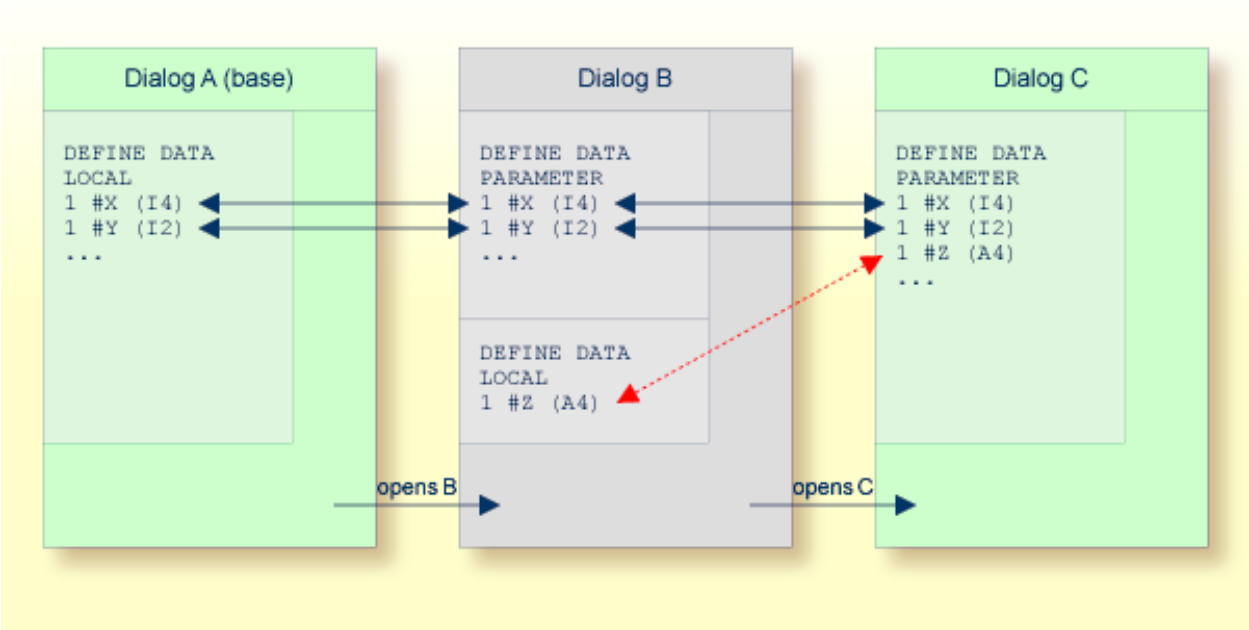
The reference to the permanent data is kept by saving the parameter data area internally during opening of the dialog. This reference is reused when

- 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.

Note: 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
```

63 How To Edit a Dialog's Enhanced Source Code

- What Is The Enhanced Source Code Format? 588
- Avoiding Incompatibilities Between Dialog Editor And Program Editor 589
- How To Use The Enhanced Source Code Format 590

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.

Note: 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, Natural for UNIX and Natural for OpenVMS, 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.

Note: 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**.

Or:

Choose the "Source Code" toolbar button.

Or:

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.

Note: If you are replacing strings with this command, this can make a dialog source incompatible with the dialog editor.

64 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).

65 How To Define Dialog Elements

- Introduction 594
- HANDLE OF GUI 595
- NULL-HANDLE 595

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.

Note: 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.

66 How To Manipulate Dialog Elements

- Introduction 598
- Querying, Setting and Modifying Attribute Values 598
- Querying and Modifying Unicode Attribute Values 599
- Restrictions 600
- Numeric/Alphanumeric Assignment 600

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.

Important: You must have **defined a handle**.

Note: 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.

Querying and Modifying Unicode Attribute Values

All alphanumeric attributes are stored in Unicode form internally and are thus locale-independent.

However, in order to allow attribute values to be used as Natural statement operands, it is necessary for the data to be transferred through a temporary internal data field that is accessible to the Natural Run-time. This internal field has the format and length as is documented for the corresponding attribute, which (for historical, technical and compatibility reasons) is format A for alphanumeric attributes by default and cannot be changed. The upshot of this is that alphanumeric attribute accesses will fail by default if the value being transferred contains characters that cannot be represented in the active code page. For example:

```
#PB-1.STRING := U'ошúбка'    /* --> NAT3413 ERROR in non-Cyrillic locales
```

The solution to this problem is to inform Natural that the temporary internal data field should be created with format U rather than with format A. This is done by appending a "-U" suffix to the name of any alphanumeric attribute. For example:

```
#PB-1.STRING-U := U'ошúбка'  /* works in all locales
```

It is important to note that only the use of the "-U" suffix only has an effect on the format of the temporary internal data field, not on the stored attribute. For example, there is only one `STRING` attribute, regardless of whether it is addressed via the STRING or STRING-U notation. The same storage location is used in both cases.

As a general rule, *if alphanumeric attribute values are being set or retrieved that contain at least one character that is not representable in the current code page, then the attribute name should be suffixed with a* "-U".

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*)

67 How To Create and Delete Dialog Elements Dynamically

- Introduction 604
- Global Attribute List 604
- Creating Dialog Elements Statically and Dynamically 604
- How to Handle Events of Dynamically Created Dialog Elements 606

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.

Note: 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.

Note: 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
```

68 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`.

69 Defining and Using Context Menus

- Introduction 612
- Construction 612
- Association 613
- Invocation 614
- Manual Invocation 617
- Sharing of Context Menus 618

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
```

70 Using the Clipboard and Drag and Drop

- Introduction 622
- Clipboard Specifics 624
- Drag and Drop Specifics 625
- Drag and Drop Insertion Marks 627
- Drag-Drop Checklist 628

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
```

71 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.

72 Generated Variables

- #DLG$PARENT 634
- #DLG$WINDOW 634

#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.

Note: 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.

Note: 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
...
```

73 Using the TERMINATE or STOP Statements within Dialog-based Applications

- Introduction 636
- Solution 636
- Example 636

Introduction

The execution of the TERMINATE and STOP statements relies on stack-based exception handling in order to immediately abort the application's execution, which is not guaranteed to work within event-driven (i.e., dialog-based) applications in consumer versions of Windows. In particular, this can be the case where one or more events on the Natural call stack have been synchronously raised in response to the corresponding system event(s). This is because the relevant section of the system call stack in such situations can contain Windows system code that has been compiled with a small performance optimization to not generate the required stack frame information at run-time.

Furthermore, because the Windows system code involved changes over time, a TERMINATE or STOP statement that works on one version of Windows is not guaranteed to continue to work on all later versions, or even on all later Service Packs for the same version.

For these reasons, the use of the TERMINATE and STOP statements in dialog-based applications is discouraged.

Solution

A universal solution is not available. However, in many cases, the problem can be circumvented by performing the termination asynchronously. For example, instead of coding a TERMINATE or STOP statement directly, one instead asynchronously invokes a user-defined event for the dialog via the CALL-DIALOG action, and places the TERMINATE or STOP statement within the event handler for the user-defined event, as illustrated in the example below.

In particular, this workaround is applicable in situations where no dialog boxes are active.

Example

Suppose that the CLOSE event for a modeless dialog contains a TERMINATE statement.

```
/* CLOSE event
TERMINATE
```

This TERMINATE statement fails in some Windows versions in the case where the user attempts to close the dialog via the Close (X) icon.

To solve this problem, we can firstly define a user-defined event for the dialog (e.g., with the name "TERMINATE"), and insert the TERMINATE statement there:

```
/* TERMINATE event
TERMINATE
```

Secondly, we replace the original TERMINATE statement in the CLOSE event with the following two statements:

```
/* CLOSE event
PROCESS GUI ACTION CALL-DIALOG WITH
  #DLG$WINDOW NULL-HANDLE 'TERMINATE' FALSE GIVING *ERROR
ESCAPE ROUTINE IMMEDIATE
```

The CALL-DIALOG action with the `FALSE` parameter causes the TERMINATE event to be executed asynchronously. The ESCAPE statement by-passes any subsequent event code and is absolutely necessary in this case to ensure that the DELETE-WINDOW action generated by the Dialog Editor is not performed, so that the dialog remains active until the TERMINATE event is received.

74 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.

Note: 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.

75 Triggering User-Defined Events

- Introduction 642
- Passing Parameters to the Dialog 643

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 a hash (#) 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
```

Note: 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.

76 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.

77 Menu Structures, Toolbars and the MDI

- Creating a Menu Structure 648
- Parent-Child Hierarchy in Menu Structures 650
- Creating a Toolbar 650
- Sharing Menu Structures, Toolbars and DILs (MDI Application) 651

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.

Note: 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 = *menubarhandlename* 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.

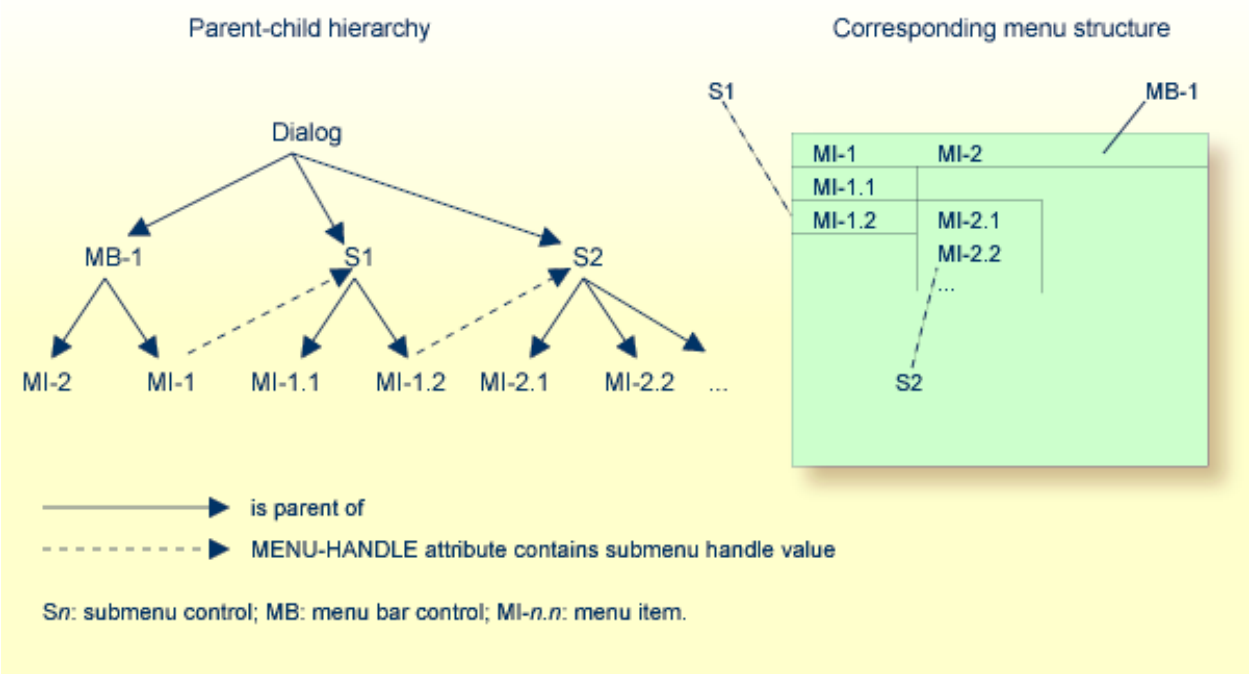
3 Create the controls and items in the sequence menu-bar control, submenu control with menu items.

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.

Note: 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.

Note: `#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.

Note: `#DLG$PARENT` is a generated variable.

78 Executing Standardized Procedures

- Introduction 654
- PROCESS GUI Statement 654

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.

79 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.

80 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.

81 Storing and Retrieving Client Data for a Dialog Element

- Introduction 660
- Integer Data 660
- Handle Data 661
- Keyed Alphanumeric Client Data 661
- Keyed Client Data in Native Format 664
- Key Enumeration 667

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
...
```

Note: 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

Note: 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
```

Note: 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.

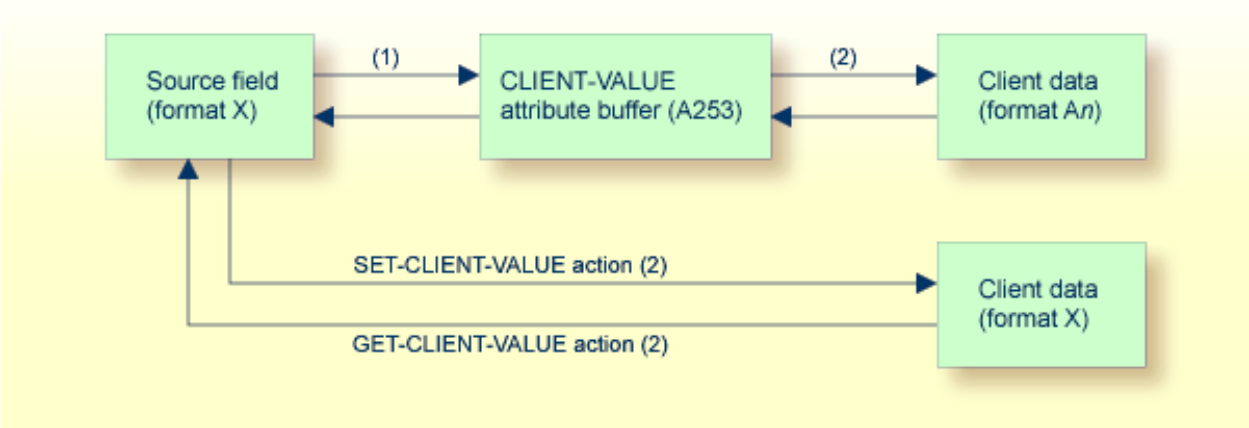
To update client data for a dialog element using the action-based technique

- 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.

Note: 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.

82 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.

Note: 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
```

83 Label Editing in Tree View and List View Controls

- Introduction 674
- Label Editing 674
- Changing an Item's Label Programmatically 676

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.

84 Working with ActiveX Controls

- Terminology 678
- How To Define an ActiveX Control 678
- How To Create an ActiveX Control 678
- Accessing Simple Properties 679
- Colors 680
- Pictures 681
- Fonts 681
- Variants 683
- Arrays 684
- Using the PROCESS GUI Statement 684

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

OLE Automation data type	NATURAL data type
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.

Note: 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>> WITHhandlename [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
```

85 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'
```

86 Working with Control Boxes

- Introduction 694
- Purpose of Exclusive Control Boxes 694
- Examples of Use of Exclusive Control Boxes 695
- Creation of the Wizard Pages 696

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

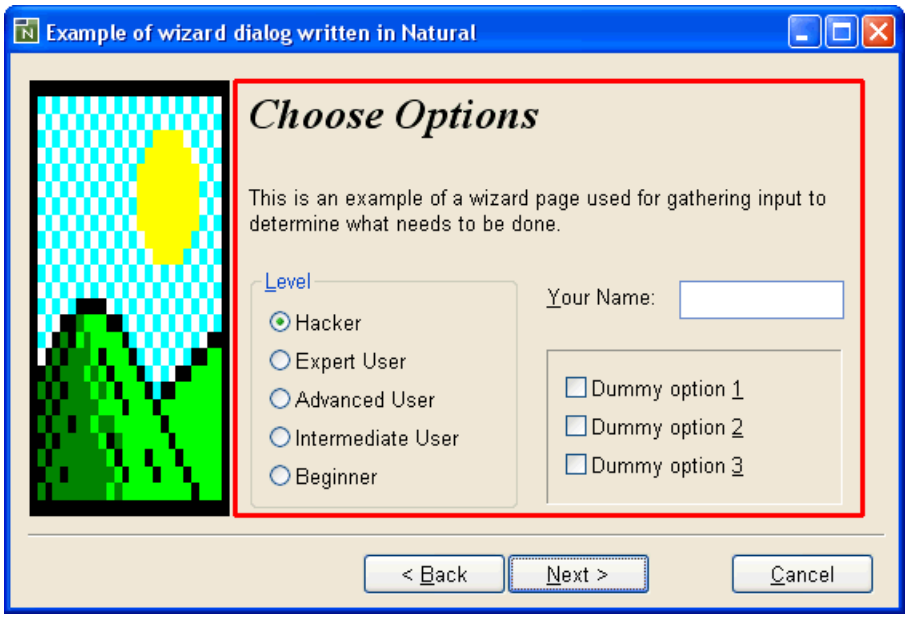
Exclusive control boxes, as described above, are primarily intended for situations where it is necessary to manage several overlapping “pages” of controls occupying the same region of a dialog. Without the auto-hiding feature which exclusive control boxes provide, it would be very difficult indeed for 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.

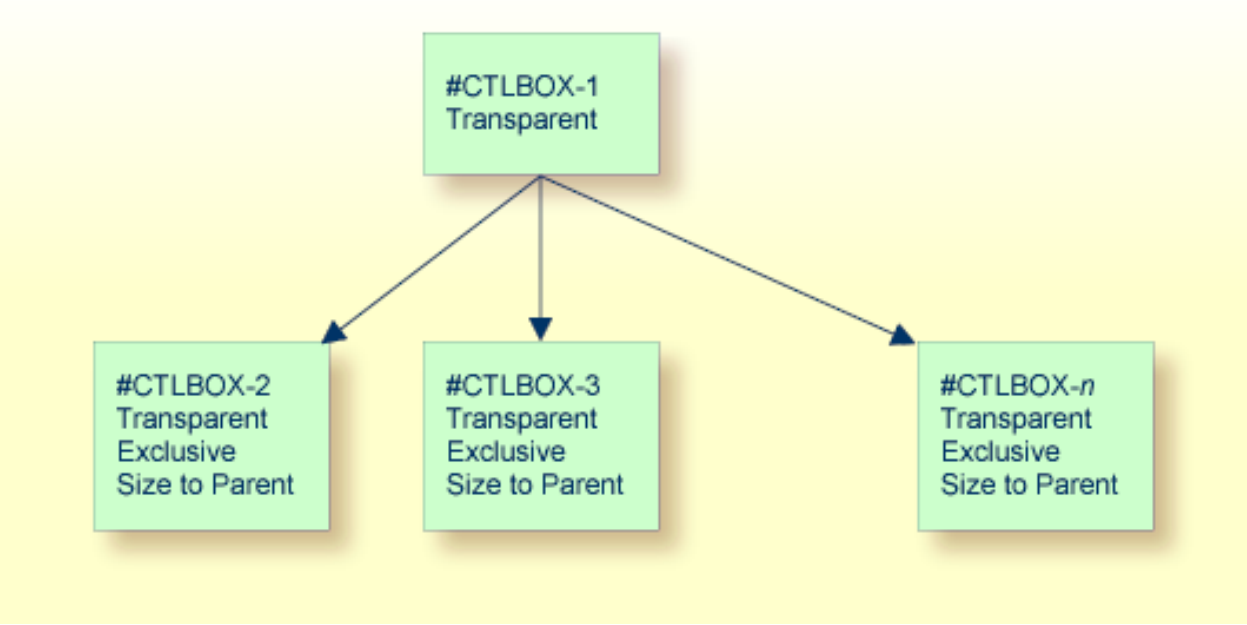
Examples of Use of Exclusive Control Boxes

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.

87 Working with Date and Time Picker (DTP) Controls

- Introduction 702
- Date and Time Formats 702
- Inputting Dates and Times 703
- Null Values 704
- Calendar Colors and Font 704

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.
hh	The two-digit hour in 12-hour format. Single-digit values are preceded by a zero.

Specifier	Description
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.

88 Working with Dialog Bar Controls

- Introduction 706
- Creating a Dialog Bar Control 706
- Types of Dialog Bar Control 706
- UI Transparency 709
- Client-Size Event 709
- Close Button 710
- Sample Code 710

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 run-time by using a `PROCESS GUI ACTION ADD` statement with the `TYPE` attribute set to `DIALOGBAR`.

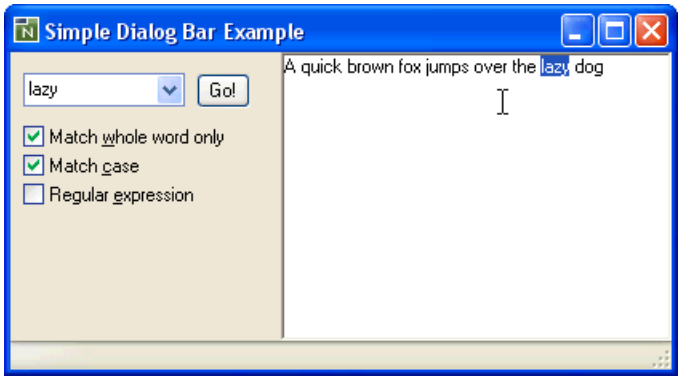
Types of Dialog Bar Control

A dialog bar control can exist in one of the following three basic forms (in order of complexity):

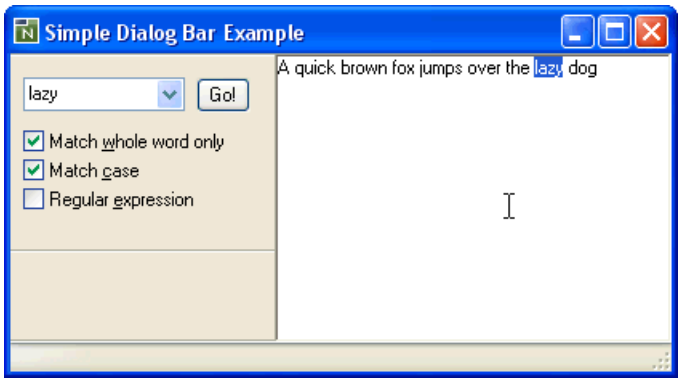
1. Neither dockable nor sizeable.
2. Dockable, but not sizeable.
3. Dockable and sizeable.

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:

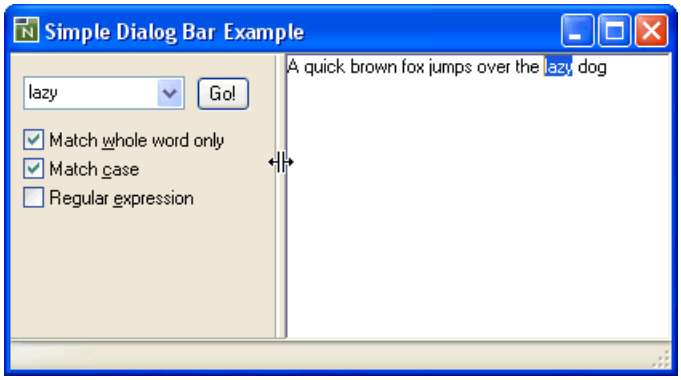


Setting the "dockable" state in the **Dialog Bar Attributes** window in the dialog editor causes the window to be draggable by the user. Note also that the dialog bar no longer extends to occupy the full length of the side to which it is docked (another dialog bar control or tool bar control could be docked underneath it):

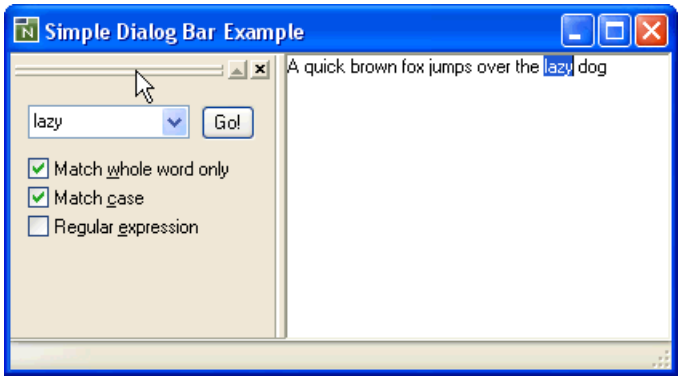


The user can drag the dialog bar control and re-dock it to another side of the owner dialog, or float it in its own separate window, as shown below:

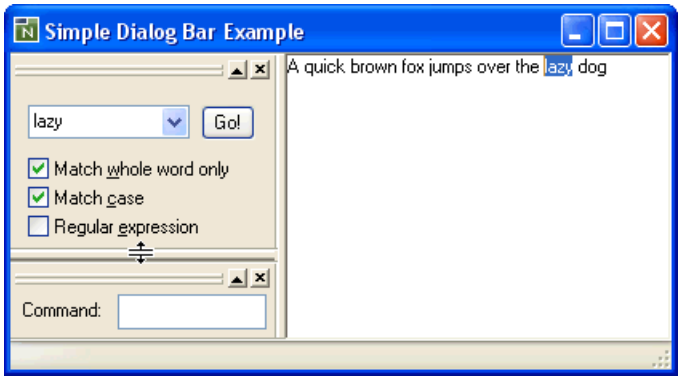
If the dialog bar control is made sizeable (by checking the "Dynamic (Y)" style flag in the **Attributes** window), a longitudinal splitter bar appears, allowing the dialog bar control to be resized. Note that sizeable dialog bar controls expand to fill the entire length of the side they are docked to 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:

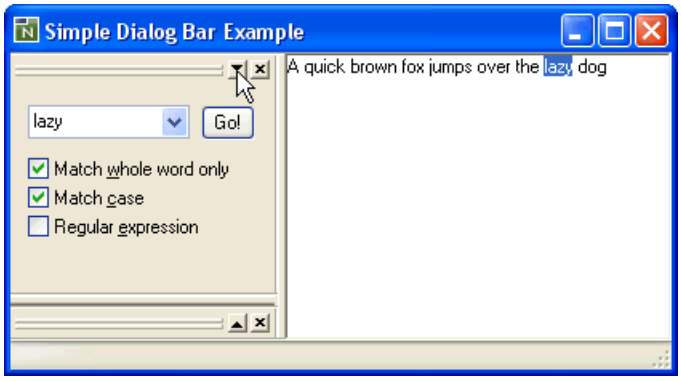


If a second sizeable dialog bar control is added, and docked alongside the first on the same row, a transverse splitter bar appears allowing the relative sizes of the two dialog bar controls to be changed. Note that the zoom button is now enabled:



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 ac-

tual 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
```

89 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
```

90 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**.

91 Working with Image List Controls

- Introduction 720
- Creating the Image List Control 720
- Adding Images 720
- Composite Images 721
- Scaling and Transparency 721
- Bitmaps vs. Icons 722
- Using an Image List 723
- Referencing Images from the Image List 723
- Overlay Images 724
- Modifying Images 725
- Deleting Images 726
- Deleting the Image List Control 726

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) back-

ground 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`.

92 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.

93 Working with List View Controls

- Introduction 734
- View Modes 734
- Setting Item Images 736
- Item Placement 736
- Item Selection 738
- Item Activation 739
- List View Columns and Sub-items 740
- Sorting 743
- Label Editing 745
- Multiple Context Menus 747
- Drag and Drop 748

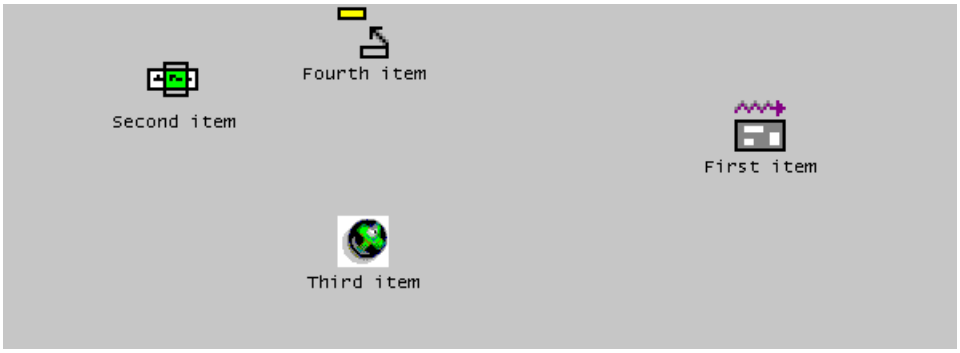
Introduction

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.

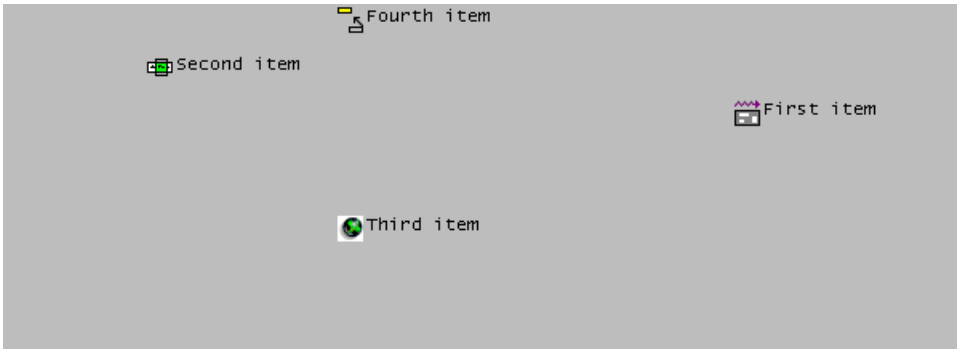
View Modes

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:

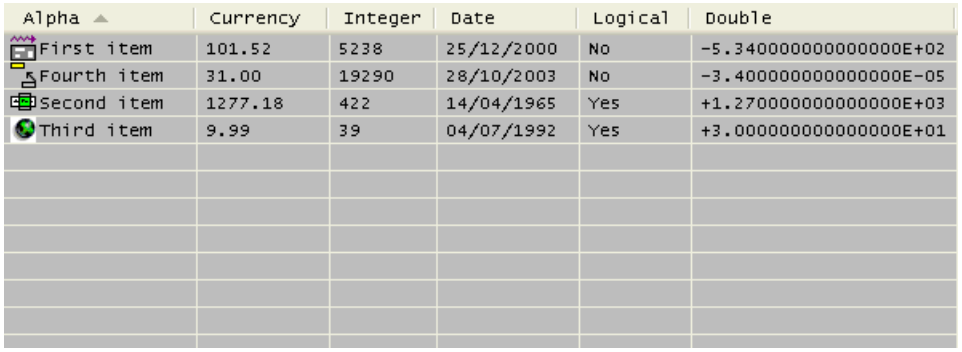


In small icon view mode, the item labels are displayed alongside the icons. As for the icon view, the items can optionally be displayed in arbitrary positions:



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. Iter-

ative 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 = '1'
   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.

94 Working with Nested Controls

- Introduction 754
- Which Control Types can be Containers? 755
- Creating a Nested Control 755
- Multiple Selection, Control Sequence and Clipboard Operations 756

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.

Note: 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.

95 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
...
```

Note: 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.

96 Working with Spin Controls

- Introduction 762
- Up-Down Control 762
- Buddy Control 762
- Date and Time Formats 763
- Inputting Dates and Times 764
- Null Values 765
- Calendar Colors and Font 765

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.

97 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'
...
```

Note: 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.

98 Working with Status Bar Controls

- Introduction 770
- Creating a Status Bar Control 770
- Using Status Bar Controls without Panes 770
- Outputting Text to a Status Bar Control 771
- Sharing a Status Bar in an MDI Application 772
- Pane-specific Context Menus 773

Introduction

Note: 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.

Note: 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'
```

Note: 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.

Note: 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
```

Note: 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.

99 Working with Tab Controls

- Creating a Tab Control 776
- Assigning Controls to Tabs 776
- Use of Control Boxes as Tab Control Pages 777
- Switching Between Controls Belonging To Different Tabs 778
- Mixing Tab-dependent and Tab-independent Controls 779
- Keyboard Navigation 779
- Tab Switching Events 780

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.

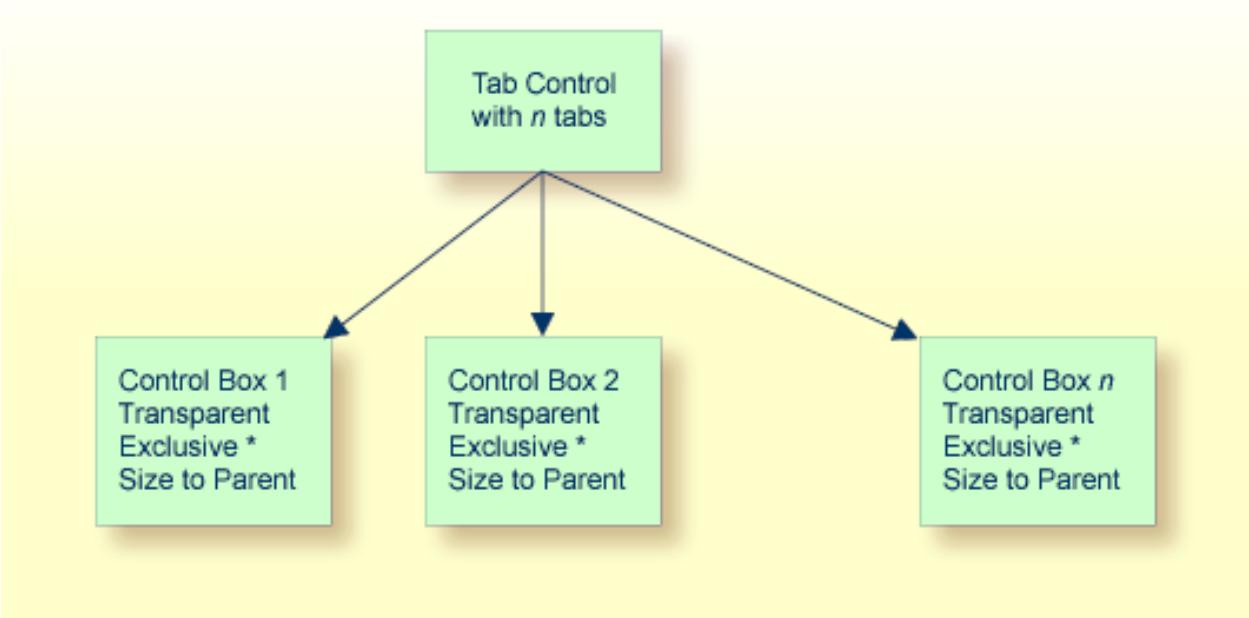
Note, however, that the dialog editor only automatically sets the child control's `OWNER` attribute if the tab control's "UI active (U)" `STYLE` flag is set, which is the default setting. Otherwise the child control's `OWNER` attribute is left unset (i.e., `NULL-HANDLE`). In the latter case, the child 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

Note: 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).

Note: 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.

Note: 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.

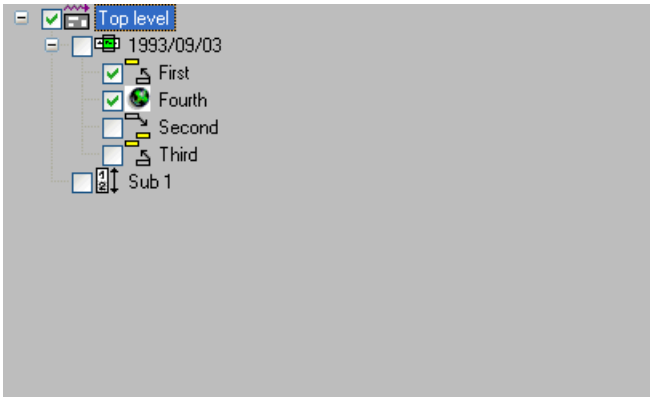
Note that no initial `ENTER` event is raised for the selected tab when the control is created, and that the tab control does not receive an initial `CHANGE` event either. Furthermore, when the dialog containing the tab control is closed, the currently-selected tab does not receive a `LEAVE` event.

100 Working with Tree View Controls

- Introduction 782
- Setting Item Images 782
- Item Selection 783
- Item Activation 783
- Item Data 784
- Sorting 784
- Label Editing 785
- Multiple Context Menus 786
- Dynamic Item Creation 786
- Drag and Drop 788

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.

101 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.

102 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.

103 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)
```

Note: 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.

104 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.

Note: 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`.

105 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.

Note: 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`.

Note: 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.

106 Defining Mnemonic and Accelerator Keys

- Introduction 804
- Defining a Mnemonic Key 804
- Defining an Accelerator Key 805
- Displaying Accelerator Keys in Menus 805

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".

Note: 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.

107 Dynamic Data Exchange - DDE

- Concepts 808
- Developing a DDE Server Application 809
- Developing a DDE Client Application 810
- Return Codes 811

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.

Note: 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.

Note: 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.

108 Object Linking and Embedding - OLE

- What is OLE in the Natural Context? 816
- OLE Documents Support 816
- Embedding and Linking 816
- Visual Editing - In-place Activation 817
- ActiveX Controls Support 818
- OLE Container Control 818
- Attributes, Events and PROCESS GUI Statement Actions 821

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`

XI Results Interface

109 Results Interface

- Purpose of the Results Interface 826
- Results Window Control Bar Access 826
- Tab Handling 827
- Image Handling 827
- Context Menu Handling 828
- Command Handling 828
- Column Handling 829
- Row Handling 829
- Data Handling 830
- Selection Handling 830

Purpose of the Results Interface

The Results Interface enables programmers to display data within the results window of Natural Studio. See also *Results Window* in the *Using Natural Studio* documentation.

Note: The results of the menu commands **Find Objects** and **Cat All** are not affected by the Results Interface.

The design and the usage of a tab in the results window can be determined via application programming interfaces (API). In general, a detailed view with columns and lines is used.

A context menu can be created for each entry, so that after the user-defined tab is shown it can be used for further processing.

This processing has to be defined within two programs:

1. In an update command handler before a context menu is shown.
2. In a command handler if an item is selected.

The application programming interfaces for the Results Interface are `USR5001N` - `USR5017N` and can be found in the library `SYSEXT`.

An example of the various functions is available in `USR5001P` with the update command handler in `USR5001A` and the command handler in `USR5001B`.

Notes:

1. Modifications of the pre-defined tabs (for example, the tabs with the labels **Find Objects** and **Cat All**) are not possible with this interface.
2. The results window and the Results Interface can be accessed only from Natural Studio.

Results Window Control Bar Access

The following application programming interface can be used to access the results window control bar.

Interface	Functionality
USR5001N	Turns results window on/off. Checks visibility of the results window.

Tab Handling

The application programming interfaces listed below can be used to define the general layout of a tab.

A tab can contain all or one of the following:

- Check box
- Full row selection
- Single row selection
- Images

A tab can be defined with the following attributes:

- Layout of the view (large/small icons, list or details view).
- Several usages (check boxes, images, grid lines, full or single row selection, view change).
- Layout of the tab label (text, bitmap or icon).

Interface	Functionality
USR5004N	Add, replace, delete and maintain layout of a tab.
USR5005N	Set and get active tab. Set tab active and set the focus on this tab

Image Handling

The following application programming interface can be used to specify bitmaps (**.bmp*) and icons (**.ico*) for a previously defined tab.

Interface	Functionality
USR5002N	Add and delete bitmaps and icons for a specified tab.

Context Menu Handling

The following application programming interfaces can be used to specify user-defined context menus.

Interface	Functionality
USR5003N	Add, remove and delete context menus of a tab.
USR5007N	Set and get checked/enabled state of context menu items.

The hierachy of the context menu must be defined manually.

The following array components can be defined:

Array Component	Value	Description
Type	1 to 4	1 - Context menu handling. 2 - Separator line. 3 - Begin of submenu. 4 - End of submenu.
Command ID	1 to 255	Free selectable number to identify a certain item in a context menu (used within the command handler).
Label	alphanumeric text	Text for the context menu items of type 1 and 3. A text for the status bar can be separated with H'0'A.
Image	Handle of image	Handle of a previously defined image (bitmap or icon). The image will be placed before the text of the context menu item.

Command Handling

A program can be assigned as an update command handler or as a command handler.

User-defined data can be saved/restored in the internal work area of the command handlers.

For example: handles of tabs.

The following application programming interfaces are available:

Interface	Functionality
USR5006N	Define update command handler and command handler.
USR5016N	Set and get data for the command handler work area.

Column Handling

The following application programming interfaces can be used to define the general layout of a column.

A column can contain all or one of the following:

- Title
- Width
- Data position
- Column sort

In addition, the default width and specified width of the column can be set up individually.

Interface	Functionality
USR5008N	Add, insert and delete columns of a tab.
USR5009N	Count number of columns.
USR5010N	Set and get default column width and width for specified columns.

Row Handling

The following application programming interfaces can be used to define the rows with images and context menus.

Interface	Functionality
USR5009N	Count number of rows.
USR5011N	Add, insert and delete rows of a tab.
USR5015N	A row can be scrolled into the visible area of the result window.

Data Handling

The following application programming interfaces can be used to write user-defined data into defined columns/rows.

If check boxes have been defined for a tab, they can be activated/deactivated for every row.

Interface	Functionality
USR5012N	Set and get data into a tab.
USR5013N	Set and get checked state of a row.

Selection Handling

The following application programming interfaces can be used to select rows individually.

Interface	Functionality
USR5014N	■ Set, reset and get selected rows. ■ Count amount of selected rows. ■ Set and reset row selection.
USR5015N	Set and get row of focus.
USR5017N	Copy selected rows to the Clipboard.

XII Designing Character-Based User Interfaces for Your Application

The user interface of an application, that is, the way an application presents itself to the user, is a key consideration when writing an application.

This part provides information on the various possibilities Natural offers for designing character-based user interfaces that are uniform in presentation and provide powerful mechanisms for user guidance and interaction.

Note: For information on the design of graphical user interfaces (GUI), refer to *Introduction to Event-Driven Programming*.

When designing user interfaces, standards and standardization are key factors.

Using Natural, you can offer the end user common functionality across various hardware and operating systems.

This includes the general screen layout (information, data and message areas), function-key assignment and the layout of windows.

This part covers the following topics:

Screen Design
Dialog Design

110 Screen Design

- Control of the Message Line - Terminal Command %M 834
- Assigning Colors to Fields - Terminal Command %= 834
- Infoline - Terminal Command %X 835
- Windows 836
- Standard/Dynamic Layout Maps 842
- Multilingual User Interfaces 842
- Skill-Sensitive User Interfaces 847

Control of the Message Line - Terminal Command %M

Various options of the terminal command `%M` are available for defining how and where the Natural message line is to be displayed.

Below is information on:

- Positioning the Message Line
- Message Line Color

Positioning the Message Line

%MB
The message line is displayed at the bottom of the screen.

%MT
The message line is displayed at the top of the screen.

Other options for the positioning of the message line are described in *%M - Control of Message Line* in the *Terminal Commands* documentation.

Message Line Color

%M=*color-code*
The message line is displayed in the specified color (for an explanation of color codes, see the session parameter `CD` as described in the *Parameter Reference*).

Assigning Colors to Fields - Terminal Command %=

You can use the terminal command `%=` to assign colors to field attributes for programs that were originally not written for color support. The command causes all fields/text defined with the specified attributes to be displayed in the specified color.

If predefined color assignments are not suitable for your terminal type, you can use this command to override the original assignments with new ones.

You can also use the `%=` terminal command within Natural editors, for example to define color assignments dynamically during map creation.

Codes	Description
blank	Clear color translate table.
F	Newly defined colors are to override colors assigned by the program.
N	Color attributes assigned by program are not to be modified.
O	Output field.
M	Modifiable field (output and input).
T	Text constant.
B	Blinking
C	Italic
D	Default
I	Intensified
U	Underlined
V	Reverse video
BG	Background
BL	Blue
GR	Green
NE	Neutral
PI	Pink
RE	Red
TU	Turquoise
YE	Yellow

Example:

```
%=TI=RE,OB=YE
```

This example assigns the color red to all intensified text fields and yellow to all blinking output fields.

Infoline - Terminal Command %X

The terminal command %X controls the display of the Natural infoline.

For further information, see the description of the terminal command `%X` in the *Terminal Commands* documentation.

Windows

Below is information on:

- What is a Window?
- DEFINE WINDOW Statement
- INPUT WINDOW Statement

What is a Window?

A *window* is that segment of a logical page, built by a program, which is displayed on the terminal screen.

A *logical page* is the output area for Natural; in other words the logical page contains the current report/map produced by the Natural program for display. This logical page may be larger than the physical screen.

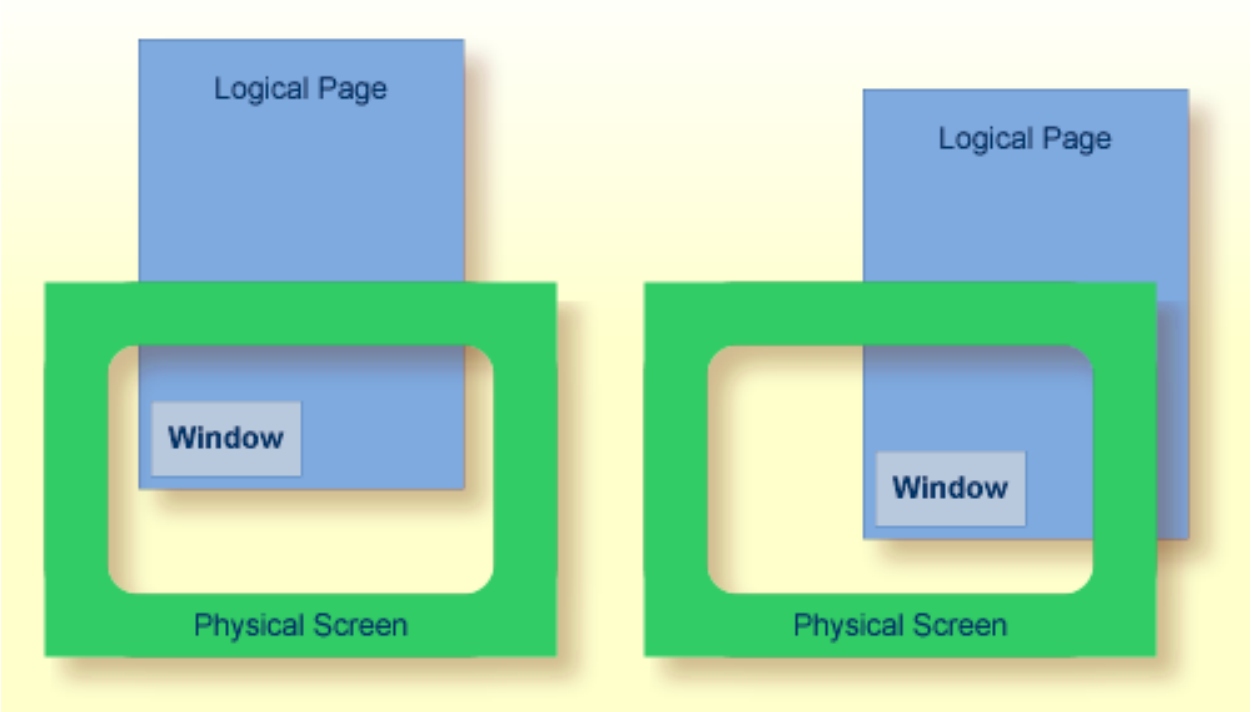
There is always a window present, although you may not be aware of its existence. Unless specified differently (by a `DEFINE WINDOW` statement), the size of the window is identical to the physical size of your terminal screen.

You can manipulate a window in two ways:

- You can control the size and position of the window on the *physical screen*.
- You can control the position of the window on the *logical page*.

Positioning on the Physical Screen

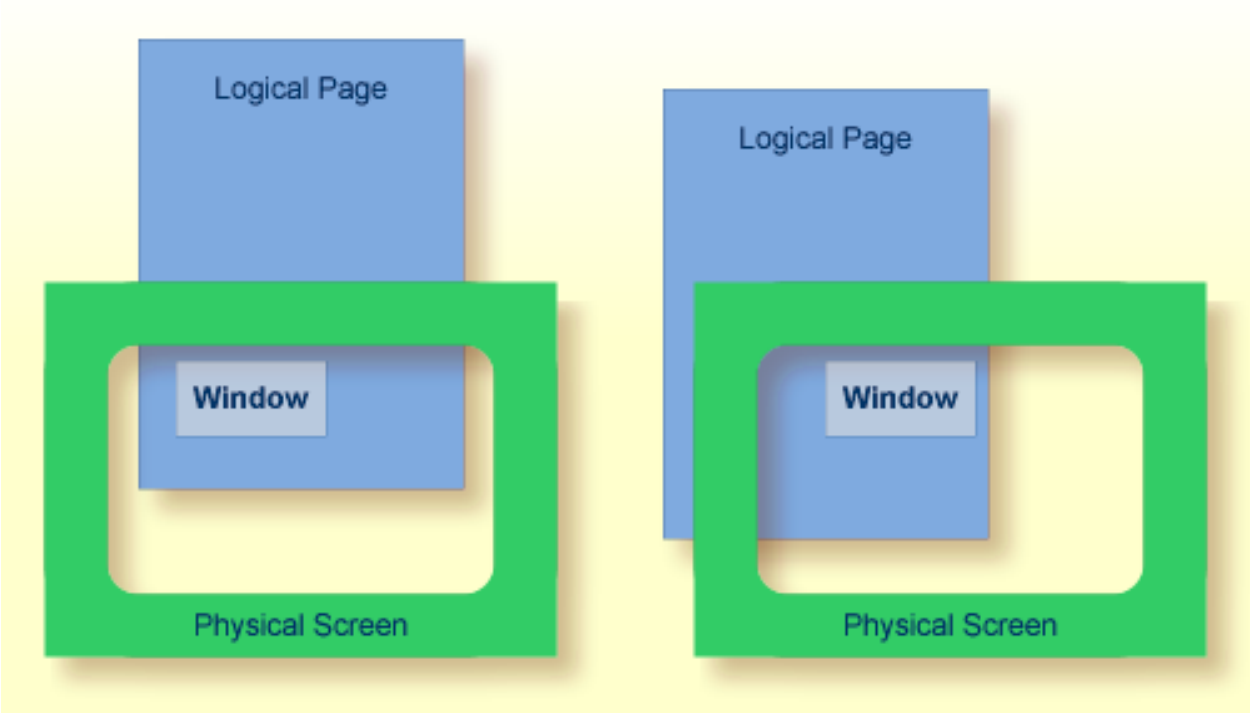
The figure below illustrates the positioning of a window on the physical screen. Note that the same section of the logical page is displayed in both cases, only the position of the window on the screen has changed.



Positioning on the Logical Page

The figure below illustrates the positioning of a window on the logical page.

When you change the position of the window on the *logical page*, the size and position of the window on the *physical screen* will remain unchanged. In other words, the window is not moved over the page, but the page is moved "underneath" the window.



DEFINE WINDOW Statement

You use the `DEFINE WINDOW` statement to specify the size, position and attributes of a window on the *physical screen*.

A `DEFINE WINDOW` statement does not activate a window; this is done with a `SET WINDOW` statement or with the `WINDOW` clause of an `INPUT` statement.

Various options are available with the `DEFINE WINDOW` statement. These are described below in the context of the following example.

The following program defines a window on the physical screen.

```
** 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
```

The window-name identifies the window. The name may be up to 32 characters long. For a window name, the same naming conventions apply as for user-defined variables. Here the name is `TEST`.

The window size is set with the `SIZE` option. Here the window is 5 lines high and 25 columns (positions) wide.

The position of the window is set by the `BASE` option. Here the top left-hand corner of the window is positioned on line 5, column 40.

With the `TITLE` option, you can define a title that is to be displayed in the window frame (of course, only if you have defined a frame for the window).

With the `CONTROL` clause, you determine whether the PF-key lines, the message line and the statistics line are displayed in the window or on the full physical screen. Here `CONTROL WINDOW` causes the message line to be displayed inside the window. `CONTROL SCREEN` would cause the lines to be displayed on the full physical screen outside the window. If you omit the `CONTROL` clause, `CONTROL WINDOW` applies by default.

With the `FRAMED` option, you define that the window is to be framed. This frame is then cursor-sensitive. Where applicable, you can page forward, backward, left or right within the window by simply placing the cursor over the appropriate symbol (<, -, +, or >; see `POSITION` clause) and then pressing Enter. In other words, you are moving the *logical page* underneath the window on the physical screen. If no symbols are displayed, you can page backward and forward within the window by placing the cursor in the top frame line (for backward positioning) or bottom frame line (for forward positioning) and then pressing Enter.

With the `POSITION` clause of the `FRAMED` option, you define that information on the position of the window on the logical page is to be displayed in the frame of the window. This applies only if the logical page is larger than the window; if it is not, the `POSITION` clause will be ignored. The position information indicates in which directions the logical page extends above, below, to the left and to the right of the current window.

If the POSITION clause is omitted, POSITION SYMBOL TOP RIGHT applies by default.

POSITION SYMBOL causes the position information to be displayed in form of symbols: “More: < - + >”. The information is displayed in the top and/or bottom frame line.

TOP/BOTTOM determines whether the position information is displayed in the top or bottom frame line.

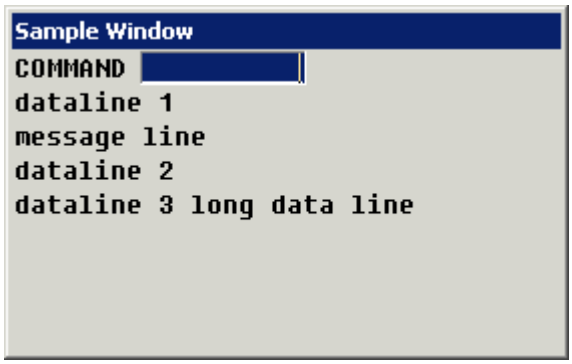
LEFT/RIGHT determines whether the position information is displayed in the left or right part of the frame line.

INPUT WINDOW Statement

The INPUT WINDOW statement activates the window defined in the DEFINE WINDOW statement. In the example, the window TEST is activated. Note that if you wish to output data in a window (for example, with a WRITE statement), you use the SET WINDOW statement.

When the above program is run, the window is displayed with one input field COMMAND. The session parameter AD is used to define that the value of the field is displayed intensified and an underscore is used as filler character.

Output of Program WINDX01:



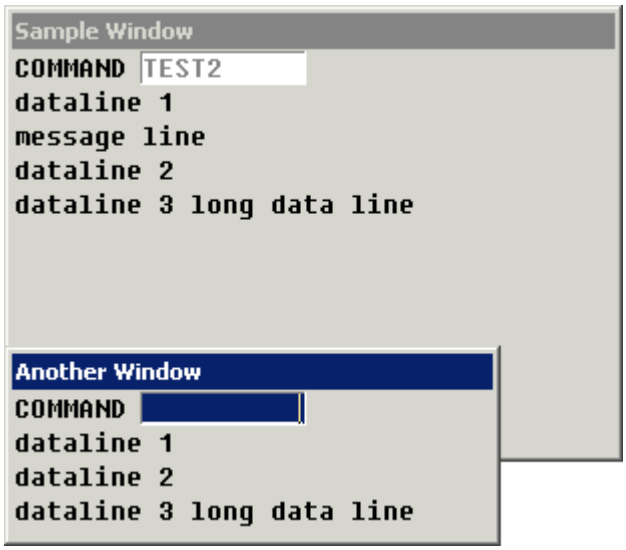
Multiple Windows

You can, of course, open multiple windows. However, only one Natural window is active at any one time, that is, the most recent window. Any previous windows may still be visible on the screen, but are no longer active and are ignored by Natural. You may enter input only in the most recent window. If there is not enough space to enter input, the window size must be adjusted first.

When TEST2 is entered in the COMMAND field, the program WINDX02 is executed.

```
** 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
```

A second window is opened. The other window is still visible, but it is inactive.



Note that for the new window the message line is now displayed at the bottom of the output window and not in the window. This was defined by the `CONTROL SCREEN` clause in the `WINDX02` program.

For further details on the statements `DEFINE WINDOW`, `INPUT WINDOW` and `SET WINDOW`, see the corresponding descriptions in the *Statements* documentation.

Standard/Dynamic Layout Maps

Standard Layout Maps

A *standard layout* can be defined in the map editor. This layout guarantees a uniform appearance for all maps that reference it throughout the application.

When a map that references a standard layout is initialized, the standard layout becomes a fixed part of the map. This means that if this standard layout is modified, all affected maps must be re-cataloged before the changes take effect.

Dynamic Layout Maps

In contrast to a standard layout, a *dynamic layout* does not become a fixed part of a map that references it, rather it is executed at runtime.

This means that if you define the layout map as "dynamic" on the Define Map Settings For MAP screen in the map editor, any modifications to the layout map are also carried out on all maps that reference it. The maps need not be re-cataloged.

For further details on layout maps, see *Map Editor* in the *Editors* documentation.

Multilingual User Interfaces

Using Natural, you can create multilingual applications for international use.

Maps, helproutines, error messages, programs, functions, subprograms and copycodes can be defined in up to 60 different languages (including languages with double-byte character sets).

Below is information on:

- Language Codes
- Defining the Language of a Natural Object
- Defining the User Language
- Referencing Multilingual Objects

- Programs
- Error Messages
- Edit Masks for Date and Time Fields

Language Codes

In Natural, each language has a *language code* (from 1 to 60). The table below is an extract from the full table of language codes. For a complete overview, refer to the description of the system variable `*LANGUAGE` in the *System Variables* documentation.

Language Code	Language	Map Code in Object Names
1	English	1
2	German	2
3	French	3
4	Spanish	4
5	Italian	5
6	Dutch	6
7	Turkish	7
8	Danish	8
9	Norwegian	9
10	Albanian	A
11	Portuguese	B

The language code (left column) is the code that is contained in the system variable `*LANGUAGE`. This code is used by Natural internally. It is the code you use to define the user language (see ***Defining the User Language*** below). The code you use to identify the language of a Natural object is the *map code* in the right-hand column of the table.

Example:

The language code for Portuguese is “11”. The code you use when cataloging a Portuguese Natural object is “B”.

For the full table of language codes, see the system variable `*LANGUAGE` as described in the *System Variables* documentation.

Defining the Language of a Natural Object

To define the language of a Natural object (map, helproutine, program, function, subprogram or copycode), you add the corresponding map code to the object name. Apart from the map code, the name of the object must be identical for all languages.

In the example below, a map has been created in English and in German. To identify the languages of the maps, the map code that corresponds to the respective language has been included in the map name.

Example of Map Names for a Multilingual Application

`DEMO1` = English map (map code 1)

`DEMO2` = German map (map code 2)

Defining Languages with Alphabetical Map Codes

Map codes are in the range 1-9, A-Z or a-y. The alphabetical map codes require special handling.

Normally, it is not possible to catalog an object with a lower-case letter in the name - all characters are automatically converted into capitals.

This is however necessary, if for example you wish to define an object for Kanji (Japanese) which has the language code 59 and the map code `x`.

To catalog such an object, you first set the correct language code (here 59) using the terminal command `%L=`*`nn`*, where *`nn`* is the language code.

You then catalog the object using the ampersand (&) character instead of the actual map code in the object name. So to have a Japanese version of the map `DEMO`, you stow the map under the name `DEMO&`.

If you now look at the list of Natural objects, you will see that the map is correctly listed as `DEMOx`.

Objects with language codes 1-9 and upper case A-Z can be cataloged directly without the use of the ampersand (&) notation.

In the example list below, you can see the three maps `DEMO1`, `DEMO2` and `DEMOx`. To delete the map `DEMOx`, you use the same method as when creating it, that is, you set the correct language with the terminal command `%L=59` and then confirm the deletion with the ampersand (&) notation (`DEMO&`).

Defining the User Language

You define the language to be used per user - as defined in the system variable *LANGUAGE - with the profile parameter ULANG (which is described in the *Parameter Reference*) or with the terminal command %L=*nn* (where *nn* is the language code).

Referencing Multilingual Objects

To reference multilingual objects in a program, you use the ampersand (&) character in the name of the object.

The program below uses the maps DEMO1 and DEMO2. The ampersand (&) character at the end of the map name stands for the map code and indicates that the map with the current language as defined in the *LANGUAGE system variable is to be used.

```
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
...
```

When this program is run, the English map (DEMO1) is displayed. This is because the current value of *LANGUAGE is 1 = English.

```
                                                    MAP DEMO1


                          SAMPLE MAP



                Please select a function!


                1.) Employee information

                2.) Vehicle information


                Enter code here: _
```

In the example below, the language code has been switched to `2` = German with the terminal command `%L=2`.

When the program is now run, the German map (`DEMO2`) is displayed.

```
                          BEISPIEL-MAP



                Bitte wählen Sie eine Funktion!


                1.) Mitarbeiterdaten

                2.) Fahrzeugdaten


              Code hier eingeben: _
```

Programs

For some applications it may be useful to define multilingual programs. For example, a standard invoicing program, might use different subprograms to handle various tax aspects, depending on the country where the invoice is to be written.

Multilingual programs are defined with the same technique as described above for maps.

Error Messages

Using the Natural utility SYSERR, you can translate Natural error messages into up to 60 languages, and also define your own error messages.

Which message language a user sees, depends on the `*LANGUAGE` system variable.

For further information on error messages, see *SYSERR Utility* in the *Utilities* documentation.

Edit Masks for Date and Time Fields

The language used for date and time fields defined with edit masks also depends on the system variable `*LANGUAGE`.

For details on edit masks, see the session parameter `EM` as described in the *Parameter Reference*.

Skill-Sensitive User Interfaces

Users with varying levels of skill may wish to have different maps (of varying detail) while using the same application.

If your application is not for international use by users speaking different languages, you can use the techniques for multilingual maps to define maps of varying detail.

For example, you could define language code 1 as corresponding to the skill of the beginner, and language code 2 as corresponding to the skill of the advanced user. This simple but effective technique is illustrated below.

The following map (PERS1) includes instructions for the end user on how to select a function from the menu. The information is very detailed. The name of the map contains the map code 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
```

The same map, but without the detailed instructions is saved under the same name, but with map code 2.

```
                                                MAP PERS2


                            SAMPLE MAP

                  Please select a function


                  1.) Employee information  _

                  2.) Vehicle information   _

                      Enter code:  _
```

In the example above, the map with the detailed instructions is output, if the ULANG profile parameter has the value 1, the map without the instructions if the value is 2. See also the description of the profile parameter ULANG (in the *Parameter Reference*).

111 Dialog Design

- Field-Sensitive Processing 850
- Simplifying Programming 852
- Line-Sensitive Processing 853
- Column-Sensitive Processing 854
- Processing Based on Function Keys 854
- Processing Based on Function-Key Names 855
- Processing Data Outside an Active Window 856
- Copying Data from a Screen 859
- Statements REINPUT/REINPUT FULL 862
- Object-Oriented Processing - The Natural Command Processor 863

This chapter tells you how you can design character-based user interfaces that make user interaction with the application simple and flexible.

Note: For information on the design of graphical user interfaces (GUI), refer to *Introduction to Event-Driven Programming*.

Field-Sensitive Processing

*CURS-FIELD and POS(field-name)

Using the system variable `*CURS-FIELD` together with the system function `POS(field-name)`, you can define processing based on the field where the cursor is positioned at the time the user presses Enter.

`*CURS-FIELD` contains the internal identification of the field where the cursor is currently positioned; it cannot be used by itself, but only in conjunction with `POS(field-name)`.

You can use `*CURS-FIELD` and `POS(field-name)`, for example, to enable a user to select a function simply by placing the cursor on a specific field and pressing Enter.

The example below illustrates such an application:

```
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
```

And the result:

```
                                SAMPLE MAP

                    Please select a function


                    1.) Employee information  _
                    2.) Vehicle information   _  <== Cursor positioned
                                                       on field

                        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
```

If the user places the cursor on the input field (#EMP) next to Employee information, and presses Enter, the program LISTEMP displays a list of employee names:

```
 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
```

Notes:

1. In Natural for Ajax applications, *CURS-FIELD identifies the operand that represents the value of the control that has the input focus. You may use *CURS-FIELD in conjunction with the POS function to check for the control that has the input focus and perform processing depending on that condition.

2. The values of *CURS-FIELD and POS(*field-name*) serve for internal identification of the fields only. They cannot be used for arithmetical operations.

Simplifying Programming

System Function POS

The Natural system function POS(*field-name*) contains the internal identification of the field whose name is specified with the system function.

POS(*field-name*) may be used to identify a specific field, regardless of its position in a map. This means that the sequence and number of fields in a map may be changed, but POS(*field-name*) will still uniquely identify the same field. With this, for example, you need only a single REINPUT statement to make the field to be MARKed dependent on the program logic.

Note: The value POS(*field-name*) serves for internal identification of the fields only. It cannot be used for arithmetical operations.

Example:

```
...
DECIDE ON FIRST VALUE OF ...
   VALUE ...
      COMPUTE #FIELDX = POS(FIELD1)
   VALUE ...
      COMPUTE #FIELDX = POS(FIELD2)
   ...
END-DECIDE
...
REINPUT ... MARK #FIELDX
...
```

Full details on *CURS-FIELD and POS(*field-name*) are described in the *System Variables* and *System Functions* documention.

Line-Sensitive Processing

System Variable *CURS-LINE

Using the system variable *CURS-LINE, you can make processing dependent on the line where the cursor is positioned at the time the user presses Enter.

Using this variable, you can make user-friendly menus. With the appropriate programming, the user merely has to place the cursor on the line of the desired menu option and press Enter to execute the option.

The cursor position is defined within the current active window, regardless of its physical placement on the screen.

Note: The message line, function-key lines and statistics line/infoline are not counted as data lines on the screen.

The example below demonstrates line-sensitive processing using the *CURS-LINE system variable. When the user presses Enter on the map, the program checks if the cursor is positioned on line 8 of the screen which contains the option Employee information. If this is the case, the program that lists the names of employees LISTEMP is executed.

```
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
```

Output:

```
                              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
```

The user places the cursor indicated by square brackets [] on the line of the desired option and presses Enter and the corresponding program is executed.

Column-Sensitive Processing

System Variable *CURS-COL

The system variable `*CURS-COL` can be used in a similar way to `*CURS-LINE` described above. With `*CURS-COL` you can make processing dependent on the column where the cursor is positioned on the screen.

Processing Based on Function Keys

System Variable *PF-KEY

Frequently you may wish to make processing dependent on the function key a user presses.

This is achieved with the statement `SET KEY`, the system variable `*PF-KEY` and a modification of the default map settings (`Standard Keys = Y`).

The `SET KEY` statement assigns functions to function keys during program execution. The system variable `*PF-KEY` contains the identification of the last function key the user pressed.

The example below illustrates the use of `SET KEY` in combination with `*PF-KEY`.

```
...
SET KEY PF1
*
INPUT USING MAP 'DEMO&'
IF *PF-KEY = 'PF1'
  WRITE 'Help is currently not active'
END-IF
...
```

The SET KEY statement activates PF1 as a function key.

The IF statement defines what action is to be taken when the user presses PF1. The system variable *PF-KEY is checked for its current content; if it contains PF1, the corresponding action is taken.

Further details regarding the statement SET KEY and the system variable *PF-KEY are described in the *Statements* and the *System Variables* documentation respectively.

Processing Based on Function-Key Names

System Variable *PF-NAME

When defining processing based on function keys, further comfort can be added by using the system variable *PF-NAME. With this variable you can make processing dependent on the name of a function, not on a specific key.

The variable *PF-NAME contains the name of the last function key the user pressed (that is, the name as assigned to the key with the NAMED clause of the SET KEY statement).

For example, if you wish to allow users to invoke help by pressing either PF3 or PF12, you assign the same name (in the example below: INFO) to both keys. When the user presses either one of the keys, the processing defined in the IF statement is performed.

```
...
SET KEY PF3  NAMED 'INFO'
        PF12 NAMED 'INFO'
INPUT USING MAP 'DEMO&'
IF *PF-NAME = 'INFO'
  WRITE 'Help is currently not active'
END-IF
...
```

The function names defined with NAMED appear in the function-key lines:

Processing Data Outside an Active Window

Below is information on:

- System Variable *COM
- Example Usage of *COM
- Positioning the Cursor to *COM - the %T* Terminal Command

System Variable *COM

As stated in the section *Screen Design - **Windows***, only *one* window is active at any one time. This normally means that input is only possible within that particular window.

Using the `*COM` system variable, which can be regarded as a communication area, it is possible to enter data outside the current window.

The prerequisite is that a map contains `*COM` as a modifiable field. This field is then available for the user to enter data when a window is currently on the screen. Further processing can then be made dependent on the content of `*COM`.

This allows you to implement user interfaces as already used, for example, by Con-nect, Software AG's office system, where a user can always enter data in the command line, even when a window with its own input fields is active.

Note that `*COM` is only cleared when the Natural session is ended.

Example Usage of *COM

In the example below, the program ADD performs a simple addition using the input data from a map. In this map, `*COM` has been defined as a modifiable field (at the bottom of the map) with the length specified in the `AL` field of the Extended Field Editing. The result of the calculation is displayed in a window. Although this window offers no possibility for input, the user can still use the `*COM` field in the map outside the window.

Program 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
```

Output of Program 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:
```

In this example, by entering the value `M`, the user initiates a multiplication function; the two values from the input map are multiplied and the result is displayed in a second window:

```
                 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
```

Positioning the Cursor to *COM - the %T* Terminal Command

Normally, when a window is active and the window contains no input fields (`AD=M` or `AD=A`), the cursor is placed in the top left corner of the window.

With the terminal command `%T*`, you can position the cursor to a `*COM` system variable outside the window when the active window contains no input fields.

By using `%T*` again, you can switch back to standard cursor placement.

Example:

```
...
INPUT USING MAP 'WINDOW'
*
COMPUTE #SUM3 = #VALUE1 + #VALUE2
*
SET CONTROL 'T*'
SET WINDOW 'EMP'
INPUT (AD=O) / 'Value 1 +' /
               'Value 2 =' //
               ' ' #SUM3
...
```

Copying Data from a Screen

Below is information on:

- Terminal Commands %CS and %CC
- Selecting a Line from Report Output for Further Processing

Terminal Commands %CS and %CC

With these terminal commands, you can copy parts of a screen into the Natural stack (`%CS`) or into the system variable `*COM` (`%CC`). The protected data from a specific screen line are copied field by field.

The full options of these terminal commands are described in the *Terminal Commands* documentation.

Once copied to the stack or `*COM`, the data are available for further processing. Using these commands, you can make user-friendly interfaces as in the example below.

Selecting a Line from Report Output for Further Processing

In the following example, the program COM1 lists all employee names from Abellan to Alestia.

Program 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
```

Output of Program 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
```

Control is now passed to the program COM2.

Program 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 this program, the terminal command `%CCC` is assigned to PF5. The terminal command copies all protected data from the line where the cursor is positioned to the system variable `*COM`. This in-

formation is then available for further processing. This further processing is defined in the program lines shown in boldface.

The user can now position the cursor on the name that interests him; when he/she now presses PF5, further employee information is supplied.

```
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 this case, the personnel ID of the selected employee is displayed:

```
 Page      1                                             2006-08-12  09:44:52

         NAME         PERSONNEL
                         ID
 -------------------- ---------

 ADAM                 50005800
```

Statements REINPUT/REINPUT FULL

If you wish to return to and re-execute an INPUT statement, you use the REINPUT statement. It is generally used to display a message indicating that the data input as a result of the previous INPUT statement were invalid.

If you specify the FULL option in a REINPUT statement, the corresponding INPUT statement will be re-executed fully:

- With an ordinary REINPUT statement (without FULL option), the contents of variables that were changed between the INPUT and REINPUT statement will not be displayed; that is, all variables on the screen will show the contents they had when the INPUT statement was originally executed.
- With a REINPUT FULL statement, all changes that have been made after the initial execution of the INPUT statement will be applied to the INPUT statement when it is re-executed; that is, all variables on the screen contain the values they had when the REINPUT statement was executed.
- If you wish to position the cursor to a specified field, you can use the MARK option, and to position to a particular position within a specified field, you use the MARK POSITION option.

The example below illustrates the use of REINPUT FULL with 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
```

The user enters 3 in field #B and 3 in field #C and presses Enter.

```
  #A            #B      3 #C      3
```

The program requires field #A to be non-blank. The REINPUT FULL statement with MARK POSITION 5 IN *#A returns the input screen; the now modified variable #B contains the value 6 (after the COMPUTE calculation has been performed). The cursor is positioned to the 5th position in field #A ready for new input.

```
Enter name of field
  #A    _       #B     6 #C     0



Enter a value
```

This is the screen that would be returned by the same statement, without the `FULL` option. Note that the variables `#B` and `#C` have been reset to their status at the time of execution of the `INPUT` statement (each field contains the value 3).

```
   #A    _       #B     3 #C     3
```

Object-Oriented Processing - The Natural Command Processor

The Natural Command Processor is used to define and control navigation within an application. It consists of two parts: The development part and the run-time part.

- The development part is the utility SYSNCP. With this utility, you define commands and the actions to be performed in response to the execution of these commands. From your definitions, SYSNCP generates decision tables which determine what happens when a user enters a command.
- The run-time part is the statement `PROCESS COMMAND`. This statement is used to invoke the Command Processor within a Natural program. In the statement you specify the name of the `SYSNCP` table to be used to handle the data input by a user at that point.

For further information regarding the Natural Command Processor, see *SYSNCP Utility* in the *Utilities* documentation and the statement `PROCESS COMMAND` as described in the *Statements* documentation.

XIII Natural Native Interface

This part covers the following topics:

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

112 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`.

113 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.

114 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.

115 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.

116 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, UNIX and OpenVMS, 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`.

117 Interface Functions

- nni_get_interface 879
- nni_free_interface 880
- nni_initialize 880
- nni_is_initialized 882
- nni_uninitialize 882
- nni_enter 883
- nni_try_enter 883
- nni_leave 884
- nni_logon 885
- nni_logoff 885
- nni_callnat 886
- nni_create_object 887
- nni_send_method 888
- nni_get_property 890
- nni_set_property 891
- nni_delete_object 893
- nni_create_parm 894
- nni_create_module_parm 895
- nni_create_method_parm 896
- nni_create_prop_parm 897
- nni_parm_count 898
- nni_init_parm_s 898
- nni_init_parm_sa 899
- nni_init_parm_d 901
- nni_init_parm_da 901
- nni_get_parm_info 903
- nni_get_parm 903
- nni_get_parm_array 905
- nni_get_parm_array_length 906
- nni_put_parm 907
- nni_put_parm_array 908
- nni_resize_parm_array 909

- nni_delete_parm 910
- nni_from_string 911
- nni_to_string 912

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_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 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 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	For example, `-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

118 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 indicated with 0. The length of an individual occurrence must then be determined with the function `nni_get_parm_array_length`.

Format	Element Name	Content
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]

119 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 and OpenVMS 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 *%NAT-DIR%\%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

120 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.

121 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.

122 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.

123 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 *%NATDIR%\%NATVERS%\samples\sysexnni*.

124 Threading Issues

A Natural process on Windows, UNIX and OpenVMS 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.

XIV NaturalX

This part describes how to develop and distribute object-based applications.

The following topics are covered:

Introduction to NaturalX
Developing NaturalX Applications
Distributing NaturalX Applications
ActiveX Component SoftwareAG.NaturalX.Utilities
Interface INaturalXUtilities
Interface IRunningObjects
ActiveX Component SoftwareAG.NaturalX.Enumerator
Interface IEnumerator

125 Introduction to NaturalX

- Why NaturalX? 932
- Programming Techniques 933

This chapter contains a short introduction to component-based programming involving the use of the NaturalX interface and a dedicated set of Natural statements.

Why NaturalX?

Software applications that are based on component architecture offer many advantages over traditional designs. These include the following:

- Faster development. Programmers can build applications faster by assembling software from prebuilt components.
- Reduced development costs. Having a common set of interfaces for programs means less work integrating the components into complete solutions.
- Improved flexibility. It is easier to customize software for different departments within a company by just changing some of the components that constitute the application.
- Reduced maintenance costs. In the case of an upgrade, it is often sufficient to change some of the components instead of having to modify the entire application.
- Easier distribution. Components encapsulate data structures and functionality in distributable units.

Using NaturalX you can create component-based applications.

You can use NaturalX in conjunction with DCOM. This enables you to:

- allow your components to be accessed by other components,
- execute these components on local and/or remote servers,
- access components written in a variety of programming languages across process and machine boundaries from within Natural programs,
- provide your existing Natural applications with (quasi) standardized interfaces.

The following scenario illustrates how a company could exploit these advantages. A company introduces a new sales management system that is based on an application design using components. There are numerous data entry components in the application, one for each sales point. But all of these sales points use a common tax calculation component that runs on a server. If the tax legislation is changed, then only the tax component has to be updated instead of changing the data entry components at each site. In addition, the life of the programmers is made easier because they do not have to worry about network programming and the integration of components that are written in different languages.

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.

126 Developing NaturalX Applications

- Development Environments 938
- Defining Classes 938
- Using Classes and Objects 943

This chapter describes how to develop an application by defining and using classes.

Development Environments

- **Developing Classes on Windows Platforms**
  On Windows platforms, Natural provides the Class Builder as the tool to develop Natural classes. The Class Builder shows a Natural class in a structured hierarchical order and allows the user to manage the class and its components efficiently. If you use the Class Builder, no knowledge or only a basic knowledge of the syntax elements described below is required.
- **Developing Classes Using SPoD**
  In a Natural Single Point of Development (SPoD) environment that includes a Mainframe, UNIX and/or OpenVMS remote development server, you can use the Class Builder available with the Natural Studio front-end to develop classes on Mainframe, UNIX and/or OpenVMS platforms. In this case, no knowledge or only a basic knowledge of the syntax elements described below is required.
- **Developing Classes on Mainframe, UNIX or OpenVMS Platforms**
  If you do not use SPoD, you develop classes on these platforms using the Natural program editor. In this case, you should know the syntax of class definition described below.

Defining Classes

When you define a class, you must create a Natural class module, within which you create a `DEFINE CLASS` statement. Using the `DEFINE CLASS` statement, you assign the class an externally usable name and define its interfaces, methods and properties. You can also assign an object data area to the class, which describes the layout of an instance of the class. The `DEFINE CLASS` statement is also used to supply a global unique identifier to those classes that are to be registered as COM classes.

This section covers the following topics:

- Creating a Natural Class Module
- Specifying a Class
- Defining an Interface
- Assigning an Object Data Variable to a Property
- Assigning a Subprogram to a Method

- Implementing Methods

Creating a Natural Class Module

To create a Natural class module

- Use the `CREATE OBJECT` statement to create a Natural object of type Class.

Specifying a Class

The `DEFINE CLASS` statement defines the name of the class, the interfaces the class supports and the structure of its objects. For classes that are to be registered as COM classes, it specifies also the globally unique ID of the class and its activation policy.

To specify a class

- Use the `DEFINE CLASS` statement as described in the *Statements* documentation.

Defining an Interface

Each interface of a class is specified with an `INTERFACE` statement inside the class definition. An `INTERFACE` statement specifies the name of the interface and a number of properties and methods. For classes that are to be registered as COM classes, it specifies also the globally inique ID of the interface.

A class can have one or several interfaces. For each interface, one `INTERFACE` statement is coded in the class definition. Each `INTERFACE` statement contains one or several `PROPERTY` and `METHOD` clauses. Usually the properties and methods contained in one interface are related from either a technical or a business point of view.

The `PROPERTY` clause defines the name of a property and assigns a variable from the object data area to the property. This variable is used to store the value of the property.

The `METHOD` clause defines the name of a method and assigns a subprogram to the method. This subprogram is used to implement the method.

To define an interface

- Use the `INTERFACE` statement as described in the *Statements* documentation.

Assigning an Object Data Variable to a Property

The PROPERTY statement is used only when several classes are to implement the same interface in different ways. In this case, the classes share the same interface definition and include it from a Natural **copycode**. The PROPERTY statement is then used to assign a variable from the object data area to a property, *outside* the interface definition. Like the PROPERTY clause of the INTERFACE statement, the PROPERTY statement defines the name of a property and assigns a variable from the object data area to the property. This variable is used to store the value of the property.

To assign an object data variable to a property

- Use the PROPERTY statement as described in the *Statements* documentation.

Assigning a Subprogram to a Method

The METHOD statement is used only when several classes are to implement the same interface in different ways. In this case, the classes share the same interface definition and include it from a Natural **copycode**. The METHOD statement is then used to assign a subprogram to the method, *outside* the interface definition. Like the METHOD clause of the INTERFACE statement, the METHOD statement defines the name of a method and assigns a subprogram to the method. This subprogram is used to implement the method.

To assign a subprogram to a method

- Use the METHOD statement as described in the *Statements* documentation.

Implementing Methods

A method is implemented as a Natural subprogram in the following general form:

```
DEFINE DATA statement
*
* Implementation code of the method
*
END
```

For information on the DEFINE DATA statement see the *Statements* documentation.

All clauses of the DEFINE DATA statement are optional.

It is recommended that you use data areas instead of inline data definitions to ensure data consistency.

If a PARAMETER clause is specified, the method can have parameters and/or a return value.

Parameters that are marked BY VALUE in the parameter data area are input parameters of the method.

Parameters that are not marked BY VALUE are passed "by reference" and are input/output parameters. This is the default.

The first parameter that is marked BY VALUE RESULT is returned as the return value for the method. If more than one parameter is marked in this way, the others will be treated as input/output parameters.

Parameters that are marked OPTIONAL need not be specified when the method is called. They can be left unspecified by using the *n*X notation in the SEND METHOD statement.

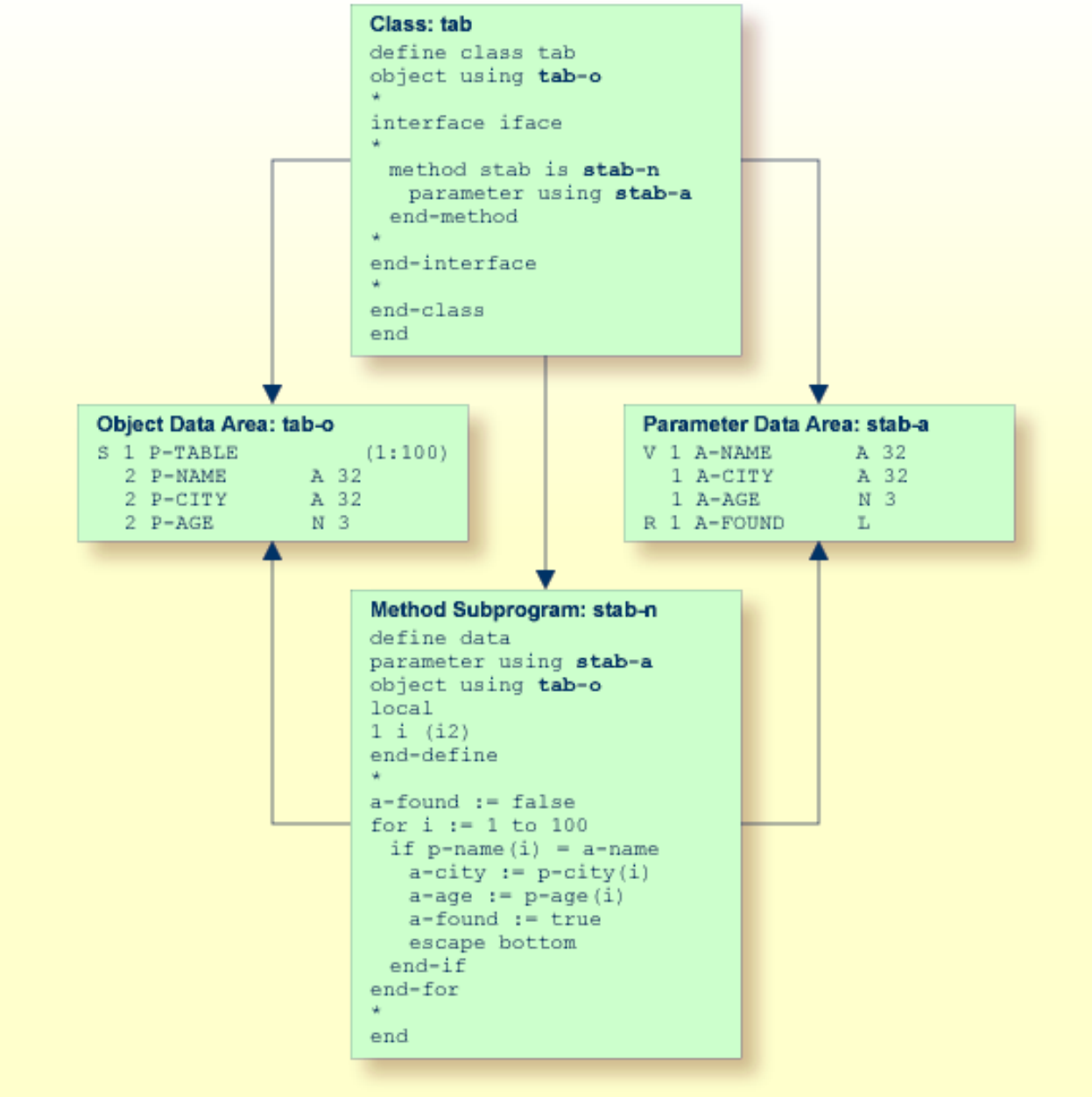
To make sure that the method subprogram accepts exactly the same parameters as specified in the corresponding METHOD statement in the class definition, use a parameter data area instead of inline data definitions. Use the same parameter data area as in the corresponding METHOD statement.

To give the method subprogram access to the object data structure, the OBJECT clause can be specified. To make sure that the method subprogram can access the object data correctly, use a local data area instead of inline data definitions. Use the same local data area as specified in the OBJECT clause of the DEFINE CLASS statement.

The GLOBAL, LOCAL and INDEPENDENT clauses can be used as in any other Natural program.

While technically possible, it is usually not meaningful to use a CONTEXT clause in a method subprogram.

The following example retrieves data about a given person from a table. The search key is passed as a BY VALUE parameter. The resulting data is returned through "by reference" parameters ("by reference" is the default definition). The return value of the method is defined by the specification BY VALUE RESULT.


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

Using Classes and Objects

Objects created in a local Natural session can be accessed by other modules in the same Natural session.

Objects created in other processes or on remote machines can be accessed via DCOM.

In both cases the rules for accessing and using classes and their objects are the same.

The statement `CREATE OBJECT` is used to create an object (also known as an instance) of a given class.

To reference objects in Natural programs, object handles have to be defined in the `DEFINE DATA` statement. Methods of an object are invoked with the statement `SEND METHOD`. Objects can have properties, which can be accessed using the normal assignment syntax.

Note: In order to use a NaturalX class via DCOM, the class must first be registered.

These steps are described below:

- Defining Object Handles
- Creating an Instance of a Class
- Invoking a Particular Method of an Object
- Accessing Properties
- Sample Application

Defining Object Handles

To reference objects in Natural programs, object handles have to be defined as follows in the `DEFINE DATA` statement:

```
DEFINE DATA
  level-handle-name [(array-definition)] HANDLE OF OBJECT
  ...
END-DEFINE
```

Example:

```
DEFINE DATA LOCAL
1 #MYOBJ1 HANDLE OF OBJECT
1 #MYOBJ2 (1:5) HANDLE OF OBJECT
END-DEFINE
```

Creating an Instance of a Class

To create an instance of a class

- Use the `CREATE OBJECT` statement as described in the *Statements* documentation.

Invoking a Particular Method of an Object

To invoke a particular method of an object

- Use the `SEND METHOD` statement as described in the *Statements* documentation.

Accessing Properties

Properties can be accessed using the `ASSIGN` (or `COMPUTE`) statement as follows:

```
ASSIGN operand1.property-name = operand2
ASSIGN operand2 = operand1.property-name
```

Object Handle - *operand1*

operand1 must be defined as an object handle and identifies the object whose property is to be accessed. The object must already exist.

operand2

As *operand2*, you specify an operand whose format must be data transfer-compatible to the format of the property. Please refer to the **data transfer compatibility rules** for further information.

If the object is to be accessed via DCOM, you must also take into account the rules for data type conversion which are outlined in the section *Using Type Information* in the *Operations* documentation.

property-name

The name of a property of the object.

If the property name conforms to Natural identifier syntax, it can be specified as follows

```
create object #o1 of class "Employee"
  #age := #o1.Age
```

If the property name does not conform to Natural identifier syntax, it must be enclosed in angle brackets:

```
create object #o1 of class "Employee"
  #salary := #o1.<<%Salary>>
```

The property name can also be qualified with an interface name. This is necessary if the object has more than one interface containing a property with the same name. In this case, the qualified property name must be enclosed in angle brackets:

```
create object #o1 of class "Employee"
  #age := #o1.<<PersonalData.Age>>
```

Example:

```
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
```

In order to use arrays of object handles and properties that have arrays as values correctly, it is important to know the following:

A property of an occurrence of an array of object handles is addressed with the following index notation:

```
#p.Salary(2) := #salary
```

A property that has an array as value is always accessed as a whole. Therefore no index notation is necessary with the property name:

```
#o.SalaryHistory := #history(1:10)
```

A property of an occurrence of an array of object handles which has an array as value is therefore addressed as follows:

```
#p.SalaryHistory(2) := #history(1:10)
```

Sample Application

An example application is provided in the libraries SYSEXCOM and SYSEXCOC. See the A-README members in these libraries for information about how to run the example.

127 Distributing NaturalX Applications

- General 948
- Globally Unique Identifiers - GUIDs 950

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.

128 ActiveX Component SoftwareAG.NaturalX.Utilities

- Purpose 952
- Interfaces 954

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
```

This example will run successfully only if it is executed in a Natural Studio session. If it is executed in a Natural runtime session or under the Natural debugger, the call to `GetThisNaturalStudio` will return `NULL-HANDLE`. This is because only a Natural Studio session has an `INatAutoStudio` interface. If the program is running under the Natural debugger, it is effectively executed in a Natural runtime session outside Natural Studio; therefore, the call to `GetThisNaturalStudio` will also return `NULL-HANDLE`.

To make this example run also under the Natural debugger, modify it as follows, in order to retrieve the `INatAutoStudio` interface of the Natural Studio session:

```
define data
local
1 #util handle of object
1 #studio handle of object
1 #rot handle of object
1 #ro (a) dynamic
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
* We might be in a debugging session.
* Try to locate the Natural Studio session
* from which the debugger has been started.
* 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, we access it.
    if substring(#ro,1,13) eq 'NaturalStudio'
      send 'BindToObject' to #util
      with #ro (ad=o) return #studio
      escape bottom
    end-if
  end-repeat
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`

129 Interface INaturalXUtilities

- Purpose 956
- Methods 956

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
```

130 Interface IRunningObjects

- Purpose 960
- Methods 962

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.

131 ActiveX Component SoftwareAG.NaturalX.Enumerator

- Purpose 964
- Interface 965

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`

132 Interface IEnumerator

- Purpose 968
- Methods 968

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.

XV

- 133 Natural Reserved Keywords 973
- 134 Referenced Example Programs 991

133 Natural Reserved Keywords

- Alphabetical List of Natural Reserved Keywords 974
- Performing a Check for Natural Reserved Keywords 989

This chapter contains a list of all keywords that are reserved in the Natural programming language.

Important: To avoid any naming conflicts, you are strongly recommended not to use Natural reserved keywords as names for variables.

Alphabetical List of Natural Reserved Keywords

The following list is an overview of Natural reserved keywords and is for general information only. In case of doubt, use the **keyword check** function of the compiler.

[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

Performing a Check for Natural Reserved Keywords

There is a subset of Natural keywords which, when used as names for variables, would be ambiguous. These are in particular keywords which identify Natural statements (`ADD`, `FIND`, etc.) or system functions (`ABS`, `SUM`, etc.). If you use such a keyword as the name of a variable, you cannot use this variable in the context of optional operands (with `CALLNAT`, `WRITE`, etc.).

Example:

```
DEFINE DATA LOCAL
1 ADD (A10)
END-DEFINE
CALLNAT 'MYSUB' ADD 4     /* ADD is regarded as ADD statement
END
```

To check variable names in a programming object against such Natural reserved keywords, you can use the Natural profile parameter `KCHECK` or the `KCHECK` option of the `COMPOPT` system command.

The following table contains a list of Natural reserved keywords that are checked by `KC` or `KCHECK`.

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			
	FRAC			

By default, no keyword check is performed.

134 Referenced Example Programs

- READ Statement 992
- FIND Statement 993
- Nested READ and FIND Statements 997
- ACCEPT and REJECT Statements 999
- AT START OF DATA and AT END OF DATA Statements 1001
- DISPLAY and WRITE Statements 1004
- DISPLAY Statement 1008
- Column Headers 1009
- Field-Output-Relevant Parameters 1011
- Edit Masks 1017
- DISPLAY VERT with WRITE Statement 1020
- AT BREAK Statement 1021
- COMPUTE, MOVE and COMPRESS Statements 1022
- System Variables 1025
- System Functions 1028

This chapter contains some additional example programs that are referenced in the *Programming Guide*.

READ Statement

The following example is referenced in the section ***Statements for Database Access***.

READX03 - READ statement (with LOGICAL clause)

```
** 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
```

Output of Program 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

The following examples are referenced in the section *Statements for Database Access*.

FINDX07 - FIND statement (with several clauses)

```
** 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
```

Output of Program FINDX07:

```
        NAME              FIRST-NAME              CITY           ANNUAL
                                                                 SALARY
-------------------- -------------------- -------------------- ----------

BAKER                PAULINE              DERBY                      4450
JONES                MARTHA               KALAMAZOO                 21000
JONES                KEVIN                DERBY                      7000
```

FINDX08 - FIND statement (with 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
```

Runtime Error Caused by Program FINDX08:

```
NAT1005 More records found than specified in search limit.
```

FINDX09 - FIND statement (using *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
```

Output of Program FINDX09:

```
  COUNTER           NAME         DEPARTMENT     ISN
                                    CODE
----------- -------------------- ---------- -----------

(TOTAL NUMBER IN BOSTON:           7 )

          1 STANWOOD             TECH10             782
          2 PERREAULT            TECH10             842
```

FINDX10 - FIND statement (combined with 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
```

Output of Program 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 (with *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
```

Output of Program 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
```

Nested READ and FIND Statements

The following examples are referenced in the section *Database Processing Loops*.

READX04 - READ statement (in combination with FIND and the system variables *NUMBER and *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
```

Output of Program 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 (for READ, FIND loop processing)

```
** 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
```

Output of Program 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 and REJECT Statements

The following examples are referenced in the section *Selecting Records Using ACCEPT/REJECT*.

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
```

Output of Program 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
```

Output of Program 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
```

Output of Program 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 and AT END OF DATA Statements

The following examples are referenced in the section *AT START/END OF DATA Statements*.

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
```

Output of Program 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
```

Output of Program 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 combination with 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
```

Output of Program 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 and WRITE Statements

The following examples are referenced in the section *Statements DISPLAY and WRITE*.

DISPLX13 - DISPLAY statement (compare with WRITEX08 using 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
```

Output of Program 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 (compare with DISPLX13 using 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
```

Output of Program 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 (with AL, SF and *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
```

Output of Program 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 combination with 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
```

Output of Program 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

The following example is referenced in the section *Page Titles, Page Breaks, Blank Lines*.

DISPLX21 DISPLAY statement (with slash '/' and compare with 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
```

Output of Program 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

REGISTER OF
SALT LAKE CITY
```

```
SAMUELSON            SALT LAKE CITY       7610 W. 86TH STREET
MARTIN

                                  REGISTER OF
                                SALT LAKE CITY
------------------------------------------------------------------------------
```

Column Headers

The following example is referenced in the section *Column Headers*.

DISPLX15 - DISPLAY statement (with 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
```

Output of Program 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 (with '/', '*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
```

Output of Program 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
```

Field-Output-Relevant Parameters

The following examples are referenced in the section *Parameters to Influence the Output of Fields*.

They are provided to demonstrate the use of the parameters LC, IC, TC, AL, NL, IS, ZP and ES, and the SUSPEND IDENTICAL SUPPRESS statement:

DISPLX17 - DISPLAY statement (with 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
```

Output of Program 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 (using default settings for SF, AL, UC, LC, IC, TC and compare with 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
```

Output of Program 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 (with SF, AL, LC, IC, TC and compare with 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
```

Output of Program 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 conjunction with parameters IS, ES, ZP in 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
```

Output of Program 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 conjunction with parameters IS, ES, ZP in DISPLAY) Identical to SUSPEX01, but with 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
```

Output of Program 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

```
** 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
```

Output of Program 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
```

Edit Masks

The following examples are referenced in the section *Edit Masks - EM Parameter*.

EDITMX03 - Edit mask (different EM for alpha-numeric fields)

```
** 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^H)
                     FIRST-NAME (EM=' - 'X(15)*)
                     CITY       (EM=X..X(10))
  SKIP 1
END-READ
END
```

Output of Program 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 - Edit mask (different EM for numeric fields)

```
** 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
```

Output of Program 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 - Edit mask (EM for date and time system variables)

```
** 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
```

Output of Program 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 with WRITE Statement

WRITEX10 - WRITE statement (with *n*T, Tfield* and 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
```

Output of Program 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

The following example is referenced in the section ***Control Breaks***.

ATBREX06 - AT BREAK OF statement (comparing NMIN, NAVER, NCOUNT with 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
```

Output of Program 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
```

COMPUTE, MOVE and COMPRESS Statements

The following examples are referenced in the section *Data Computation*.

WRITEX11 - WRITE statement (with *n*X, *n*/*n* and 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
```

Output of Program 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
```

Output of Program 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 (using parameters 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
```

Output of Program 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
```

System Variables

The following examples are referenced in the section ***System Variables and System Functions***.

EDITMX05 - Edit mask (EM for date and time system variables)

```
** 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
```

Output of Program 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 combination with FIND and the system variables *NUMBER and *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
```

Output of Program 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 (with *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
```

Output of Program WTITLX01:

```
YEAR:  1980               PAGE      1
YEAR          MAKE          MAINT-COST
----- -------------------- ----------

 1980 RENAULT                   20000
 1980 RENAULT                   20000
 1980 PEUGEOT                   20000
```

System Functions

The following examples are referenced in the section *System Variables and System Functions*.

ATBREX06 - AT BREAK OF statement (comparing NMIN, NAVER, NCOUNT with 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
```

Output of Program 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 (with system function available via GIVE SYSTEM FUNCTIONS in DISPLAY)

```
** 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
```

Output of Program 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
```