NATURAL

Natural

Statements

Version 5.1.1 for Windows
Version 3.1.6 for Mainframes
Version 5.1.1 for UNIX and OpenVMS

software AG

This document applies to Natural Version 5.1.1 for Windows, Version 3.1.6 for Mainframes, Version 5.1.1 for UNIX and OpenVMS, and to all subsequent releases. Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

© Macrh 2002, Software AG
All rights reserved

Software AG and/or all Software AG products are either trademarks or registered trademarks of Software AG. Other products and company names mentioned herein may be the trademarks of their respective owners.

Table of Contents

Statements - Overview . . . 1
Statements - Overview . . . 1
Syntax Symbols and Operand Definition Tables . . . 3
Syntax Symbols and Operand Definition Tables . . . 3
Syntax Symbols . . . 3
Operand Definition Table . . . 5
Possible Structure . . . 5
Possible Formats . . . 5
Reference Permitted . . . 6
Dynamic Definition . . . 6
Statements Grouped by Functions . . . 7
Statements Grouped by Functions . . . 7
Database Access and Update . . . 8
Arithmetic and Data Movement Operations . . . 9
Loop Execution . . . 9
Creation of Output Reports . . . 10
Screen Generation for Interactive Processing . . . 10
Processing of Logical Conditions . . . 11
Invoking Programs and Routines . . . 11
Control of Work Files . . . 11
Component Based Programming . . . 12
Event-Driven Programming . . . 12
Miscellaneous . . . 13
ACCEPT/REJECT . . . 14
ACCEPT/REJECT . . . 14
Function . . . 14
Fields used as Logical Criteria . . . 15
Processing of Multiple ACCEPT/REJECT Statements . . . 15
Limit Notation . . . 15
Hold Status . . . 15
Example 1 . . . 16
Example 2 . . . 17
ADD . . . 18
ADD . . . 18
Function . . . 18
Operands . . . 18
Result Field - operand2 . . . 19
TO . . . 19
GIVING . . . 19
ROUNDED . . . 19
Related Statement . . . 19
Example . . . 20
ASSIGN . . . 21
ASSIGN . . . 21
AT... . . . 22
AT... . . . 22
AT BREAK . . . 23
AT BREAK . . . 23
Structured Mode Syntax . . . 23
Reporting Mode Syntax . . . 23
Function . . . 23
Reference Notation - r . . . 24
Control Field - operand1 . . . 24

/n/ 24
Example 1 24
Example 2 25
System Functions 26
Multiple Break Levels 27
Example 3 28
AT END OF DATA 30
AT END OF DATA 30
Structured Mode Syntax 30
Reporting Mode Syntax 30
Function 30
Restrictions 30
Reference to a Specific Processing Loop - r 30
Values of Database Fields 31
System Functions 31
Example 32
AT END OF PAGE 33
AT END OF PAGE 33
Structured Mode Syntax 33
Reporting Mode Syntax 33
Function 33
Report Specification - rep 33
Logical Page Size 34
Last-Page Handling 34
System Functions 34
INPUT Statement with AT END OF PAGE 34
Example 1 34
Example 2 36
AT START OF DATA 37
AT START OF DATA 37
Structured Mode Syntax 37
Reporting Mode Syntax 37
Function 37
Value of Database Fields 37
Positioning 37
Reference to a Specific Processing Loop - *r* 38
Example 38
AT TOP OF PAGE 40
AT TOP OF PAGE 40
Structured Mode Syntax 40
Reporting Mode Syntax 40
Function 40
Restriction 40
Report Specification - *rep* 41
Example 41
BACKOUT TRANSACTION 43
BACKOUT TRANSACTION 43
Function 43
Considerations for DL/I Databases 43
Considerations for SQL Databases 43
Backout Transaction Issued by Natural 43
Additional Information 43
Example 44
BEFORE BREAK PROCESSING 46
BEFORE BREAK PROCESSING 46
Structured Mode Syntax 46

 Copyright © Software AG 2002

Reporting Mode Syntax . . . 46
Function . . . 46
Restrictions . . . 46
Example . . . 47
CALL . . . 48
CALL . . . 48
CALL on Mainframe Computers . . . 48
Function . . . 48
Program Name - *operand1* . . . 48
Parameters - *operand2* . . . 49
Return Code . . . 49
Register Usage . . . 50
Boundary Alignment . . . 50
Adabas Calls . . . 51
Direct/Dynamic Loading . . . 51
Example . . . 51
Linkage Conventions . . . 52
Calling a PL/I Program . . . 55
Part I: CALL under OpenVMS, UNIX and Windows . . . 58
Function . . . 58
Name of Called Function - *operand1* . . . 58
Parameters - *operand2* . . . 58
INTERFACE4 . . . 59
INTERFACE4 - External 3GL Program Interface . . . 59
Operand Structure for Interface4 . . . 60
INTERFACE4 - Parameter Access . . . 61
Exported Functions . . . 61
Part II: CALL under OpenVMS, UNIX and Windows . . . 64
Return Code . . . 64
User Exits under Windows . . . 64
User Exits under OpenVMS . . . 64
User Exits under UNIX . . . 65
CALL FILE . . . 69
CALL FILE . . . 69
Structured Mode Syntax . . . 69
Reporting Mode Syntax . . . 69
Function . . . 69
Restriction . . . 69
Control Field - *operand1* . . . 70
Record Area - *operand2* . . . 70
Example . . . 70
Calling Program: . . . 70
Called COBOL Program: . . . 71
CALL LOOP . . . 72
CALL LOOP . . . 72
Structured Mode Syntax . . . 72
Reporting Mode Syntax . . . 72
Function . . . 72
Program Name - operand1 . . . 72
Parameters - operand2 . . . 73
Loop Termination . . . 73
Restriction . . . 73
Example . . . 73
CALLNAT . . . 74
CALLNAT . . . 74
Function . . . 74

Subprogram Name - operand1 74
Parameters - operand2 74
AD= 75
AD=M 76
AD=O 76
AD=A 76
nX 76
Other Considerations 76
Parameter Transfer with Dynamic Variables 77
Example 1 77
Invoking Program: 77
Invoked Subprogram: 78
Example 2 79
Invoking Program: 79
Invoked Subprogram: 79
CLOSE CONVERSATION 80
CLOSE CONVERSATION 80
Function 80
Conversation to be Closed 80
operand1 80
*CONVID 80
ALL 80
Further Information and Examples 80
CLOSE DIALOG 81
CLOSE DIALOG 81
Function 81
Dialog to be Closed 81
operand1 81
*DIALOG-ID 81
Further Information and Examples 81
CLOSE PC 82
CLOSE PC 82
CLOSE PRINTER 83
CLOSE PRINTER 83
Function 83
Printer 83
Example 84
CLOSE WORK FILE 85
CLOSE WORK FILE 85
Function 85
Work File 85
Automatic Closing 85
Example 86
COMPOSE 87
COMPOSE 87
Function 87
Clauses 87
Formatting Process 88
Dialog Mode 88
Dialog Mode for Input 88
Dialog Mode for Output 89
Dialog Mode for Input and Output 89
Execution of COMPOSE Statements in Dialog Mode 89
Non-Natural Programs - only Mainframe 90
RESETTING-clause 91
MOVING-clause 91

 Copyright © Software AG 2002

Syntax 1 . . . 91
Syntax 2 . . . 91
Syntax 3 . . . 92
Syntax 1 . . . 92
Syntax 2 . . . 92
Syntax 3 . . . 92
ASSIGNING-clause . . . 93
FORMATTING-clause . . . 94
OUTPUT Subclause . . . 95
INPUT-subclause . . . 97
STATUS-subclause . . . 98
PROFILE-subclause . . . 99
MESSAGES-subclause . . . 100
ERRORS-subclause . . . 100
ENDING-subclause . . . 101
STARTING-subclause . . . 102
EXTRACTING-clause . . . 102
Example 1 . . . 103
Example 2 . . . 103
Example 3 . . . 103
Example 4 . . . 104
Text Block "XYZ" in "XYLIB": . . . 104
Natural Program: . . . 104
Input Map produced by Program: . . . 104
Resulting Output: . . . 104
Example 5 . . . 104
COMPRESS . . . 108
COMPRESS . . . 108
Function . . . 108
Source Fields - operand1 . . . 109
Target Field - operand2 . . . 109
FULL . . . 109
NUMERIC . . . 110
parameter . . . 110
PM=I . . . 110
DF . . . 110
SUBSTRING . . . 110
WITH DELIMITER - operand7 . . . 111
ALL . . . 111
Processing . . . 111
Example 1 . . . 111
Example 2 . . . 112
Example 3 . . . 113
COMPUTE . . . 114
COMPUTE . . . 114
Structured Mode Syntax . . . 114
Reporting Mode Syntax . . . 114
Function . . . 114
Result Field - operand1 . . . 114
ROUNDED . . . 115
arithmetic-expression . . . 115
Result Precision of a Division . . . 115
SUBSTRING . . . 116
Example 1 . . . 116
Example 2 . . . 116

CREATE OBJECT 118
CREATE OBJECT 118
Function 118
Object Handle - operand1 118
Class-Name - operand2 118
Node - operand3 119
GIVING - operand4 119
DECIDE FOR 120
DECIDE FOR 120
Function 120
FIRST/EVERY 120
WHEN logical-condition 120
WHEN ANY 120
WHEN ALL 120
WHEN NONE 121
Example 1 122
Example 2 123
DECIDE ON 124
DECIDE ON 124
Function 124
FIRST/EVERY 124
Selection Field - operand1 125
VALUES Clause 125
ANY 125
ALL 125
NONE 125
Example 1 125
Example 2 127
DEFINE... 128
DEFINE... 128
DEFINE CLASS 129
DEFINE CLASS 129
Function 129
class-name 129
WITH ACTIVATION POLICY Clause 130
OBJECT Clause 130
LOCAL Clause 130
ID Clause 130
INTERFACE USING Clause 130
copycode 130
DEFINE DATA 132
DEFINE DATA 132
General Syntax 132
Function 132
DEFINE DATA in Structured Mode 132
DEFINE DATA in Reporting Mode 133
data areas 133
global data area 133
parameter data area 133
local data area 133
block 133
.block 133
data-definition 134
parameter-data-definition 135
Example of BY VALUE: 137
Example of BY VALUE for Dialog: 137

 Copyright © Software AG 2002

parameter-handle-definition . . . 137
handle-definition . . . 137
view-definition . . . 138
redefinition . . . 139
variable-definition . . . 141
init-definition . . . 143
array-definition . . . 144
array-init-definition . . . 146
emhdpm . . . 148
AIV-data-definition . . . 150
AIV-definition . . . 151
context-data-definition . . . 152
DEFINE DATA OBJECT . . . 154
Qualifying Data Structures . . . 154
Example 1 . . . 155
Example 2 . . . 155
Example 3 . . . 156
Example 4 . . . 157
Example 5 . . . 157
Example 6 . . . 158
Example 7 . . . 158
DEFINE PRINTER . . . 159
DEFINE PRINTER . . . 159
Function . . . 159
Printer . . . 159
OUTPUT *operand1* . . . 160
Printers under OS/390 with Access Method AM=STD - Standard Batch . . . 160
Printers under VM/CMS with Access Method AM=STD - Standard Batch . . . 164
Printers under BS2000/OSD with Access Method AM=STD - Standard Batch . . . 165
Printers under Com-plete . . . 169
Printers under Natural Advanced Facilities . . . 169
Additional Reports . . . 170
PROFILE/FORMS/NAME/DISP/CLASS/COPIES/PRTY . . . 171
Example 1 . . . 172
Example 2 . . . 172
Example 3 . . . 172
Example 4 . . . 172
DEFINE SUBROUTINE . . . 174
DEFINE SUBROUTINE . . . 174
Function . . . 174
Inline/External Subroutines . . . 174
subroutine-name . . . 174
Subroutine Termination . . . 174
Restrictions . . . 174
Data Available in a Subroutine . . . 175
Inline Subroutines . . . 175
External Subroutines . . . 176
Example 1 . . . 177
Example 2 . . . 178
DEFINE WINDOW . . . 179
DEFINE WINDOW . . . 179
Function . . . 179
Control of Full Screen . . . 180
window-name . . . 180
SIZE . . . 181
BASE . . . 181

REVERSED 182
REVERSED - CD=background-color 182
TITLE operand5 182
CONTROL 182
CONTROL WINDOW 182
CONTROL SCREEN 182
FRAMED 183
FRAMED - CD=frame-color 183
position-clause 183
POSITION SYMBOL 184
POSITION TEXT 184
POSITION OFF 184
Protection of Input Fields in a Window 184
Invoking Different Windows 185
Example 185
DEFINE WORK FILE 186
DEFINE WORK FILE 186
Function 186
Work File Number - *n* 186
Work File Name - *operand1* 186
Work File Name on OpenVMS, UNIX and Windows 186
Work File Name on Mainframe Computers 187
Work File Type - *operand2* 187
Work File Name under OS/390 187
Work File Name under VM/CMS 190
Work File Name under BS2000/OSD 192
Link Name 193
File Name 193
Generic File Name 194
File Name and Link Name 194
Generic File Name and Link Name 194
DELETE 195
DELETE 195
Function 195
Considerations for DL/I Databases 195
Considerations for SQL Databases 195
Considerations for VSAM Databases 195
Statement Reference - r 195
Restriction 195
Hold Status 196
Example 1 196
Example 2 196
DISPLAY 198
DISPLAY 198
Function 198
Report Specification - rep 198
options 198
Page Title/NOTITLE 198
Column Headers/NOHDR 200
GIVE SYSTEM FUNCTIONS 201
statement-parameters 201
Line Advance - Slash 201
output-format 201
Field Positioning Notations 203
Override Column Heading Assignment 203
attributes 204

 Copyright © Software AG 2002

Vertical/Horizontal Display . . . 205
output-element . . . 205
Defaults . . . 207
Example 1 . . . 208
Example 2 . . . 209
Example 3 . . . 210
Example 4 . . . 211
Example 5 . . . 212
Example 6 . . . 213
Example 7 . . . 214
DIVIDE . . . 215
DIVIDE . . . 215
Function . . . 215
Result Field . . . 215
Division by Zero . . . 216
REMAINDER Option . . . 216
Example . . . 217
DO/DOEND . . . 218
DO/DOEND . . . 218
Function . . . 218
Restrictions . . . 218
Example . . . 218
DOWNLOAD . . . 219
DOWNLOAD . . . 219
EJECT . . . 220
EJECT . . . 220
Syntax 1 . . . 220
Function . . . 220
Report Specification - *rep* . . . 220
Syntax 2 . . . 220
Function . . . 221
Report Specification - *rep* . . . 221
IF LESS THAN *operand1* LINES LEFT . . . 221
Processing . . . 221
Example . . . 222
END . . . 224
END . . . 224
Function . . . 224
period - . . . 224
Considerations for Program Execution . . . 224
Examples . . . 224
END TRANSACTION . . . 225
END TRANSACTION . . . 225
Function . . . 225
Databases Affected . . . 225
Storage of Transaction Data - operand1 . . . 226
Considerations for DL/I Databases . . . 226
Considerations for SQL Databases . . . 226
Considerations for VSAM Databases . . . 226
Restriction . . . 226
Example 1 . . . 227
Example 2 . . . 228
ESCAPE . . . 229
ESCAPE . . . 229
Structured Mode Syntax . . . 229
Reporting Mode Syntax . . . 229

Function 229
ESCAPE TOP 229
ESCAPE BOTTOM 229
ESCAPE ROUTINE 230
Additional Considerations 230
Example 231
EXAMINE 233
EXAMINE 233
Function 233
operand1 233
operand4 233
FULL 234
SUBSTRING 234
PATTERN 234
DELIMITERS-option 234
DELETE-REPLACE-clause 235
GIVING-clause 235
GIVING INDEX 236
Example 1 237
Example 2 239
EXAMINE TRANSLATE 240
EXAMINE TRANSLATE 240
Function 240
operand1 240
SUBSTRING 240
INTO UPPER/LOWER CASE 241
Translation Table 241
INVERTED 241
Example 241
EXPAND 243
EXPAND 243
Function 243
operand1 243
operand2 243
FETCH 244
FETCH 244
Function 244
REPEAT 244
RETURN 244
Program Name - operand1 244
Parameters - operand2 245
parameter 245
Additional Considerations 245
Example 246
Invoking Program: 246
Invoked Program: 247
FIND 248
FIND 248
Function 248
Considerations for DL/I Databases 249
Considerations for SQL Databases 249
Considerations for VSAM Databases 249
Entire System Server Restrictions 249
Processing Limit - ALL/operand1 249
FIND FIRST, FIND NUMBER, FIND UNIQUE 250
view-name 250

 Copyright © Software AG 2002

PASSWORD Clause . . . 250
Example of PASSWORD Clause: . . . 250
CIPHER Clause . . . 251
Example of CIPHER Clause: . . . 251
WITH Clause . . . 252
Search Criterion for Adabas Files - basic-search-criterion . . . 252
Search Criterion for VSAM Files - basic-search-criterion . . . 257
Search Criterion for DL/I Files - basic-search-criterion . . . 258
COUPLED-clause . . . 259
Physical Coupling without VIA clause . . . 260
Logical Coupling - VIA clause . . . 260
STARTING WITH ISN=operand5 . . . 261
SORTED BY-clause . . . 261
Example of SORTED BY Clause: . . . 262
RETAIN-clause . . . 263
Set Name - operand6 . . . 263
Releasing Sets . . . 263
Updates by Other Users . . . 264
Restriction . . . 264
Example of a RETAIN Clause: . . . 264
WHERE-clause . . . 264
Example of WHERE Clause: . . . 265
IF NO RECORDS FOUND-clause . . . 265
Structured Mode Syntax . . . 265
Reporting Mode Syntax . . . 266
Database Values . . . 266
Evaluation of System Functions . . . 266
Restriction . . . 266
Example of IF NO RECORDS FOUND Clause: . . . 266
System Variables with the FIND Statement . . . 267
*ISN . . . 268
*NUMBER . . . 269
*COUNTER . . . 269
Example using System Variables: . . . 269
Multiple FIND Statements . . . 269
Example of Multiple FIND Statements: . . . 270
FIND FIRST . . . 271
Restrictions . . . 271
System Variables with FIND FIRST . . . 271
Example of FIND FIRST . . . 271
FIND NUMBER . . . 271
Restrictions . . . 272
System Variables with FIND NUMBER . . . 272
Example of FIND NUMBER: . . . 273
FIND UNIQUE . . . 273
Restrictions . . . 273
Example of FIND UNIQUE . . . 273
FOR . . . 274
FOR . . . 274
Function . . . 274
Control Variable - *operand1* and Initial Setting - *operand2* . . . 274
TO Value - *operand3* . . . 274
STEP Value - *operand4* . . . 275
Consistency Check . . . 275
Example . . . 276

FORMAT 277
FORMAT 277
Function 277
Report Specification - *rep* 277
Parameters 277
Example 279
GET 280
GET 280
Function 280
Restrictions 280
view-name 280
PASSWORD and CIPHER 280
*ISN / operand3 281
Reference to Database Fields - operand4 281
Example 282
GET SAME 283
GET SAME 283
Structured Mode Syntax 283
Reporting Mode Syntax 283
Function 283
Statement Reference - *r* 283
operand1 283
Restrictions 284
Example 284
GET TRANSACTION DATA 286
GET TRANSACTION DATA 286
Function 286
System Variable *ETID 286
Field Specification operand1 286
No Transaction Data Stored 286
Example 287
HISTOGRAM 288
HISTOGRAM 288
Function 288
Restrictions 288
Processing Loop Limit - operand1/ALL 289
view-name 289
PASSWORD Clause 289
SEQUENCE Clause 290
Descriptor - operand4 290
STARTING-ENDING-clause 291
WHERE Clause 291
System Variables 292
Example 293
IF 294
IF 294
Structured Mode Syntax 294
Reporting Mode Syntax 294
Function 294
logical-condition 294
THEN 295
ELSE 295
Example 296
IF SELECTION 298
IF SELECTION 298
Structured Mode Syntax 298

 Copyright © Software AG 2002

Reporting Mode Syntax . . . 298
Function . . . 298
Selection Field - operand1 . . . 299
Example . . . 299
IGNORE . . . 300
IGNORE . . . 300
Function . . . 300
Example . . . 300
INCLUDE . . . 301
INCLUDE . . . 301
Function . . . 301
copycode-name . . . 301
operand1 . . . 301
Example 1 . . . 302
Example 2 . . . 302
Example 3 . . . 303
INPUT . . . 305
INPUT . . . 305
Function . . . 305
Input Modes . . . 305
Screen Mode . . . 305
Non-Screen Modes . . . 305
Entering Data in Response to an INPUT Statement . . . 306
Error Correction . . . 306
Split-Screen Feature . . . 306
Syntax 1 - Dynamic Screen Layout Specification . . . 307
INPUT WINDOW='window-name' . . . 307
NO ERASE . . . 307
statement-parameters . . . 307
WITH TEXT-option . . . 308
MARK-option . . . 310
ALARM-option . . . 311
Default Prompting Text . . . 311
Field Positioning, Text Specification, Attribute Assignment . . . 311
***IN, *OUT and *OUTIN** . . . 313
Field Specification - operand1 . . . 313
parameters . . . 313
Example 1 - Syntax 1 . . . 313
Example 2 - Syntax 1 . . . 314
Example 3 - Syntax 1 . . . 315
Syntax 2 - Using Predefined Map Layout . . . 315
INPUT USING MAP Without Parameter List . . . 316
INPUT Fields Defined in the Program . . . 317
INPUT WINDOW='window-name' . . . 317
WITH TEXT/MARK/ALARM-options . . . 317
USING MAP . . . 317
NO ERASE . . . 317
Field Specification - operand1 . . . 317
Using the INPUT Statement in Non-Screen Modes . . . 318
Processing Data from the Natural Stack . . . 318
Using the INPUT Statement in Batch Mode on Mainframe Computers . . . 319
INTERFACE . . . 321
INTERFACE . . . 321
Function . . . 321
interface-name . . . 321
EXTERNAL . . . 322

ID Clause . . . 322
property-definition . . . 322
property-name . . . 322
format-length/array-definition . . . 323
ID Clause . . . 323
READONLY . . . 323
IS Clause . . . 323
Examples . . . 324
method-definition . . . 325
method-name . . . 327
ID Clause . . . 327
IS Clause . . . 327
PARAMETER Clause . . . 327
LIMIT . . . 328
LIMIT . . . 328
Function . . . 328
Limit Specification - *n* . . . 328
Record Counting . . . 328
Example 1 . . . 329
Example 2 . . . 330
LOOP . . . 331
LOOP . . . 331
Function . . . 331
Statement Reference Notation - *r* . . . 331
Database Variable References . . . 331
Restrictions . . . 331
Example 1 . . . 331
Example 2 . . . 332
METHOD . . . 333
METHOD . . . 333
Function . . . 333
Example . . . 333
MOVE . . . 337
MOVE . . . 337
Function . . . 338
ROUNDED . . . 339
parameter . . . 339
PM=I . . . 339
DF . . . 339
SUBSTRING . . . 339
MOVE BY NAME . . . 340
MOVE BY NAME with Arrays . . . 340
MOVE BY POSITION . . . 342
MOVE EDITED . . . 342
MOVE LEFT/RIGHT JUSTIFIED . . . 342
MOVE LEFT/RIGHT JUSTIFIED with PM=I . . . 343
Other Considerations . . . 343
Example 1 . . . 344
Example 2 . . . 345
MOVE ALL . . . 346
MOVE ALL . . . 346
Function . . . 346
Source Operand - operand1 . . . 346
Target Operand - operand2 . . . 346
UNTIL Option - operand3 . . . 346
Example . . . 347

 Copyright © Software AG 2002

MULTIPLY . . . 348
MULTIPLY . . . 348
Syntax 1 . . . 348
Syntax 2 . . . 348
Function . . . 348
Result Field . . . 348
Example . . . 349
NEWPAGE . . . 350
NEWPAGE . . . 350
Function . . . 350
Report Specification - rep . . . 350
EVEN IF TOP OF PAGE . . . 350
WHEN LESS THAN operand1 LINES LEFT . . . 351
WITH TITLE . . . 351
Example . . . 351
NOTITLE... . . . 354
NOTITLE... . . . 354
OBTAIN . . . 355
OBTAIN . . . 355
Function . . . 355
operand1 . . . 355
Examples . . . 355
ON ERROR . . . 356
ON ERROR . . . 356
Structured Mode Syntax . . . 356
Reporting Mode Syntax . . . 356
Function . . . 356
ON ERROR Processing within Subroutines . . . 356
Restriction . . . 356
System Variables *ERROR-NR and *ERROR-LINE . . . 357
Exiting from an ON ERROR Block . . . 357
Example . . . 357
OPEN CONVERSATION . . . 358
OPEN CONVERSATION . . . 358
Function . . . 358
Subprogram Names - operand1 . . . 358
Further Information and Examples . . . 358
OPEN DIALOG . . . 359
OPEN DIALOG . . . 359
Function . . . 359
Dialog Name - operand1 . . . 359
Handle Name - operand2 . . . 359
Dialog ID - operand3 . . . 360
AD= . . . 360
AD=M . . . 360
AD=O . . . 360
AD=A . . . 360
Passing Parameters to the Dialog . . . 360
*n*X . . . 361
PARAMETERS-clause . . . 361
Further Information and Examples . . . 361
OPTIONS . . . 362
OPTIONS . . . 362
PASSW . . . 363
PASSW . . . 363
Function . . . 363

Password - operand1 . . . 363
Natural Security Considerations . . . 364
Restriction . . . 364
Password Display Protection - Mainframe only . . . 364
Example . . . 364
PERFORM . . . 366
PERFORM . . . 366
Function . . . 366
subroutine-name . . . 366
Data Available in a Subroutine . . . 366
Inline Subroutines . . . 366
External Subroutines . . . 366
Parameters - operand2 . . . 367
AD= . . . 367
AD=M . . . 368
AD=O . . . 368
AD=A . . . 368
*n***X** . . . 368
Nested PERFORM Statements . . . 368
Parameter Transfer with Dynamic Variables . . . 368
Example 1 . . . 369
Example 2 . . . 370
PERFORM BREAK PROCESSING . . . 372
PERFORM BREAK PROCESSING . . . 372
Function . . . 372
Statement Reference Notation - *r* . . . 372
AT BREAK *statement...* . . . 372
Example . . . 373
PRINT . . . 374
PRINT . . . 374
Function . . . 374
Report Specification - rep . . . 374
NOTITLE . . . 374
NOHDR . . . 375
statement-parameters . . . 375
Field Positioning, Text, Attribute Assignment . . . 375
Field Positioning Notations . . . 376
Text/Attribute Assignment . . . 376
operand1 . . . 378
parameters . . . 378
Example . . . 379
PROCESS . . . 380
PROCESS . . . 380
Function . . . 380
USING . . . 380
GIVING . . . 380
PROCESS COMMAND . . . 381
PROCESS COMMAND . . . 381
Structured Mode Syntax . . . 381
Reporting Mode Syntax . . . 382
Function . . . 382
CLOSE . . . 383
CHECK . . . 383
EXEC . . . 383
HELP . . . 383
HELP for Keywords . . . 384

 Copyright © Software AG 2002

HELP for Synonyms . . . 384
HELP for Global Functions . . . 385
HELP for Local Functions . . . 386
HELP for IKN . . . 386
HELP for IFN . . . 387
TEXT . . . 388
TEXT for General Information . . . 388
TEXT for Keyword Information . . . 388
TEXT for Function Information . . . 389
GET . . . 389
SET . . . 389
USING Clause . . . 391
GIVING Clause . . . 392
DDM "COMMAND" . . . 392
Security Considerations . . . 393
Example 1 . . . 393
Example 2 . . . 394
PROCESS GUI . . . 395
PROCESS GUI . . . 395
Function . . . 395
action-name . . . 395
Passing Parameters to the Action . . . 395
PARAMETERS-clause . . . 395
nX . . . 396
GIVING operand2 . . . 396
PROCESS REPORTER . . . 397
PROCESS REPORTER . . . 397
Function . . . 397
Actions . . . 398
WITH Clause . . . 398
PARAMETERS-clause . . . 398
Parameters for OPEN Action . . . 398
Parameters for REPLACE-TABLE Action . . . 399
Parameter for SET-PRINTER Action . . . 399
Parameters for SET-PRINT-OPTIONS Action . . . 399
Parameter for CLOSE, PRINT, PREVIEW, EDIT Actions . . . 401
GIVING operand2 . . . 401
PROPERTY . . . 402
PROPERTY . . . 402
Function . . . 402
READ . . . 403
READ . . . 403
Function . . . 403
Number of Records - operand1/ALL . . . 403
view-name . . . 404
PASSWORD and CIPHER Clauses . . . 404
WITH REPOSITION . . . 404
sequence/range-specification . . . 405
READ IN PHYSICAL SEQUENCE . . . 406
READ BY ISN . . . 406
READ IN LOGICAL SEQUENCE . . . 406
ASCENDING/DESCENDING/VARIABLE SEQUENCE . . . 406
STARTING FROM / ENDING AT . . . 407
STARTING WITH ISN=operand4 . . . 408
Access to Adabas . . . 408
Access to VSAM . . . 408

Examples . . . 408
WHERE Clause . . . 409
System Variables . . . 409
Example 1 . . . 410
Example 2 - Combining READ with FIND . . . 412
READ WORK FILE . . . 413
READ WORK FILE . . . 413
Structured Mode Syntax . . . 413
Reporting Mode Syntax . . . 413
Function . . . 414
work-file-number . . . 414
ONCE Option . . . 414
Variable Index Range . . . 414
RECORD Option . . . 414
SELECT Option - default . . . 415
Field Lengths . . . 415
GIVING LENGTH operand3 . . . 416
AT END OF FILE . . . 416
Handling of Large and Dynamic Variables . . . 416
Example . . . 417
REDEFINE . . . 418
REDEFINE . . . 418
Function . . . 418
Method of Redefinition . . . 418
Further Redefinition . . . 418
Filler Notation . . . 418
Example 1 . . . 419
Example 2 . . . 419
Example 3 . . . 419
Example 4 . . . 420
REDUCE . . . 421
REDUCE . . . 421
Function . . . 421
operand1 . . . 421
operand2 . . . 421
REINPUT . . . 422
REINPUT . . . 422
Function . . . 422
REINPUT FULL . . . 422
statement-parameters . . . 423
USING HELP . . . 423
WITH TEXT-option . . . 423
Message Text from Natural Message File - *operand1 . . . 424
Message Text - operand2 and Attributes - attributes . . . 424
Dynamic Replacement of Message Text - operand3 . . . 424
MARK-option . . . 425
Field to be Marked - operand5 . . . 425
MARK POSITION . . . 426
attributes . . . 426
ALARM-option . . . 426
Example 1 . . . 426
Example 2 . . . 428
Example 3 . . . 428
REJECT . . . 430
REJECT . . . 430

 Copyright © Software AG 2002

RELEASE 431
RELEASE 431
Function 431
RELEASE STACK 431
RELEASE SET 431
RELEASE VARIABLES 431
Example 432
REPEAT 433
REPEAT 433
Syntax 1 433
Syntax 2 433
Function 433
UNTIL 434
WHILE 434
Example 1 434
Example 2: 434
REQUEST DOCUMENT 436
REQUEST DOCUMENT 436
Function 437
operand1 438
operand2 438
operand3 438
operand4/5 438
Header Name for Operand4 438
Header Value for Operand5 438
General Information 438
Automatically Generated Headers (operand 4/5) 438
operand6 439
operand7/8 439
operand9 441
operand10/11 441
operand12 441
operand13 442
Overview of Response Numbers - for HTTP/HTTPs Requests 442
operand14 444
Examples 444
General Request 444
Simple Get Request (no data) 444
Simple Head Request (no return page) 444
Simple Post Request (default) 445
Simple Put Request (with data all) 445
RESET 446
RESET 446
Function 446
INITIAL 446
Default Initial Values 446
Example 447
RETRY 448
RETRY 448
Function 448
Example 449
RUN 450
RUN 450
Function 450
REPEAT 450
Program Name - operand1 450

Parameters - operand2 . . . 450
parameter . . . 451
Dynamic Source Text Creation/Execution . . . 451
Example . . . 452
Program containing RUN statement: . . . 452
Program FIND-EMP executed by RUN statement: . . . 452
SEND EVENT . . . 453
SEND EVENT . . . 453
Function . . . 453
Operands . . . 453
AD= . . . 453
AD=M . . . 454
AD=O . . . 454
AD=A . . . 454
Passing Parameters to the Dialog . . . 454
nX . . . 454
PARAMETERS-clause . . . 454
Further Information and Examples . . . 455
SEND METHOD . . . 456
SEND METHOD . . . 456
Function . . . 456
Method-Name - operand1 . . . 457
Object Handle - operand2 . . . 457
Parameter - operand3 . . . 457
AD= . . . 457
AD=M . . . 458
AD=O . . . 458
AD=A . . . 458
Parameter - nX . . . 458
RETURN - operand4 . . . 459
GIVING - operand5 . . . 459
SEPARATE . . . 460
SEPARATE . . . 460
Function . . . 460
Source Operand - operand1 . . . 460
SUBSTRING . . . 461
LEFT JUSTIFIED . . . 461
Target Operand - operand4 . . . 461
IGNORE / REMAINDER . . . 461
DELIMITER Option . . . 461
WITH RETAINED DELIMITERS . . . 462
GIVING NUMBER . . . 462
Example 1 . . . 463
Example 2 . . . 464
Example 3 . . . 465
SET CONTROL . . . 466
SET CONTROL . . . 466
Function . . . 466
Example 1 . . . 466
Example 2 . . . 466
SET GLOBALS . . . 467
SET GLOBALS . . . 467
Function . . . 467
Parameters . . . 467
Example . . . 467

 Copyright © Software AG 2002

SET KEY 468
SET KEY 468
Syntax 1 - Affecting All Keys 468
Syntax 2 - Affecting Individual Keys 468
Syntax 3 - Affecting Individual Keys 469
Function 469
Making Keys Program-Sensitive 469
Assigning Commands/Programs 470
Assigning Input DATA 471
COMMAND OFF/ON 471
Assigning HELP 472
DYNAMIC 472
DISABLED 472
SET KEY Statements on Different Program Levels 473
Example of SET KEY Statements on Different Program Levels: 473
Assigning Names 474
Example 476
SET TIME 477
SET TIME 477
Function 477
Example 477
SET WINDOW 478
SET WINDOW 478
Function 478
Example 478
SKIP 479
SKIP 479
Function 479
Report Specification - rep 479
Number of Lines to be Skipped - operand1 479
Additional Considerations 479
Example 480
SORT 481
SORT 481
Structured Mode Syntax 481
Reporting Mode Syntax 481
Function 481
Restrictions 482
Processing Loops 482
Sort Criteria - operand1 482
USING-clause 482
GIVE-clause 483
(NL=nn) 483
SORT Statement Processing 484
1st Phase - Selecting the Records to be Sorted 484
2nd Phase - Sorting the Records 484
3rd Phase -Processing the Sorted Records 484
Example 484
First Phase: 485
Second Phase: 486
Third Phase: 486
STACK 487
STACK 487
Function 487
TOP 487
DATA 488

FORMATTED 488
COMMAND operand1 490
COMMAND operand1 operand2... 490
parameter 490
Example 491
STOP 492
STOP 492
Function 492
Example 492
STORE 493
STORE 493
Structured Mode Syntax 493
Reporting Mode Syntax 493
Function 494
view-name 494
PASSWORD/CIPHER 494
USING/GIVING NUMBER 494
SET/WITH 494
DL/I Considerations 495
USING SAME 495
System Variable *ISN 495
Example 496
SUBTRACT 498
SUBTRACT 498
Syntax 1 498
Syntax 2 498
Function 498
Operands 498
Result Field 499
ROUNDED 499
Example 499
SUSPEND IDENTICAL SUPPRESS 500
SUSPEND IDENTICAL SUPPRESS 500
Function 500
Report Specification - rep 500
Example 500
Program with SUSPEND IDENTICAL SUPPRESS: 500
Same as Previous Program, but without SUSPEND IDENTICAL SUPPRESS: 501
TERMINATE 503
TERMINATE 503
Function 503
operand1 503
UNIX 503
Windows 503
Mainframes 503
operand2 503
Program Receiving Control after Termination 504
Example 504
UPDATE 505
UPDATE 505
Structured Mode Syntax 505
Reporting Mode Syntax 505
Function 505
Considerations for DL/I Databases 505
Considerations for SQL Databases 506
Considerations for VSAM Databases 506

 Copyright © Software AG 2002

Restrictions 506
Statement Reference - r 506
USING SAME 506
SET/WITH operand1 = operand2 507
Hold Status 507
Example 508
UPLOAD 509
UPLOAD 509
WRITE 510
WRITE 510
Syntax 1 - Dynamic Formatting 510
Function 510
Report Specification - rep 511
NOTITLE 511
NOHDR 511
statement-parameters 511
Output Format 512
Field Positioning Notations 512
Text/Attribute Assignment 513
Syntax 2 - Using Predefined Map 514
FORM/MAP 515
operand1 515
operand2 515
NOTITLE/NOHDR 515
Example 1 515
Example 2 515
Example 3 517
Example 4 517
Example 5 518
WRITE TITLE 519
WRITE TITLE 519
Function 519
Restrictions 519
Report Specification - rep 519
Justification and Underlining 520
statement-parameters 520
operand1 520
SKIP - operand2 520
Example 521
WRITE TRAILER 522
WRITE TRAILER 522
Function 522
Restrictions 522
Processing 522
Logical Page Size 523
Report Specification - rep 523
Justification and Underlining 523
statement-parameters 523
operand1 523
SKIP - operand2 523
Example 524
WRITE WORK FILE 525
WRITE WORK FILE 525
Function 525
work-file-number 525
VARIABLE 525

Fields - operand1 . . . 525
Variable Index Range . . . 526
External Representation of Fields . . . 526
Handling of large and dynamic variables . . . 526
Example . . . 527
Old Statements . . . 528
Old Statements . . . 528
SQL Statements Overview . . . 529
SQL Statements Overview . . . 529
Common Set and Extended Set . . . 530
Common Set and Extended Set . . . 530
Basic Syntactical Items . . . 531
Basic Syntactical Items . . . 531
Constants . . . 531
constant . . . 531
integer . . . 531
Names . . . 532
authorization-identifier . . . 532
ddm-name . . . 532
view-name . . . 532
column-name . . . 532
table-name . . . 532
correlation-name . . . 533
Parameters . . . 535
parameter . . . 535
Natural View Concept . . . 540
Natural View Concept . . . 540
Scalar Expressions . . . 541
Scalar Expressions . . . 541
scalar-operator . . . 541
factor . . . 541
atom . . . 542
column-reference . . . 542
aggregate-function . . . 542
special-register . . . 543
scalar-function . . . 543
units . . . 544
case-expression . . . 545
Search Conditions . . . 546
Search Conditions . . . 546
search-condition . . . 546
predicate . . . 547
Comparison Predicate . . . 548
BETWEEN Predicate . . . 549
LIKE Predicate . . . 549
NULL Predicate . . . 549
IN Predicate . . . 550
Quantified Predicate . . . 550
EXISTS Predicate . . . 550
Select Expressions . . . 552
Select Expressions . . . 552
select-expression . . . 552
selection . . . 552
table-expression . . . 553
Flexible SQL . . . 557
Flexible SQL . . . 557

 Copyright © Software AG 2002

Text Variables 558
LINDICATOR Option 558
CALLDBPROC 560
CALLDBPROC 560
Function 560
dbproc 560
ddm-name 561
parameter 561
AD= 561
result-set 561
GIVING sqlcode 562
CALLMODE 562
Example 562
COMMIT 564
COMMIT 564
Function 564
Consideration for Non-Natural-Programs 564
DELETE 565
DELETE 565
Syntax 1 - Searched DELETE 565
Syntax 2 - Positioned DELETE 565
Function 565
FROM Clause 565
WHERE Clause 565
Statement Reference - r 566
INSERT 567
INSERT 567
Function 567
INTO Clause 567
column-list 567
VALUES Clause 567
VALUES Clause with Preceding Asterisk Notation 567
VALUES Clause with Preceding Column List 568
select-expression 568
PROCESS SQL 570
PROCESS SQL 570
Function 570
ddm-name 570
statement-string 570
Parameters 570
:U:host-variable 570
:G:host-variable 571
Examples 571
Examples for DB2 (under OS/390): 571
Example for Adabas D: 571
Example of Calling a Procedure Stored in Adabas D: 571
ENTIRE ACCESS Options 571
READ RESULT SET 572
READ RESULT SET 572
Function 572
limit 572
ddm-name 572
GIVING sqlcode 572
Example 573
ROLLBACK 574
ROLLBACK 574

Function 574
Consideration for Non-Natural Programs 574
SELECT 575
SELECT 575
Cursor-Oriented Selection 575
Non-Cursor Selection 575
table-expression 576
INTO Clause 578
parameter 579
VIEW Clause 579
correlation-name 580
Query involving UNION 580
ORDER BY Clause 582
IF NO RECORDS FOUND-clause 584
Structured Mode Syntax 584
Reporting Mode Syntax 584
Database Values 584
Evaluation of System Functions 585
Join Queries 586
UPDATE 587
UPDATE 587
Syntax 1 - Searched UPDATE 587
Syntax 2 - Positioned UPDATE 587
Function 587
SET Clause 587
assignment-list 588
WHERE search-condition 588
Statement Reference - r 588
Examples 589
Example of Searched UPDATE: 589
Example of Searched UPDATE with *assignment-list*: 589
Example of Positioned UPDATE: 590
Example of Positioned UPDATE with *assignment-list*: 590

 Copyright © Software AG 2002

Statements - Overview

This section describes the Natural programming language statements.

- Syntax Symbols and Operand Definition Tables
- Statements Grouped by Functions
- Statements in Alphabetical Order

ACCEPT/REJECT	ADD	ASSIGN	AT BREAK
AT END OF DATA	AT END OF PAGE	AT START OF DATA	AT TOP OF PAGE
BACKOUT TRANSACTION	BEFORE BREAK PROCESSING	CALL	CALL FILE
CALL LOOP	CALLNAT	CLOSE CONVERSATION	CLOSE DIALOG
CLOSE PC	CLOSE PRINTER	CLOSE WORK FILE	COMPOSE
COMPRESS	COMPUTE	CREATE OBJECT	DECIDE FOR
DECIDE ON	DEFINE CLASS	DEFINE DATA	DEFINE PRINTER
DEFINE SUBROUTINE	DEFINE WINDOW	DEFINE WORK FILE	DELETE
DISPLAY	DIVIDE	DO/DOEND	DOWNLOAD
EJECT	END	END TRANSACTION	ESCAPE
EXAMINE	EXAMINE TRANSLATE	EXPAND	FETCH
FIND	FOR	FORMAT	GET
GET SAME	GET TRANSACTION DATA	HISTOGRAM	IF
IF SELECTION	IGNORE	INCLUDE	INPUT
INTERFACE	LIMIT	LOOP	METHOD
MOVE	MOVE ALL	MULTIPLY	NEWPAGE
OBTAIN	ON ERROR	OPEN CONVERSATION	OPEN DIALOG
OPTIONS	PASSW	PERFORM	PERFORM BREAK PROCESSING
PRINT	PROCESS	PROCESS COMMAND	PROCESS GUI
PROCESS REPORTER	PROPERTY	READ	READ WORK FILE
REDEFINE	REDUCE	REINPUT	REJECT
RELEASE	REPEAT	REQUEST DOCUMENT	RESET
RETRY	RUN	SEND EVENT	SEND METHOD
SEPARATE	SET CONTROL	SET GLOBALS	SET KEY
SET TIME	SET WINDOW	SKIP	SORT
STACK	STOP	STORE	SUBTRACT
SUSPEND IDENTICAL SUPPRESS	TERMINATE	UPDATE	UPLOAD
WRITE	WRITE TITLE	WRITE TRAILER	WRITE WORK FILE

- Natural SQL Statements
  CALLDBPROC | COMMIT | DELETE | INSERT | PROCESS SQL | READ RESULT SET | ROLLBACK | SELECT | UPDATE

  With these statements, you can use SQL directly in Natural programs.

Copyright © Software AG 2002

Example Programs

Generally, the example programs shown in these subsections are written in structured mode. For statements where the reporting-mode syntax differs considerably from the structured-mode syntax, references to equivalent reporting-mode examples are also provided.

The example programs shown in this section are also available online in the Natural library "SYSEXRM".

 Copyright © Software AG 2002

Syntax Symbols and Operand Definition Tables

This section covers the following topics:

- Syntax Symbols
- Operand Definition Table

Syntax Symbols

The following symbols are used within the diagrams that describe the syntax of Natural statements:

ABCDEF	Upper-case letters indicate that the term is either a Natural keyword or a Natural reserved word that must be entered exactly as specified.
<u>ABCDEF</u>	If an optional term in upper-case letters is completely underlined (not a hyperlink!), this indicates that the term is the default value. If you omit the term, the underlined value applies.
<u>ABC</u>DEF	If a term in upper-case letters is partially underlined (not a hyperlink!), this indicates that the underlined portion is an acceptable abbreviation of the term.
abcdef	Letters in italics are used to represent variable information. You must supply a valid value when specifying this term.
[ ]	Elements contained within square brackets are optional. If the square brackets contain several lines stacked one above the other, each line is an optional alternative. You may choose at most one of the alternatives.
{ }	If the braces contain several lines stacked one above the other, each line is an alternative. You must choose exactly one of the alternatives.
|	The vertical bar separates alternatives.
...	A term preceding an ellipsis may optionally be repeated. A number after the ellipsis indicates how many times the term may be repeated. If the term preceding the ellipsis is an expression enclosed in square brackets or braces, the ellipsis applies to the entire bracketed expression.
,...	A term preceding a comma-ellipsis may optionally be repeated; if it is repeated, the repetitions must be separated by commas. A number after the comma-ellipsis indicates how many times the term may be repeated. If the term preceding the comma-ellipsis is an expression enclosed in square brackets or braces, the comma-ellipsis applies to the entire bracketed expression.
:...	A term preceding a colon-ellipsis may optionally be repeated; if it is repeated, the repetitions must be separated by colons. A number after the colon-ellipsis indicates how many times the term may be repeated. If the term preceding the colon-ellipsis is an expression enclosed in square brackets or braces, the colon-ellipsis applies to the entire bracketed expression.
Other symbols (except [ ] { } | ... ,... :...)	All other symbols except those defined in this table must be entered exactly as specified. *Exception*: The SQL scalar concatenation operator is represented by two vertical bars that must be entered literally as they appear in the syntax definition.

Example:

```
WRITE [USING] {FORM} operand1 [operand2...]
              {MAP}
```

- WRITE, USING, MAP and FORM are Natural keywords which you must enter as specified.
- *operand1* and *operand2* are user-supplied variables for which you specify the names of the objects you wish to deal with.
- The braces indicate that you must choose whether to specify either `FORM` or `MAP`; however, you must specify one of the two.

 Copyright © Software AG 2002

- The square brackets indicate that USING and *operand2* are optional elements which you can, but need not, specify.
- The ellipsis indicates that you may specify *operand2* several times.

Operand Definition Table

Whenever one or more operands appear in the syntax of a Natural statement, the following table is provided:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S	A	G	N/M	A	N	P	I	F	B	D	T	L	C	G	O	yes/no	yes/no

This table provides the following information on each operand:

Possible Structure

Indicates the structure which the operand may take:

C	Constant.
S	Single occurrence (scalar; that is, a field/variable which is neither an array range nor a group).
A	Array.
G	Group.
N/M	Natural system variable: **N** = **All** system variables can be used. **M** = Only **modifiable** system variables can be used. For information on which system variables are modifiable, see the section System Variables in the Natural Reference documentation.

Possible Formats

Indicates the format which the operand may take:

A	Alphanumeric
N	Numeric unpacked
P	Packed numeric
I	Integer
F	Floating point
B	Binary
D	Date
T	Time
L	Logical
C	Attribute control
G	GUI handle
O	Object handle

Reference Permitted

Indicates whether the operand may be referenced via a statement label or the source code line number.

Dynamic Definition

Indicates whether the field may be dynamically defined within the body of the program. This is possible in reporting mode only.

 Copyright © Software AG 2002

Statements Grouped by Functions

This section provides an overview of the statements grouped by their functions.

This section covers the following topics:

- Database Access and Update
- Arithmetic and Data Movement Operations
- Loop Execution
- Creation of Output Reports
- Screen Generation for Interactive Processing
- Processing of Logical Conditions
- Invoking Programs and Routines
- Control of Work Files
- Component Based Programming
- Event-Driven Programming
- Miscellaneous

Copyright © Software AG 2002

Database Access and Update

The following statements are used to access and manipulate information contained in a database.

READ	Reads a database file in physical or logical sequence of records.
FIND	Selects records from a database file based on user-specified criteria.
HISTOGRAM	Reads the values of a database field.
GET	Reads a record with a given ISN (internal sequence number) or RNO (record number).
GET SAME	Re-reads the record currently being processed.
ACCEPT/REJECT	Accepts/reject records based on user-specified criteria.
PASSW	Provides password for access to a password-protected file.
LIMIT	Limits the number of executions of a READ, FIND or HISTOGRAM processing loop.
STORE	Adds a new record to the database.
UPDATE	Updates a record in the database.
DELETE	Deletes a record from the database.
END TRANSACTION	Indicates the end of a logical transaction.
BACKOUT TRANSACTION	Backs out a partially completed logical transaction.
GET TRANSACTION DATA	Reads transaction data stored with a previous END TRANSACTION statement.
RETRY	Attempts to re-read a record which is in hold status for another user.
AT START OF DATA	Specifies statements to be performed when the first of a set of records is processed in a processing loop.
AT END OF DATA	Specifies statements to be performed after the last of a set of records has been processed in a processing loop.
AT BREAK	Specifies statements to be performed when the value of a control field changes (break processing).
BEFORE BREAK PROCESSING	Specifies statements to be performed before performing break processing.
PERFORM BREAK PROCESSING	Immediately invokes break processing.

 Copyright © Software AG 2002

Arithmetic and Data Movement Operations

The following statements are used for arithmetic and data movement operations:

COMPUTE	Performs arithmetic operations or assigns values to fields.
ADD	Adds two or more operands.
SUBTRACT	Subtracts one or more operands from another operand.
MULTIPLY	Multiplies two or more operands.
DIVIDE	Divides one operand into another.
MOVE	Moves the value of an operand to one or more fields.
MOVE ALL	Moves multiple occurrences of a value to another field.
COMPRESS	Concatenates the value of two or more fields into a single field.
SEPARATE	Separates the content of a field into two or more fields.
EXAMINE	Scans a field for a specific value and replaces it, and/or counts how often it occurs.
RESET	Sets the value of a field to zero (if numeric) or blank (if alphanumeric), or to its initial value.

Loop Execution

The following statements are related to the execution of processing loops:

REPEAT	Initiates a processing loop (and terminates it based on a specified condition).
FOR	Initiates a processing loop and controls the number of times the loop is to be processed.
ESCAPE	Stops the execution of a processing loop.

Copyright © Software AG 2002

Creation of Output Reports

The following statements are used for the creation of output reports:

FORMAT	Specifies output parameter settings.
DISPLAY	Specifies fields to be output in column form.
WRITE/PRINT	Specifies fields to be output in non-column form.
WRITE TITLE	Specifies text to be output at the top of each page of a report.
WRITE TRAILER	Specifies text to be output at the bottom of each page of a report.
AT TOP OF PAGE	Specifies processing to be performed when a new output page is started.
AT END OF PAGE	Specifies processing to be performed when the end of an output page is reached.
SKIP	Generates one or more blank lines in a report.
EJECT	Causes a page advance without titles or headings.
NEWPAGE	Causes a page advance with titles and headings.
SUSPEND IDENTICAL SUPPRESS	Suspends identical suppression for a single record.
DEFINE PRINTER	Allocates a report to a logical output destination.
CLOSE PRINTER	Closes a printer.

Screen Generation for Interactive Processing

The following statements are used to create data screens (maps) for the purpose of interactive processing of data:

INPUT	Creates a formatted screen (map) for data display/ entry.
REINPUT	Re-executes an INPUT statement (if invalid data were entered in response to the previous INPUT statement).
DEFINE WINDOW	Specifies the size, position and attributes of a window.
SET WINDOW	Activates and de-activates a window.

 Copyright © Software AG 2002

Processing of Logical Conditions

The following statements are used to control the execution of statements based on conditions detected during the execution of a Natural program:

IF	Performs statements depending on a logical condition.
IF SELECTION	Verifies that in a sequence of alphanumeric fields one and only one contains a value.
DECIDE FOR	Performs statements depending on logical conditions.
DECIDE ON	Performs statements depending on the contents of a variable.
ON ERROR	Intercepts runtime errors which would otherwise result in a Natural error message, followed by the termination of the Natural program.

Invoking Programs and Routines

The following statements are used in conjunction with the execution of programs and routines:

FETCH	Invokes a Natural program.
CALLNAT	Invokes a Natural subprogram.
PERFORM	Invokes a Natural subroutine.
DEFINE SUBROUTINE	Defines a Natural subroutine.
ESCAPE	Stops the execution of a routine.
CALL	Invokes a non-Natural program from a Natural program.
CALL FILE	Invokes a non-Natural program to read a record from a non-Adabas file.
CALL LOOP	Generates a processing loop containing a call to a non-Natural program.

Control of Work Files

The following Natural statements are used to read/write data to a physical sequential (non-Adabas) work file:

WRITE WORK FILE	Writes data to a work file.
READ WORK FILE	Reads data from a work file.
CLOSE WORK FILE	Closes a work file.
DEFINE WORK FILE	Assigns a file name to a work file.

Copyright © Software AG 2002

Component Based Programming

The following Natural statements are used in conjunction with component based programming:

DEFINE CLASS	Specifies a class from within a Natural class module.
CREATE OBJECT	Creates an object (also known as an instance) of a given class.
SEND METHOD	Invokes a method of an object.
INTERFACE	Defines an interface (a collection of methods and properties) for a certain feature of a class.
METHOD	Assigns a subprogram as the implementation of a method, outside an interface definition.
PROPERTY	Assigns an object data variable as the implementation to a property, outside an interface definition.

Event-Driven Programming

The following Natural statements are used for event-driven programming:

OPEN DIALOG	Opens a dialog.
CLOSE DIALOG	Closes a dialog.
SEND EVENT	Triggers a user-defined event.
PROCESS GUI	Performs a standard procedure in an event-driven application.

 Copyright © Software AG 2002

Miscellaneous

DEFINE DATA	Defines the data elements which are to be used in a Natural program or routine.
END	Indicates the end of the source code of a Natural program or routine.
EXPAND	Expands the allocated memory of dynamic variables to a given size.
EXAMINE TRANSLATE	Translates the characters contained in a field into upper-case or lower-case, or into other characters.
INCLUDE	Incorporates Natural copycode at compilation.
PROCESS COMMAND	Invokes a command processor.
REDUCE	Reduces the allocated memory of dynamic variables.
RELEASE	Deletes the contents of the Natural stack; releases sets of ISNs/RNOs retained via a FIND statement; releases Natural global variables.
REQUEST DOCUMENT	Allows you to access an external system.
RUN	Compiles and executes a source program.
SET CONTROL	Performs a Natural terminal command from within a Natural program.
SET KEY	Assigns functions to terminal keys.
SETTIME	Establishes a point-in-time reference for a *TIMD system variable.
SORT	Sorts records, using the sort program provided by the operating system.
STACK	Places data and/or commands into the Natural stack.
STOP	Terminates the execution of an application.
TERMINATE	Terminates the Natural session.

Copyright © Software AG 2002

ACCEPT/REJECT

```
{ACCEPT} [IF] logical-condition
{REJECT}
```

Function

The statements ACCEPT and REJECT are used for accepting/rejecting a record based on user-specified logical criterion. The ACCEPT/REJECT statement may be used in conjunction with statements which read data records in a processing loop (FIND, READ, HISTOGRAM, CALL FILE, SORT or READ WORK FILE). The criterion is evaluated **after** the record has been selected/read.

Whenever an ACCEPT/REJECT statement is encountered for processing, it will internally refer to the innermost currently active processing loop initiated with one of the above mentioned statements.

When ACCEPT/REJECT statements are placed in a subroutine, in case of a record reject, the subroutine(s) entered in the processing loop will automatically be terminated and processing will continue with the next record of the innermost currently active processing loop.

 Copyright © Software AG 2002

Fields used as Logical Criteria

The fields used to specify the logical criterion may be database fields or user-defined variables. For additional information on logical conditions, see the Natural Reference documentation.

When ACCEPT/REJECT is used with a HISTOGRAM statement, only the database field specified in the HISTOGRAM statement may be used as a logical criterion.

Processing of Multiple ACCEPT/REJECT Statements

Normally, only one ACCEPT or REJECT statement is required in a single processing loop. If more than one ACCEPT/REJECT is specified **consecutively**, the following conditions apply:

- If consecutive ACCEPT and REJECT statements are contained in the same processing loop, they are processed in the specified order.
- If an ACCEPT condition is satisfied, the record will be accepted and consecutive ACCEPT/REJECT statements will be ignored.
- If a REJECT condition is satisfied, the record will be rejected and consecutive ACCEPT/REJECT statements will be ignored.
- If the processing continues to the last ACCEPT/REJECT statement, the last statement will determine whether the record is accepted or rejected.

If other statements are interleaved between multiple ACCEPT/REJECT statements, each ACCEPT/REJECT will be handled independently.

Limit Notation

If a LIMIT statement or other limit notation has been specified for a processing loop containing an ACCEPT or REJECT statement, each record processed is counted against the limit regardless of whether or not the record is accepted or rejected.

Hold Status

ACCEPT/REJECT processing does not cause a held record to be released from hold status unless the profile parameter RI has been set to RI = ON (this parameter is only available on mainframe computers; see also the Natural Operations for Mainframes documentation).

Copyright © Software AG 2002

Example 1

```
/* EXAMPLE 'ACREX1S': ACCEPT (STRUCTURED MODE)
DEFINE DATA LOCAL
 1 EMPLOY-VIEW VIEW OF EMPLOYEES
   2 NAME
   2 SEX
   2 MAR-STAT
END-DEFINE
LIMIT 50
READ EMPLOY-VIEW
  ACCEPT IF SEX='M' AND MAR-STAT = 'S'
  WRITE  NOTITLE '=' NAME '=' SEX 5X '=' MAR-STAT
END-READ
END
```

```
NAME: MORENO              S E X: M     MARITAL STATUS: S
NAME: VAUZELLE            S E X: M     MARITAL STATUS: S
NAME: BAILLET             S E X: M     MARITAL STATUS: S
NAME: HEURTEBISE          S E X: M     MARITAL STATUS: S
NAME: LION                S E X: M     MARITAL STATUS: S
NAME: DEZELUS             S E X: M     MARITAL STATUS: S
NAME: BOYER               S E X: M     MARITAL STATUS: S
NAME: BROUSSE             S E X: M     MARITAL STATUS: S
NAME: DROMARD             S E X: M     MARITAL STATUS: S
NAME: DUC                 S E X: M     MARITAL STATUS: S
NAME: BEGUERIE            S E X: M     MARITAL STATUS: S
NAME: FOREST              S E X: M     MARITAL STATUS: S
NAME: GEORGES             S E X: M     MARITAL STATUS: S
NAME: BOUCLY              S E X: M     MARITAL STATUS: S
```

Equivalent reporting-mode example: See program ACREX1R in library SYSEXRM.

 Copyright © Software AG 2002

Example 2

```
/* EXAMPLE 'ACREX2S': ACCEPT/REJECT (STRUCTURED MODE)
DEFINE DATA LOCAL
 1 EMPLOY-VIEW VIEW OF EMPLOYEES
   2 NAME
   2 FIRST-NAME
   2 SALARY (1)
 1 #PROC-COUNT (N8) INIT <0>
END-DEFINE
EMP. FIND EMPLOY-VIEW WITH NAME = 'JACKSON'
      WRITE NOTITLE *COUNTER NAME FIRST-NAME 'SALARY:' SALARY(1)
/*    ********************************************************
      ACCEPT IF SALARY (1) LT 50000
      WRITE *COUNTER 'ACCEPTED FOR FURTHER PROCESSING'
/*    ********************************************************
      REJECT IF SALARY (1) GT 30000
      WRITE *COUNTER 'NOT REJECTED'
/*    ********************************************************
      ADD 1 TO #PROC-COUNT
     END-FIND
     SKIP 2
     WRITE  NOTITLE   'TOTAL PERSONS FOUND   ' *NUMBER (EMP.)
            /         'TOTAL PERSONS SELECTED' #PROC-COUNT
END
```

```
1 JACKSON              CLAUDE              SALARY:      33000
1 ACCEPTED FOR FURTHER PROCESSING
2 JACKSON              FORTUNA             SALARY:      36000
2 ACCEPTED FOR FURTHER PROCESSING
3 JACKSON              CHARLIE             SALARY:      23000
3 ACCEPTED FOR FURTHER PROCESSING
3 NOT REJECTED


TOTAL PERSONS FOUND            3
TOTAL PERSONS SELECTED         1
```

Equivalent reporting-mode example: See program ACREX2R in library SYSEXRM.

ADD

```
ADD [ROUNDED] operand1... TO operand2
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S	A		N		N	P	I	F		D	T				yes	no
Operand2		S	A		M		N	P	I	F		D	T				yes	yes

```
ADD [ROUNDED] operand1... GIVING operand2
```

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
Operand1	C	S	A		N		N	P	I	F		D	T						yes	no
Operand2		S	A		M	A	N	P	I	F	B	D	T						yes	yes

Related Statement: COMPUTE

Function

The ADD statement is used to add two or more operands.

Operands

At the time the ADD statement is executed, each operand used in the arithmetic operation must contain a valid value.

For additions involving arrays, see also the section Arithmetic Operations with Arrays in the Natural Reference documentation.

As for the formats of the operands, see also the section Performance Considerations for Mixed Formats in the Natural Reference documentation.

 Copyright © Software AG 2002

Result Field - operand2

TO

If the keyword TO is used, *operand2* will be included in the addition and will contain the result of the addition.

GIVING

If the keyword GIVING is used, *operand2* will be used to store the result only. If GIVING is used and *operand2* is defined with alphanumeric format, the result will be converted to alphanumeric.

If a database field is used as the result field, the addition only results in an update to the internal value that is used within the program. The value of the field in the database is not affected.

ROUNDED

If the keyword ROUNDED is used, the result will be rounded. For rules on rounding, see the section Rules for Arithmetic Assignment in the Natural Reference documentation.

Related Statement

COMPUTE.

Example

```
* EXAMPLE 'ADDEX1': ADD
************************
DEFINE DATA LOCAL
 1 #A (P2)
 1 #B (P1.1)
 1 #C (P1)
 1 #DATE (D)
 1 #ARRAY1 (P5/1:4,1:4) INIT (2,*) <5>
 1 #ARRAY2 (P5/1:4,1:4) INIT (4,*) <10>
END-DEFINE
*
ADD +5 -2 -1 GIVING #A
WRITE NOTITLE 'ADD +5 -2 -1 GIVING #A' 15X '=' #A
*
ADD .231 3.6 GIVING #B
WRITE /       'ADD .231 3.6 GIVING #B' 15X '=' #B
*
ADD ROUNDED 2.9 3.8 GIVING #C
WRITE /       'ADD ROUNDED 2.9 3.8 GIVING #C' 8X '=' #C
*
MOVE *DATX TO #DATE
ADD 7 TO #DATE
WRITE / 'CURRENT DATE:'     *DATX (DF=L)13X
        'CURRENT DATE + 7:' #DATE (DF=L)
*
WRITE /       '#ARRAY1 AND #ARRAY2 BEFORE ADDITION'
      /       '=' #ARRAY1 (2,*)
              '=' #ARRAY2 (4,*)
ADD #ARRAY1 (2,*) TO #ARRAY2 (4,*)
WRITE /       '#ARRAY1 AND #ARRAY2 AFTER ADDITION'
      /       '=' #ARRAY1 (2,*)
              '=' #ARRAY2 (4,*)
*
END
```

```
ADD +5 -2 -1 GIVING #A               #A:   2

ADD .231 3.6 GIVING #B               #B:  3.8

ADD ROUNDED 2.9 3.8 GIVING #C        #C:  7

CURRENT DATE: 1999-01-19             CURRENT DATE + 7: 1999-01-26

#ARRAY1 AND #ARRAY2 BEFORE ADDITION
#ARRAY1:      5      5      5      5 #ARRAY2:     10     10     10     10

#ARRAY1 AND #ARRAY2 AFTER ADDITION
#ARRAY1:      5      5      5      5 #ARRAY2:     15     15     15     15
```

 Copyright © Software AG 2002

ASSIGN

See the statement COMPUTE.

AT...

The following is a list of the AT... statements:

- AT BREAK
- AT END OF DATA
- AT END OF PAGE
- AT START OF DATA
- AT TOP OF PAGE

 Copyright © Software AG 2002

AT BREAK

Structured Mode Syntax

```
[AT] BREAK [(r)] [OF] operand1 [/n/]
    statement...
END-BREAK
```

Reporting Mode Syntax

```
[AT] BREAK [(r)] [OF] operand1 [/n/]
    { statement                 }
    { DO statement... DOEND     }
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S				A	N	P	I	F	B	D	T	L				yes	no

Note:
Dynamic or large variables are not allowed to be used as Operand1.

Related Statements: BEFORE BREAK PROCESSING | FIND | READ | HISTOGRAM | SORT | READ WORK FILE

Function

The AT BREAK statement is used to cause the execution of one or more statements whenever a change in value of a control field occurs. It is used in conjunction with automatic break processing and is available with the following statements: FIND, READ, HISTOGRAM, SORT, READ WORK FILE.

An AT BREAK statement block is only executed if the object which contains the statement is active at the time when the break condition occurs.

It is possible to initiate a new processing loop within an AT BREAK condition. This loop must also be closed within the same AT BREAK condition.

Copyright © Software AG 2002

This statement is non-procedural (that is, its execution depends on an event, not on where in a program it is located).

Reference Notation - r

By default, the final AT BREAK condition (for loop termination) is always related to the outermost active processing loop initiated with a FIND, READ, READ WORK FILE, HISTOGRAM or SORT statement.

With the notation "*(r)*" you can relate the final break condition of an AT BREAK statement to another specific currently open processing loop (that is, the loop in which the AT BREAK statement is located or any outer loop).

Example:

```
0110 ...
0120 READ ...
0130   FIND ...
0140     FIND ...
0150       AT BREAK ...
0160         FIND ...
0170         END-FIND
0180       END-BREAK
0190     END-FIND
0200   END-FIND
0210 END-READ
0220 ...
```

In this example, the final AT BREAK condition is related to the READ loop initiated in line 0120. It would be possible to have it related to one of the FIND loops initiated in line 0130 and 0140, but not to the one initiated in line 0160.

If "(r)" is specified for a break hierarchy, it must be specified with the first AT BREAK statement and applies also to all AT BREAK statements which follow.

Control Field - operand1

The field used as the break control field is usually a database field. If a user-defined variable is used, it must be initialized prior to the evaluation of automatic break processing (see BEFORE BREAK PROCESSING statement).

/n/

The notation "/n/" may be used to indicate that only the first n positions of the control field are to be checked for a change in value. This notation can only be used with operands of format A, N or P.

A specific occurrence of an array can also be used as a control field.

A control break occurs when the value of the control field changes, or when all records in the processing loop for which the AT BREAK statement applies have been processed.

Example 1

 Copyright © Software AG 2002

```
/* EXAMPLE 'ATBEX1S': AT BREAK (STRUCTURED MODE)
/************************************************
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 COUNTRY
  2 NAME
END-DEFINE
/************************************************
LIMIT 10
READ EMPLOY-VIEW BY CITY
  AT BREAK OF CITY
     SKIP 1
  END-BREAK
  DISPLAY NOTITLE CITY (IS=ON) COUNTRY (IS=ON) NAME
END-READ
/************************************************
END
```

```
      CITY          COUNTRY         NAME
-------------------- ------- --------------------

AIKEN                USA     SENKO

AIX EN OTHE.         F       GODEFROY

AJACCIO                      CANALE

ALBERTSLUND          DK      PLOUG

ALBUQUERQUE          USA     HAMMOND
                             ROLLING
                             FREEMAN
                             LINCOLN

ALFRETON             UK      GOLDBERG

ALICANTE             E       GOMEZ
```

Equivalent reporting-mode example: See program ATBEX1R in library SYSEXRM.

Example 2

```
/* EXAMPLE 'ATBEX2': AT BREAK USING /N/ NOTATION
/**********************************************
DEFINE DATA LOCAL
  1 EMPLOY-VIEW VIEW OF EMPLOYEES
    2 DEPT
    2 NAME
END-DEFINE
/**********************************************
LIMIT 10
READ EMPLOY-VIEW BY DEPT STARTING FROM 'A'
  AT BREAK OF DEPT /4/
     SKIP 1
  END-BREAK
  DISPLAY NOTITLE DEPT NAME
END-READ
/**********************************************
END
```

```
DEPARTMENT         NAME
     CODE
  ---------- --------------------

  ADMA01     JENSEN
  ADMA01     PETERSEN
  ADMA01     MORTENSEN
  ADMA01     MADSEN
  ADMA01     BUHL
  ADMA02     HERMANSEN
  ADMA02     PLOUG
  ADMA02     HANSEN

  COMP01     HEURTEBISE
  COMP01     TANCHOU
```

System Functions

Natural system functions may be used in conjunction with an AT BREAK statement as described in section System Functions of the Natural Reference documentation.

 Copyright © Software AG 2002

Multiple Break Levels

Multiple AT BREAK statements may be specified within a processing loop within the same program module. If multiple BREAK statements are specified for the same processing loop, they form a hierarchy of break levels independent of whether they are specified consecutively or interspersed within other statements. The first AT BREAK statement represents the lowest control break level, and each additional AT BREAK statement represents the next higher control break level.

Every processing loop in a loop hierarchy may have its own break hierarchy attached.

Example - Structured Mode:

```
FIND ...
   AT BREAK
   ...
   END-BREAK
   AT BREAK
   ...
   END-BREAK
   AT BREAK
   ...
   END-BREAK
END-FIND
...
```

Example - Reporting Mode:

```
FIND ...
   AT BREAK
      DO
      ...
      DOEND
   AT BREAK
      DO
      ...
      DOEND
...
```

A change in the value of a control field in a break level causes break processing to be activated for that break level and all lower break levels, regardless of the values of the control fields for the lower break levels.

For easier program maintenance, it is recommended to specify multiple breaks consecutively.

Copyright © Software AG 2002

Example 3

```
/* EXAMPLE 'ATBEX5S': AT BREAK WITH MULTIPLE BREAK LEVELS
/*                    (STRUCTURED MODE)
/*****************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 DEPT
  2 NAME
  2 LEAVE-DUE
1 LEAVE-DUE-L (N4)
END-DEFINE
/*****************************************************************
LIMIT 5
FIND EMPLOY-VIEW WITH CITY = 'PHILADELPHIA' OR = 'PITTSBURGH'
                 SORTED BY CITY DEPT
  MOVE LEAVE-DUE TO LEAVE-DUE-L
  DISPLAY CITY (IS=ON) DEPT (IS=ON) NAME LEAVE-DUE-L
    AT BREAK OF DEPT
       WRITE NOTITLE /
                 T*DEPT OLD(DEPT) T*LEAVE-DUE-L SUM(LEAVE-DUE-L) /
    END-BREAK
    AT BREAK OF CITY
       WRITE NOTITLE
                 T*CITY OLD(CITY) T*LEAVE-DUE-L SUM(LEAVE-DUE-L) //
    END-BREAK
END-FIND
/*****************************************************************
END
```

 Copyright © Software AG 2002

```
       CITY          DEPARTMENT          NAME          LEAVE-DUE-L
                        CODE
-------------------- ---------- -------------------- -----------

PHILADELPHIA          MGMT30     WOLF-TERROINE            8
                                 MACKARNESS              12

                      MGMT30                             20

                      TECH10     BUSH                     8
                                 NETTLEFOLDS              7

                      TECH10                             15

PHILADELPHIA                                             35


PITTSBURGH            MGMT10     FLETCHER                 3

                      MGMT10                              3

PITTSBURGH                                                3
```

Equivalent reporting-mode example: See program ATBEX5R in library SYSEXRM.

AT END OF DATA

Structured Mode Syntax

```
[AT] END [OF] DATA [(r)]
    statement...
END-ENDDATA
```

Reporting Mode Syntax

```
[AT] END [OF] DATA [(r)]
    { statement
    { DO statement...DOEND
```

Related Statements: AT START OF DATA | FIND | READ | HISTOGRAM | SORT | READ WORK FILE

Function

The AT END OF DATA statement is used to specify processing to be performed when all records selected for a database processing loop have been processed. It must be specified within the same program module which contains the loop creating statement.

This statement is non-procedural (that is, its execution depends on an event, not on where in a program it is located).

Restrictions

This statement can only be used in a processing loop that has been initiated with one of the following statements: FIND, READ, READ WORK FILE, HISTOGRAM or SORT. It may be used only once per processing loop.

This statement is `not` evaluated if the processing loop referenced for END OF DATA processing is not entered.

Reference to a Specific Processing Loop - r

An AT END OF DATA statement may be related to a specific active processing loop by using the "*(r)*" notation. If the "*(r)*" notation is not used, it will be related to the outermost active database processing loop.

 Copyright © Software AG 2002

Values of Database Fields

When the AT END OF DATA condition for the processing loop occurs, all database fields contain the data from the last record processed.

System Functions

Natural system functions may be used in conjunction with an AT END OF DATA statement as described in section System Functions of the Natural Reference documentation.

Example

```
/* EXAMPLE 'AEDEX1S': AT END OF DATA (STRUCTURED MODE)
/*********************************************************************
DEFINE DATA LOCAL
  1 EMPLOY-VIEW VIEW OF EMPLOYEES
    2 PERSONNEL-ID
    2 NAME
    2 FIRST-NAME
    2 SALARY (1)
    2 CURR-CODE (1)
END-DEFINE
/*********************************************************************
LIMIT 5
EMP. FIND EMPLOY-VIEW WITH CITY = 'STUTTGART'
      IF NO RECORDS FOUND
        ENTER
      END-NOREC
      DISPLAY PERSONNEL-ID NAME FIRST-NAME
              SALARY (1) CURR-CODE (1)
/*    ****************************************************************
      AT END OF DATA
         IF *COUNTER (EMP.) = 0
            WRITE 'NO RECORDS FOUND'
            ESCAPE BOTTOM
         END-IF
         WRITE NOTITLE / 'SALARY STATISTICS;'
                       / 7X 'MAXIMUM:' MAX(SALARY(1)) CURR-CODE (1)
                       / 7X 'MINIMUM:' MIN(SALARY(1)) CURR-CODE (1)
                       / 7X 'AVERAGE:' AVER(SALARY(1)) CURR-CODE (1)
      END-ENDDATA
/*    ****************************************************************
      END-FIND
/*********************************************************************
END
```

```
PERSONNEL         NAME              FIRST-NAME        ANNUAL   CURRENCY
     ID                                                 SALARY     CODE
  --------- -------------------- -------------------- ---------- --------

  11100328  BERGHAUS             ROSE                      70800 DM
  11100329  BARTHEL              PETER                     42000 DM
  11300313  AECKERLE             SUSANNE                   55200 DM
  11300316  KANTE                GABRIELE                  61200 DM
  11500304  KLUGE                ELKE                      49200 DM

  SALARY STATISTICS:
         MAXIMUM:      70800 DM
         MINIMUM:      42000 DM
         AVERAGE:      55680 DM
```

Equivalent reporting-mode example: See program AEDEX1R in library SYSEXRM.

 Copyright © Software AG 2002

AT END OF PAGE

Structured Mode Syntax

```
[AT] END [OF] PAGE [(rep)]
    statement...
END-ENDPAGE
```

Reporting Mode Syntax

```
[AT] END [OF] PAGE [(rep)]
    { statement
    { DO statement... DOEND }
```

Related Statements: AT TOP OF PAGE | DISPLAY | WRITE | INPUT | NEWPAGE

Function

The AT END OF PAGE statement is used to specify processing that is to be performed when an end-of-page condition is detected (see the session parameter PS in the Natural Reference documentation).

An end-of-page condition may also occur as a result of a SKIP or NEWPAGE statement, but not as a result of an EJECT or INPUT statement.

An AT END OF PAGE statement block is only executed if the object which contains the statement block is active at the time when the end-of-page condition occurs.

An AT END OF PAGE statement must not be placed within an inline subroutine.

This statement is non-procedural (that is, its execution depends on an event, not on where in a program it is located).

Report Specification - rep

The notation *(rep)* may be used to specify the identification of the report for which the AT END OF PAGE statement is applicable. A value in the range 0 - 31 or a logical name which has been assigned using the DEFINE PRINTER statement may be specified.
If *(rep)* is not specified, the AT END OF PAGE statement will apply to the first report (report 0).

Copyright © Software AG 2002

Logical Page Size

The end-of-page check is performed after the processing of a DISPLAY or WRITE statement is completed. Therefore, if a DISPLAY or WRITE statement produces multiple lines of output, overflow of the physical page may occur before an end-of-page condition is detected.

A logical page size (session parameter PS) which is less than the physical page size must be specified to ensure that information printed by an AT END OF PAGE statement appears on the same physical page as the title.

Last-Page Handling

Within a main program, an end-of-page condition is activated when the execution of the main program terminates via ESCAPE, STOP or END.

Within a subroutine, an end-of-page condition is `not` activated when the execution of the subroutine terminates via ESCAPE, RETURN or END.

System Functions

Natural system functions may be used in conjunction with an AT END OF PAGE statement as described in the section System Functions of the Natural Reference documentation.

If a system function is to be used within an AT END OF PAGE statement block, the GIVE SYSTEM FUNCTIONS clause must be specified in the corresponding DISPLAY statement.

INPUT Statement with AT END OF PAGE

If an INPUT statement is specified within an AT END OF PAGE statement block, no new page operation is performed. The page size (session parameter PS) must be reduced to a value that allows the lines created by the INPUT statement to appear on the same physical page. See also INPUT statement "Split Screen Feature". See also Example 2.

Example 1

 Copyright © Software AG 2002

```
/* EXAMPLE 'AEPEX1S': AT END OF PAGE (STRUCTURED MODE)
/*********************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 SALARY (1)
  2 CURR-CODE (1)
END-DEFINE
/*********************************************************************
FORMAT PS=10
LIMIT 10
READ EMPLOY-VIEW BY PERSONNEL-ID FROM '20017000'
  DISPLAY NOTITLE GIVE SYSTEM FUNCTIONS
          NAME JOB-TITLE 'SALARY' SALARY(1) CURR-CODE (1)
/* ******************************************************************
  AT END OF PAGE
   WRITE / 28T 'AVERAGE SALARY: ...' AVER(SALARY(1)) CURR-CODE (1)
  END-ENDPAGE
/* ******************************************************************
END-READ
/*********************************************************************
END
```

```
      NAME                  CURRENT            SALARY   CURRENCY
                             POSITION                       CODE
-------------------- ------------------------- ---------- --------

CREMER               ANALYST                        34000 USD
MARKUSH              TRAINEE                        22000 USD
GEE                  MANAGER                        39500 USD
KUNEY                DBA                            40200 USD
NEEDHAM              PROGRAMMER                     32500 USD
JACKSON              PROGRAMMER                     33000 USD

                           AVERAGE SALARY: ...      33533 USD
```

Equivalent reporting-mode example: See program AEPEX1R in library SYSEXRM.

Example 2

```
/* EXAMPLE 'AEPEX2': AT END OF PAGE WITH INPUT STATEMENT
/*********************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 POST-CODE
  2 CITY
1 #START-NAME(A20)
END-DEFINE
/*********************************************************************
FORMAT PS=21
REPEAT
  READ (15) EMPLOY-VIEW BY NAME = #START-NAME
    DISPLAY NOTITLE NAME FIRST-NAME POST-CODE CITY
  END-READ
  NEWPAGE
/*********************************************************************
  AT END OF PAGE
    MOVE NAME TO #START-NAME
    INPUT / '-' (79) / 10T 'Reposition to name ==>'
      #START-NAME (AD=MI) '(''.'' to exit)'
    IF #START-NAME = '.'
      STOP
    END-IF
  END-ENDPAGE
/*********************************************************************
END-REPEAT
END
```

```
      NAME               FIRST-NAME        POSTAL           CITY
                                            ADDRESS
-------------------- -------------------- ---------- --------------------

ALEXANDER            STEPHEN              19711      NEWARK
ALEXANDER            GIL                  21209      BALTIMORE
ALEXANDER            CHARLY               95616      DAVIS
ALEXANDER            HOLLY                53706      MADISON
ALLEGRE              KARL-OTTO            8100       CHARLEVILLE MEZIERES
ALLSOP               ALAN                 DE3 3NL    DERBY
ALVAREZ              RAQUEL               28015      MADRID
AMOROS               FUENSANTA            28014      MADRID
ANDERSEN             ANITA                1850  V    KÖBENHAVN
ANDERSEN             KARIN                2720       VANLÖSE
ANDERSEN             LISSI                2650       HVIDOVRE
ANDERSON             JENNY                84112      SALT LAKE CITY
ANTLIFF              JANET                DE3 3EE    DERBY
ARCHER               ROBIN                DE4 8GR    DERBY
ARCONADA             ARANZAZU             28014      MADRID

------------------------------------------------------------------------------
         Reposition to name ==> ARCONADA            ('.' to exit)
```

 Copyright © Software AG 2002

AT START OF DATA

Structured Mode Syntax

```
[AT] START [OF] DATA [(r)]
    statement...
END-START
```

Reporting Mode Syntax

```
[AT] START [OF] DATA [(r)]
    { statement
    { DO statement... DOEND
```

Related Statements: AT END OF DATA | FIND | READ | HISTOGRAM | SORT | READ WORK FILE

Function

The statement AT START OF DATA is used to perform processing immediately after the first of a set of records is read for a processing loop that has been initiated by one of the following statements: READ, FIND, HISTOGRAM, SORT or READ WORK FILE. If the loop-initiating statement contains a WHERE clause, the at-start-of-data condition will be true when the first record is read which meets both the basic search and the WHERE criteria.

This statement is non-procedural (that is, its execution depends on an event, not on where in a program it is located).

Value of Database Fields

All database fields contain the values of the record which caused the at-start-of-data condition to be true (that is, the first record of the set of records to be processed).

Positioning

This statement must be positioned **within** a processing loop, and it may be used only once per processing loop.

Copyright © Software AG 2002

Reference to a Specific Processing Loop - *r*

An AT START OF DATA statement may be related to a specific outer active processing loop by using the "*(r)*" notation. If this notation is not used, the statement is related to the outermost active processing loop.

Example

```
/* EXAMPLE 'ASDEX1S': AT START OF DATA (STRUCTURED MODE)
/*********************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
1 #CNTL (A1)  INIT <' '>
1 #CITY (A20) INIT <' '>
END-DEFINE
/*********************************************************************
REPEAT
  INPUT 'ENTER VALUE FOR CITY'  #CITY
    IF #CITY = ' ' OR = 'END'
     STOP
    END-IF
  FIND EMPLOY-VIEW WITH CITY = #CITY
    IF NO RECORDS FOUND
     WRITE NOTITLE NOHDR 'NO RECORDS FOUND'
     ESCAPE BOTTOM
    END-NOREC
/*********************************************************************
    AT START OF DATA
     INPUT (AD=O) 'RECORDS FOUND' *NUMBER //
                  'ENTER ''D'' TO DISPLAY RECORDS' #CNTL (AD=A)
     IF #CNTL NE 'D'
      ESCAPE BOTTOM
     END-IF
    END-START
/*********************************************************************
    DISPLAY NAME FIRST-NAME
  END-FIND
END-REPEAT
END
```

```
ENTER VALUE FOR CITY PARIS
```

```
RECORDS FOUND        24

ENTER 'D' TO DISPLAY RECORDS D
```

 Copyright © Software AG 2002

```
       NAME                FIRST-NAME
-------------------- --------------------

MAIZIERE             ELISABETH
MARX                 JEAN-MARIE
REIGNARD             JACQUELINE
RENAUD               MICHEL
REMOUE               GERMAINE
LAVENDA              SALOMON
BROUSSE              GUY
GIORDA               LOUIS
SIECA                FRANCOIS
CENSIER              BERNARD
DUC                  JEAN-PAUL
CAHN                 RAYMOND
MAZUY                ROBERT
VALLY                ALAIN
BRETON               JEAN-MARIE
GIGLEUX              JACQUES
XOLIN                CHRISTIAN
LEGRIS               ROGER
RIVIERE              JEAN-LUC
REICH                MARC
VVVV
```

Equivalent reporting-mode example: See program ASDEX1R in library SYSEXRM.

AT TOP OF PAGE

Structured Mode Syntax

```
[AT] TOP [OF] PAGE [(rep)]
    statement...
END-TOPPAGE
```

Reporting Mode Syntax

```
[AT] TOP [OF] PAGE [(rep)]
    { statement
    { DO statement...DOEND }
```

Related Statements: AT END OF PAGE | NEWPAGE.

Function

The statement AT TOP OF PAGE is used to specify processing which is to be performed when a new page is started.

A new page is started when the internal line counter exceeds the page size set with the session parameter PS, or when a NEWPAGE statement is executed. Either of these events cause a top-of-page condition to be true. An EJECT statement causes a new page to be started but does not cause a top-of-page condition.

An AT TOP OF PAGE statement block is only executed when the object which contains the statement is active at the time when the top-of-page condition occurs.

Any output created as a result of AT TOP OF PAGE processing will appear following the title line with an intervening blank line.

This statement is non-procedural (that is, its execution depends on an event, not on where in a program it is located).

Restriction

An AT TOP OF PAGE statement must not be placed within an inline subroutine.

 Copyright © Software AG 2002

Report Specification - *rep*

The notation *(rep)* may be used to specify the identification of the report for which the AT TOP OF PAGE statement is applicable. A value in the range 0 - 31 or a logical name which has been assigned using the DEFINE PRINTER statement may be specified.

If *(rep)* is not specified, the AT TOP OF PAGE statement applies to the first report (report 0).

Example

```
/* EXAMPLE 'ATPEX1S': AT TOP OF PAGE (STRUCTURED MODE)
/**********************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 DEPT
END-DEFINE
/**********************************************************************
FORMAT PS=15
LIMIT 15
READ EMPLOY-VIEW BY NAME STARTING FROM 'L'
 DISPLAY 2X NAME 4X FIRST-NAME CITY DEPT
 WRITE TITLE UNDERLINED 'EMPLOYEE REPORT'
 WRITE TRAILER '-' (78)
/**********************************************************************
  AT TOP OF PAGE
   WRITE 'BEGINNING NAME:' NAME
  END-TOPPAGE
/**********************************************************************
  AT END OF PAGE
   SKIP 1
   WRITE 'ENDING NAME:   ' NAME
  END-ENDPAGE
END-READ
END
```

```
                              EMPLOYEE REPORT
-------------------------------------------------------------------------------
BEGINNING NAME: LAFON
          NAME                FIRST-NAME              CITY         DEPARTMENT
                                                                      CODE
  --------------------    -------------------- -------------------- ----------

  LAFON                   CHRISTIANE           PARIS                VENT18
  LANDMANN                HARRY                ESCHBORN             MARK29
  LANE                    JACQUELINE           DERBY                MGMT02
  LANKATILLEKE            LALITH               FRANKFURT            PROD22
  LANNON                  BOB                  LINCOLN              SALE20
  LANNON                  LESLIE               SEATTLE              SALE30
  LARSEN                  CARL                 FARUM                SYSA01
  LARSEN                  MOGENS               VEMMELEV             SYSA02
-------------------------------------------------------------------------------

ENDING NAME:    LARSEN
```

Equivalent reporting-mode example: See program ATPEX1R in library SYSEXRM.

 Copyright © Software AG 2002

BACKOUT TRANSACTION

```
BACKOUT [TRANSACTION]
```

Function

This statement is used to back out all database updates performed during the current logical transaction. This statement also releases all records held during the transaction.

The BACKOUT TRANSACTION statement is executed only if a database transaction under control of Natural has taken place. For which databases the statement is executed depends on the setting of the profile parameter ET (see your Natural Installation and Operations documentation):

- If ET=OFF, the statement is executed only for the database affected by the transaction.
- If ET=ON, the statement is executed for all databases that have been referenced since the last execution of a BACKOUT TRANSACTION or END TRANSACTION statement.

Note for ENTIRE SYSTEM SERVER:
This statement is not available with ENTIRE SYSTEM SERVER.

Considerations for DL/I Databases

Because PSB scheduling is terminated by a syncpoint request, Natural saves the PSB position before executing the BACKOUT TRANSACTION statement. Before the next command execution, Natural re-schedules the PSB and tries to set the PCB position as it was before the backout. The PCB position might be shifted forward if any pointed segment had been deleted in the time period between the backout and the following command.

Considerations for SQL Databases

As most SQL databases close all cursors when a logical unit of work ends, a BACKOUT TRANSACTION statement must not be placed within a database modification loop; instead, it has to be placed after such a loop.

Backout Transaction Issued by Natural

If the user interrupts the current Natural operation with a terminal command (command "%%" or CLEAR key), Natural issues a BACKOUT TRANSACTION statement (see also the terminal command "%%" in the Natural Reference documentation).

Additional Information

For additional information on the use of the transaction backout feature, see the section Database Access of the Natural Programming Guide.

Copyright © Software AG 2002

Example

```
/* EXAMPLE 'BOTEX1S': BACKOUT TRANSACTION (STRUCTURED MODE)
/*********************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 DEPT
  2 LEAVE-DUE
  2 LEAVE-TAKEN
1 #DEPT (A6)
1 #RESP (A3)
END-DEFINE
/*********************************************************************
LIMIT 3
INPUT 'DEPARTMENT TO BE UPDATED:' #DEPT
 IF #DEPT = ' '
  STOP
 END-IF
/*********************************************************************
FIND EMPLOY-VIEW WITH DEPT = #DEPT
 IF NO RECORDS FOUND
  REINPUT 'NO RECORDS FOUND'
 END-NOREC
 INPUT 'NAME:       ' NAME (AD=O) /
       'LEAVE DUE:  ' LEAVE-DUE (AD=M) /
       'LEAVE TAKEN:' LEAVE-TAKEN (AD=M)
 UPDATE
END-FIND
/*********************************************************************
INPUT 'UPDATE TO BE PERFORMED  YES/NO:' #RESP
 DECIDE ON FIRST #RESP
           VALUE 'YES'
             END TRANSACTION
           VALUE 'NO'
             BACKOUT TRANSACTION
           NONE
             REINPUT 'PLEASE ENTER YES OR NO'
 END-DECIDE
/*********************************************************************
END
```

```
DEPARTMENT TO BE UPDATED: MGMT30
```

 Copyright © Software AG 2002

```
NAME:        POREE
LEAVE DUE:   45
LEAVE TAKEN: 31
```

```
UPDATE TO BE PERFORMED  YES/NO: NO
```

Equivalent reporting-mode example: See program BOTEX1R in library SYSEXRM.

BEFORE BREAK PROCESSING

Structured Mode Syntax

```
BEFORE [BREAK] [PROCESSING]
    statement...
END-BEFORE
```

Reporting Mode Syntax

```
BEFORE [BREAK] [PROCESSING]
    { statement                 }
    { DO statement... DOEND     }
```

Related Statement: AT BREAK

Function

The BEFORE BREAK PROCESSING statement may be used in conjunction with automatic break processing to perform processing:

- before the value of the break control field is checked;
- before the statements specified with an AT BREAK statement are executed;
- before Natural system functions are evaluated.

This statement is most often used to initialize or compute values of user-defined variables which are to be used in break processing (see AT BREAK statement).

If no break processing is to be performed (that is, no AT BREAK statement is specified for the processing loop), any statements specified with a BEFORE BREAK PROCESSING statement will **not** be executed.

This statement is non-procedural (that is, its execution depends on an event, not on where in a program it is located).

Restrictions

The BEFORE BREAK PROCESSING statement may only be used with a processing loop that has been initiated with one of the following statements: FIND, READ, HISTOGRAM, SORT or READ WORK FILE. It may be placed anywhere within the processing loop and is always related to the processing loop in which it is contained. Only one BEFORE BREAK PROCESSING statement may be specified per processing loop.

 Copyright © Software AG 2002

The statement BEFORE BREAK PROCESSING must not be used in conjunction with the statement PERFORM BREAK PROCESSING.

Example

```
/* EXAMPLE 'BBPEX1': BEFORE BREAK PROCESSING
/*********************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 SALARY (1)
  2 BONUS (1,1)
1 #INCOME (P11)
END-DEFINE
/*********************************************************************
LIMIT 7
READ EMPLOY-VIEW BY CITY = 'L'
/*********************************************************************
 BEFORE BREAK PROCESSING
  COMPUTE #INCOME = SALARY (1) + BONUS (1,1)
 END-BEFORE
/*********************************************************************
 AT BREAK OF CITY
  WRITE NOTITLE  'AVERAGE INCOME FOR' OLD (CITY) 20X AVER(#INCOME) /
 END-BREAK
/*********************************************************************
 DISPLAY CITY 'NAME' NAME 'SALARY' SALARY (1) 'BONUS' BONUS (1,1)
END-READ
END
```

```
      CITY                 NAME           SALARY     BONUS
-------------------- -------------------- ---------- ----------

LA BASSEE            HULOT                    165000      70000
AVERAGE INCOME FOR LA BASSEE                                        235000

LA CHAPELLE ST LUC   GUILLARD                 124100      23000
LA CHAPELLE ST LUC   BERGE                    198500      50000
LA CHAPELLE ST LUC   POLETTE                  124090      23000
LA CHAPELLE ST LUC   DELAUNEY                 115000      23000
LA CHAPELLE ST LUC   SCHECK                   125600      23000
LA CHAPELLE ST LUC   KREEBS                   184550      50000
AVERAGE INCOME FOR LA CHAPELLE ST LUC                               177306
```

CALL

CALL [INTERFACE4] *operand1* **[USING] [***operand2***]...** *128*

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S				A												yes	no
Operand2	C	S	A	G		A	N	P	I	F	B	D	T	L	C	G		yes	yes

- CALL on Mainframe Computers
- Part I: CALL under OpenVMS, UNIX and Windows
- INTERFACE4
- Part II: CALL under OpenVMS, UNIX and Windows

CALL on Mainframe Computers

- Function
- Program Name - operand1
- Parameters - operand2
- Return Code
- Register Usage
- Boundary Alignment
- Adabas Calls
- Direct/Dynamic Loading
- Example
- Linkage Conventions
- Calling a PL/I Program

Function

The CALL statement is used to call an external program written in another standard programming language from a Natural program and then return to the next statement after the CALL statement.

The called program may be written in any programming language which supports a standard CALL interface. Multiple CALL statements to one or more external programs may be specified.

A CALL statement may be issued within a program to be executed under control of a TP monitor, provided that the TP monitor supports a CALL interface.

Program Name - *operand1*

The name of the program to be called *(operand1)* can be specified as a constant or - if different programs are to be called dependent on program logic - as an alphanumeric variable of length 1 to 8. A program name must be placed left-justified in the variable.

 Copyright © Software AG 2002

Parameters - *operand2*

The CALL statement may contain up to 128 parameters *(operand2)*, unless the INTERFACE4 option is used. In that case, there is no limit as to how many parameters may be used. Standard linkage register conventions are used. One address is passed in the parameter list for each parameter field specified.

If a group name is used, the group is converted to individual fields; that is, if a user wishes to specify the beginning address of a group, the first field of the group must be specified.

Note:
The internal representation of positive signs of packed numbers is changed to the value specified by the PSIGNF parameter of the NTCMPO macro **before** control is passed to the external program.

Return Code

The condition code of any called program (content of register 15 upon return to Natural) may be obtained by using the Natural system function RET.

Example:

```
...
RESET #RETURN(B4)
CALL 'PROG1'
IF RET ('PROG1') > #RETURN
  WRITE 'ERROR OCCURRED IN PROGRAM1'
END-IF
...
```

Register Usage

Register	Contents
R1	Address pointer to the parameter address list.
R2	Address pointer to the field (parameter) description list. The field description list contains information on the first 128 fields passed in the parameter list. Each description is a 4-byte entry containing the following information: - the 1st byte contains the type of variable (A,B,...) If field type is "N" or "P": - the 2nd byte contains the total number of digits; - the 3rd byte contains the number of digits before the decimal point; - the 4th byte contains the number of digits after the decimal point. all other field types: - the 2nd byte is unused; - the 3rd-4th byte contain the length of field.
R3	Address pointer to list of field lengths. This list contains the length of each field passed in the parameter list. In the case of an array, the length is the sum of the individual occurrences' lengths.
R13	Address of 18-word save area.
R14	Return address.
R15	Entry address/return code.

Boundary Alignment

The Natural data area, in which all user-defined variables are stored, always begins on a double-word boundary.

If DEFINE DATA is used, all data blocks (for example, LOCAL, GLOBAL blocks) are double-word aligned, and all structures (level 1) are full-word aligned.

Alignment within the data area is the responsibility of the user and is governed by the order in which variables are defined to Natural.

 Copyright © Software AG 2002

Adabas Calls

A called program may contain a call to Adabas. The called program must not issue an Adabas open or close command. Adabas will open all database files referenced. If Adabas exclusive (EXU) update mode is to be used, the Natural profile parameter OPRB must be used in order to open all referenced files. Before you attempt to use EXU update mode, you should consult your Natural administrator.

Direct/Dynamic Loading

The called program may either be directly linked to the Natural nucleus (that is, the program is specified with the CSTATIC parameter in the Natural parameter module; see also the Natural Operations documentation for Mainframes, or it may be loaded dynamically the first time it is called. If it is to be loaded dynamically, the load module library containing the called program must be concatenated to the Natural load library in the Natural execution JCL or in the appropriate TP-monitor program library. Ask your Natural administrator for additional information.

Example

The example on the next page shows a Natural program which calls the COBOL program "TABSUB" for the purpose of converting a country code into the corresponding country name. Two parameter fields are passed by the Natural program to TABSUB: the first parameter is the country code, as read from the database; the second parameter is used to return the corresponding country name.

Calling Natural Program:

```
* EXAMPLE 'CALEX1': CALL PROGRAM 'TABSUB'
* ***************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 BIRTH
  2 COUNTRY
1 #COUNTRY (A3)
1 #COUNTRY-NAME (A15)
END-DEFINE
*
MOVE EDITED '19550701' to #FIND-FROM (EM=YYYYMMDD)
MOVE EDITED #19550731' to #FIND-TO   (EM=YYYYMMDD)
*
FIND EMPLOY-VIEW WITH BIRTH = #FIND-FROM THRU #FIND-TO
  MOVE COUNTRY TO #COUNTRY
  CALL 'TABSUB' #COUNTRY #COUNTRY-NAME
  DISPLAY NAME BIRTH (EM=YYYY-MM-DD) #COUNTRY-NAME
END-FIND
END
```

Called COBOL program "TABSUB":

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TABSUB.
REMARKS. THIS PROGRAM PROVIDES THE COUNTRY NAME
        FOR A GIVEN COUNTRY CODE.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
01 COUNTRY-CODE  PIC X(3).
01 COUNTRY-NAME  PIC X(15).
PROCEDURE DIVISION USING COUNTRY-CODE COUNTRY-NAME.
P-CONVERT.
   MOVE SPACES TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'BLG' MOVE 'BELGIUM' TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'DEN' MOVE 'DENMARK' TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'FRA' MOVE 'FRANCE' TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'GER' MOVE 'GERMANY' TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'HOL' MOVE 'HOLLAND' TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'ITA' MOVE 'ITALY' TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'SPA' MOVE 'SPAIN' TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'UK'  MOVE 'UNITED KINGDOM' TO COUNTRY-NAME.
P-RETURN.
GOBACK.
```

Linkage Conventions

- CALL using Com-plete
- CALL using CICS
- Return Codes under CICS
- Example using CICS

Standard linkage register notation is used in batch mode. Each TP monitor has its own conventions. These conventions must be followed; otherwise, unpredictable results could occur. The following sections describe conventions that apply for the supported TP monitors.

CALL using Com-plete

The called program must reside in the Com-plete online load library. This allows Com-plete to load the program dynamically. The Com-plete utility ULIB may be used to catalog the program.

 Copyright © Software AG 2002

CALL using CICS

The called program must reside in either a load module library concatenated to the CICS library or the DFHRPL library. The program must also have a PPT entry in the operating PPT so that CICS can locate the program and load it.

The linkage convention passes the parameter list address followed by the field description list address in the first fullwords of the TWA and the COMMAREA.The parameter FLDLEN in the NCIPARM parameter module controls if the field length list is also passed (by default, it is not passed). The COMMAREA length (8 or 12) reflects the number of list addresses passes (2 or 3). The last list address is indicated by the high-order bit being set. The user must ensure addressability to the TWA or to the COMMAREA respectively. This is only required if the user program has not been defined to Natural as a static or directly linked program, in which case the pointer to the parameter list is passed via register 1, the pointer to the description list via register 2, and the pointer to the field length list via register 3.

If you wish the parameter values themselves, rather than the address of their address list, to be passed in the COMMAREA, issue the Natural terminal command %P=C before the call.

Normally, when a Natural programs calls a non-Natural program and the called program issues a conversational terminal I/O, the Natural thread is blocked until the user has entered data. To prevent the Natural thread from being blocked, the terminal command %P=V can be used

Normally, when a Natural program calls a non-Natural program under CICS, the call is accomplished by an "EXEC CICS LINK" request. If standard linkage is to be used for the call instead, issue the terminal command %P=S (In this case, the called program must adhere to standard linkage conventions with standard register usage).

In 31-bit-mode environments the following applies: if a program linked with AMODE=24 is called and the threads are above 16 MB, a "call by value" will be done automatically, that is, the specified parameters which are to be passed to the called program will be copied below 16 MB.

Return Codes under CICS

CICS itself does not support condition codes for a call with CICS conventions (EXEC CICS LINK). However, the Natural CICS Interface supports return codes for the CALL statement: When control is returned from the called program, Natural checks whether the first fullword of the COMMAREA has changed. If it has, its new content will be taken as the return code. If it has not changed, the first fullword of the TWA will be checked and its new content taken as the return code. If neither of the two fullwords has changed, the return code will be "0".

Note:
When parameter values are passed in the COMMAREA (%P=C), the return code is always "0".

Example using CICS:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TABSUB.
REMARKS. THIS PROGRAM PERFORMS A TABLE LOOK-UP AND
         RETURNS A TEXT MESSAGE.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 MSG-TABLE.
   03  FILLER       PIC X(15) VALUE 'MESSAGE1       '.
   03  FILLER       PIC X(15) VALUE 'MESSAGE2       '.
   03  FILLER       PIC X(15) VALUE 'MESSAGE3       '.
   03  FILLER       PIC X(15) VALUE 'MESSAGE4       '.
   03  FILLER       PIC X(15) VALUE 'MESSAGE5       '.
   03  FILLER       PIC X(15) VALUE 'MESSAGE6       '.
   03  FILLER       PIC X(15) VALUE 'MESSAGE7       '.
01 TAB REDEFINES MSG-TABLE.
   03  MESSAGE OCCURS 7 TIMES PIC X(15).
LINKAGE SECTION.
01 TWA-DATA.
   03 PARM-POINTER USAGE IS POINTER.
01 PARM-LIST.
   03 DATA-LOC-IN  USAGE IS POINTER.
   03 DATA-LOC-OUT USAGE IS POINTER.
01 INPUT-DATA.
   03  INPUT-NUMBER      PIC 99.
01 OUTPUT-DATA.
   03  OUTPUT-MESSAGE    PIC X(15).
PROCEDURE DIVISION.
100-INIT.
    EXEC CICS ADDRESS TWA(ADDRESS OF TWA-DATA) END-EXEC.
    SET ADDRESS OF PARM-LIST   TO PARM-POINTER.
    SET ADDRESS OF INPUT-DATA  TO DATA-LOCIN.
    SET ADDRESS OF OUTPUT-DATA TO DATA-LOC-OUT.
200-PROCESS.
    MOVE MESSAGE (INPUT-NUMBER) TO OUTPUT-MESSAGE.
300-RETURN.
    EXEC CICS RETURN END-EXEC.
400-DUMMY.
    GO-BACK.
```

 Copyright © Software AG 2002

Calling a PL/I Program

- Example of Calling a PL/I Program:
- Example of Calling a PL/I Program which is Operating under CICS

A called program written in PL/I requires the following additional procedures:

- The ENTRY PLICALLA statement must be provided when the program is link-edited. This statement causes the PL/I load module to receive control as a sub-program (that is, a called program).
  If the PL/I program is to be called recursively, you may also use the program NATPLICA, which is contained in the Natural source library. NATPLICA is an example of how a PL/I program can be called recursively from a Natural program without causing any storage bottlenecks (for further details, please refer to the comments in the program NATPLICA itself). A complete description of the ENTRY PLICALLA statement and further information on how to call a PL/I program can be found in the relevant IBM PL/I documentation.
- Since the parameter list is a standard list and is not an argument list being passed from another PL/I program, the addresses passed do not point at a LOCATOR DESCRIPTOR. This problem may be resolved by defining the parameter fields as arithmetic variables. This causes PL/I to treat the parameter list as addresses of data instead of addresses of LOCATOR DESCRIPTOR control blocks.

The technique suggested for defining the parameter fields is illustrated in the following example:

```
PLIPROG: PROC(INPUT_PARM_1, INPUT_PARM_2) OPTIONS(MAIN);
      DECLARE (INPUT_PARM_1, INPUT_PARM_2) FIXED;
      PTR_PARM_1 = ADDR(INPUT_PARM_1);
      PTR_PARM_2 = ADDR(INPUT_PARM_2);
      DECLARE FIRST_PARM        PIC '99'   BASED (PTR_PARM_1);
      DECLARE SECOND_PARM       CHAR(12)   BASED (PTR_PARM_2);
```

Each parameter in the input list should be treated as a unique element. The number of input parameters should exactly match the number being passed from the Natural program. The input parameters and their attributes must match the Natural definitions or unpredictable results may occur. For additional information on passing parameters in PL/I, see the relevant IBM PL/I documentation.

Copyright © Software AG 2002

Example of calling a PL/I Program:

```
/* EXAMPLE 'CALEX2': CALL PROGRAM 'NATPLI'
/*********************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 AREA-CODE
  2 REDEFINE AREA-CODE
    3 #AC(N1)
1 #INPUT-NUMBER (N2)
1 #OUTPUT-COMMENT (A15)
END-DEFINE
/*********************************************************************
READ EMPLOY-VIEW IN LOGICAL SEQUENCE BY NAME
                 STARTING FROM 'WAGNER'
 MOVE ' ' TO #OUTPUT-COMMENT
 MOVE #AC TO #INPUT-NUMBER
CALL 'NATPLI' #INPUT-NUMBER #OUTPUT-COMMENT
END-READ
END
```

```
NATPLI:  PROC(PARM_COUNT, PARM_COMMENT) OPTIONS(MAIN);
    /*                                          */
    /* THIS PROGRAM ACCEPTS AN INPUT NUMBER     */
    /* AND TRANSLATES IT TO AN OUTPUT CHARACTER */
    /* STRING FOR PLACEMENT ON THE FINAL        */
    /* NATURAL REPORT                           */
    /*                                          */
    /*                                          */
   DECLARE PARM_COUNT, PARM_COMMENT  FIXED;
   DECLARE ADDR BUILTIN;
   COUNT_PTR = ADDR(PARM_COUNT);
   COMMENT_PTR = ADDR(PARM_COMMENT);
   DECLARE  INPUT_NUMBER    PIC '99' BASED (COUNT_PTR);
   DECLARE  OUTPUT_COMMENT  CHAR(15) BASED (COMMENT_PTR);
   DECLARE COMMENT_TABLE(9) CHAR(15) STATIC INITIAL
      ('COMMENT1   ',
       'COMMENT2   ',
       'COMMENT3   ',
       'COMMENT4   ',
       'COMMENT5   ',
       'COMMENT6   ',
       'COMMENT7   ',
       'COMMENT8   ',
       'COMMENT9   ');
    OUTPUT_COMMENT = COMMENT_TABLE(INPUT_NUMBER);
    RETURN;
END NATPLI;
```

 Copyright © Software AG 2002

Example of Calling a PL/I Program which is Operating under CICS:

```
/* EXAMPLE 'CALEX3': CALL PROGRAM 'CICSP'
/****************************************
DEFINE DATA LOCAL
1 #MESSAGE (A10) INIT <' '>
END-DEFINE
/****************************************
CALL 'CICSP' #MESSAGE
DISPLAY #MESSAGE
/****************************************
END
```

```
CICSP: PROCEDURE OPTIONS (MAIN REENTRANT);
       DCL  1        TWA_ADDRESS    BASED(TWA_POINTER);
            2        LIST_ADDRESS   POINTER;
       DCL  1 PTR_TO_LIST           BASED(LIST_ADDRESS);
            2 PARM_01               POINTER;
       DCL MESSAGE CHAR(10) BASED(PARM_01);
       EXEC CICS ADDRESS TWA(TWA_POINTER);
       MESSAGE='SUCCESS'; EXEC CICS RETURN; END CICSP;
```

Part I: CALL under OpenVMS, UNIX and Windows

- Function
- Name of Called Function (operand1)
- Parameters (operand2)

Function

The CALL statement is used to call an external function written in another standard programming language from a Natural program and then return to the next statement after the CALL statement.

The called function may be written in any programming language which supports a standard CALL interface. Multiple CALL statements to one or more external functions may be specified.

Name of Called Function - *operand1*

The name of the function to be called *(operand1)* may be specified as a constant or - if different functions are to be called dependent on program logic - as an alphanumeric variable of length 1 to 8. A function name must be placed left-justified in the variable.

Parameters - *operand2*

The CALL statement may contain up to 128 parameters *(operand2)*. One address is passed to the external function in the parameter list for each parameter field specified.

If a group name is used, the group is converted to individual fields; that is, if a user wishes to specify the beginning address of a group, the first field of the group must be specified.

Note:
If an application-independent variable (AIV) or context variable is passed as a parameter to a user exit, the following restriction applies: if the user exit invokes a Natural subprogram which creates a new AIV or context variable, the parameter is invalid after the return from the subprogram. This is true regardless of whether the new AIV/context variable is created by the subprogram itself or by another object invoked directly or indirectly by the subprogram.

 Copyright © Software AG 2002

INTERFACE4

- INTERFACE 4 - External 3GL Program Interface
- Operand Structure for Interface4
- INTERFACE4 Parameter Access
- Exported Functions

Note:
The INTERFACE4 option is not available on mainframe computers.

The keyword INTERFACE4 specifies the type of the interface that is used for the call of the external program. This keyword is optional. If this keyword is specified, the interface, which is defined as 'Interface4', is used for the call of the external program. The following table lists the differences between the CALL statement used with INTERFACE4 and the one used without INTERFACE4:

	CALL statement without keyword INTERFACE4	**Call statement with keyword INTERFACE4**
number of parameters possible	128	unlimited (32767)
maximum data size of one parameter	64 K	1 GB
retrieve array information	no	yes
support of large and dynamic operands	no	yes
parameter access via API	no	yes

INTERFACE4 - External 3GL Program Interface

The interface of the external 3GL program is defined as follows, when the INTERFACE4 is specified with the Natural CALL statement:

```
NATFCT functionname (numparm, parmhandle, traditional)
```

USR_WORD	numparm;	16 bit unsigned short value, containing the total number of transferred operands (operand2)
void	*parmhandle;	Pointer to the parameter passing structure.
void	*traditional;	Check for interface type (if it's not a NULL pointer it's the traditional CALL interface)

Operand Structure for Interface4

The operand structure of Interface4 is named ’parameter_description’ and is defined as follows. The structure is delivered with the header file natuser.h.

struct parameter_description		
void *	address	address of the parameter data, not aligned, realloc() and free() are not allowed
int	format	field data type: NCXR_TYPE_ALPHA, etc. (natuser.h)
int	length	length (before decimal point, if applicable)
int	precision	length after decimal point (if applicable)
int	byte_length	length of field in bytes int dimension number of dimensions (0 to IF4_MAX_DIM)
int	dimensions	number of dimensions (0 to IF4_MAX_DIM)
int	length_all	total data length of array in bytes
int	flags	several flag bits combined by bitwise OR, meaning: IF4_FLG_PROTECTED the parameter is write protected, IF4_FLG_DYNAMIC the parameter is a dynamic variable, IF4_FLG_NOT_CONTIGUOUS the array elements are not contiguous (have spaces between them), IF4_FLG_AIV is an application-independent variable
int	occurrences[IF4_MAX_DIM]	array occurrences in each dimension
int	indexfactors[IF4_MAX_DIM]	array indexfactors for each dimension
void *	dynp	reserved for internal use

The address of an array element (i,j,k) is computed as follows (especially if the array elements are not contiguous):

elementaddress = address + i * indexfactors[0] + j * indexfactors[1] + k * indexfactors[2].
If the array has less than 3 dimensions, leave out the last terms.

 Copyright © Software AG 2002

INTERFACE4 - Parameter Access

A set of functions is available to be used for the access of the parameters. The process flow is as follows. The 3GL program is called via the CALL statement with the INTERFACE4 option, and the parameters are passed to the 3GL program as described above. The 3GL program can now use the exported functions of Natural, to retrieve either the parameter data itself, or information about the parameter, like format, length, array information, etc. The exported functions can also be used to pass back parameter data. With this technique a parameter access is guaranteed to avoid memory overwrites done by the 3GL program. (Natural's data is safe: memory overwrites within the 3GL program's data are still possible).

Exported Functions

- Get parameter information
- Get parameter data
- Write back operand data

Get parameter information

This function is used by the 3GL program to receive all necessary information from any parameter. This information is returned in the struct parameter_description, which is documented above.

Prototype:

```
int ncxr_get_parm_info ( int parmnum, void *parmhandle, struct parameter_description *descr );
```

Parameter description:

parmnum	Ordinal number of the parameter. This identifies the parameter of the passed parameter list. Range: 0 ... numparm-1.
parmhandle	the pointer to the internal parameter structure
descr	address of a struct parameter_description
return	0: OK -1 illegal parameter number -2 internal error -7 interface version conflict

Get parameter data:

This function is used by the 3GL program to get the data from any parameter. Natural identifies the parameter by the given parameter number and writes the parameter data to the given buffer address with the given buffer size. If the parameter data is longer than the given buffer size, Natural will truncate the data to the given length. The external 3GL program can make use of the function ncxr_get_parm_info, to request the length of the parameter data. There are two functions to get parameter data: ncxr_get_parm gets the whole parameter (even if the parameter is an array), whereas ncxr_get_parm_array gets the specified array element.

If no memory of the indicated size is allocated for "buffer" by the 3GL program (dynamically or statically) results of the operation are unpredictable. Natural will only check for a null pointer.

If data gets truncated for variables of the type I2/I4/F4/F8 (buffer length not equal to the total parameter length), the results depend on the machine type (little endian/big endian). In some applications, the user exit must be programmed to use no static data to make recursion possible.

Prototypes:

```
int ncxr_get_parm( int parmnum, void *parmhandle, int buffer_length, void *buffer )
```

```
int ncxr_get_parm_array( int parmnum, void *parmhandle, int buffer_length, void *buffer, int *indexes )
```

This function is identical to ncxr_get_parm, except that the indexes for each dimension can be specified. The indexes for unused dimensions should be specified as 0.

Parameter description:

parmnum	Ordinal number of the parameter. This identifies the parameter of the passed parameter list. Range: 0 ... numparm-1.
parmhandle	pointer to the internal parameter structure
buffer_length	length of the buffer, where the requested data has to be written to
buffer	address of buffer, where the requested data has to be written to. This buffer should be aligned to allow easy access to I2/I4/F4/F8 variables.
indexes	array with index information
return	Any value < 0 indicates an error during retrieval of the information. A value of -1 indicates an illegal parameter number. A value of -2 indicates an internal error. A value of -3 indicates that data has been truncated. A value of -4 indicates that data is not an array. A value of -7 indicates an interface version conflict. A value of -100 indicates that the index for dimension 0 is out of range. A value of -101 indicates that the index for dimension 1 is out of range. A value of -102 indicates that the index for dimension 2 is out of range. A value of 0 indicates successful operation. A value > 0 indicates successful operation, but the data was only this number of bytes long (buffer was longer than the data).

Write back operand data:

These functions are used by the 3GL program to write back the data to any parameter. Natural identifies the parameter by the given parameter number and writes the parameter data from the given buffer address with the given buffer size to the parameter data. If the parameter data is shorter than the given buffer size, the data will be truncated to the parameters data length, i.e., the rest of the buffer will be ignored. If the parameter data is longer than the given buffer size, the data will copied only to the given buffer length, the rest of the parameter stays untouched. This applies to arrays in the same way. For dynamic variables as parameters, the parameter is resized to the given buffer length.

If data gets truncated for variables of the type I2/I4/F4/F8 (buffer length not equal to the total parameter length), the results depend on the machine type (little endian/big endian). In some applications, the user exit must be programmed to use no static data to make recursion possible.

Prototypes:

```
int ncxr_put_parm        ( int parmnum, void *parmhandle,
                             int buffer_length, void *buffer );
  int ncxr_put_parm_array  ( int parmnum, void *parmhandle,
                             int buffer_length, void *buffer,
                             int *indexes );
```

 Copyright © Software AG 2002

Parameter description:

parmnum	Ordinal number of the parameter. This identifies the parameter of the passed parameter list. Range: 0 ... numparm-1.
parmhandle	pointer to the internal parameter structure.
buffer_length	length of the data to be copied back to the address of buffer, where the data comes from.
indexes	index information
return	Any value < 0 indicates an error during copying of the information: A value of -1 indicates an illegal parameter number. A value of -2 indicates an internal error. A value of -3 indicates that too much data has been given. The copy back was done with parameter length. A value of -4 indicates that the parameter is not an array. A value of -5 indicates that the parameter is protected (constant or AD=O). A value of -6 indicates that the dynamic variable could not be resized due to an 'out of memory' condition. A value of -7 indicates an interface version conflict. A value of -100 indicates that the index for dimension 0 is out of range A value of -101 indicates that the index for dimension 1 is out of range A value of -102 indicates that the index for dimension 2 is out of range A value of 0 indicates successful operation. A value > 0 indicates successful operation., but the parameter was this number of bytes long (length of parameter > given length)

All function prototypes are declared in the file natuser.h.

Part II: CALL under OpenVMS, UNIX and Windows

- Return Code
- User Exits under Windows
- User Exits under Open VMS
- User Exits under UNIX

Return Code

The condition code of any called function may be obtained by using the Natural system function RET.

Example:

```
...
RESET #RETURN(B4)
CALL 'PROG1'
IF RET ('PROG1') > #RETURN
  WRITE 'ERROR OCCURRED IN PROGRAM1'
END-IF
...
```

User Exits under Windows

Under Windows, user exits are needed to be able to access external functions that are invoked with a CALL statement. The user exits have to be placed in a DLL (dynamic link library). For further information on the user exits, please refer to the following file:

%NATDIR%\%NATVERS%\samples\sysexuex\readme.txt

User Exits under OpenVMS

Under OpenVMS, user exits are needed to be able to access external functions that are invoked with a CALL statement. The user exits have to be placed in a shareable image. For further information on the user exits, please refer to the following file:

NATSAMPLES:readme.txt

 Copyright © Software AG 2002

User Exits under UNIX

- Step 1 - Defining the Jump Table
- Step 2 - Writing the External Functions
- Step 3 - Compiling and Linking
- How to Build a Shared Library
- Using the Shared Library
- How to Generate a Static Nucleus
- Example Programs

Under UNIX, user exits are needed to make external functions available and to access operating-system interfaces that are not available to Natural.

The user exits can be placed either in a shared library and thus linked dynamically, or in a library that is linked statically to the Natural nucleus.

If they are placed in shared libraries, it is not necessary to relink Natural whenever a user exit is modified. This makes the development and testing of user exits a lot easier. This feature is available under all operating systems that support shared libraries.
Under all operating systems, it is possible to place user exits in a library that is linked to the Natural nucleus; that is, to statically link the user exits with the Natural prelinked object "natraw.o".

A user exit is added to Natural in three steps:

1. A jump table has to be created that allows Natural to associate the name of a function invoked by a CALL statement with the address of the function.
2. The functions that were put into the jump table must be written.
3. In the case of a dynamic link, the shared library that contains the user exits has to be rebuilt.
   In the case of a static link, the jump table and the external functions must be linked together with the prelinked Natural nucleus, to produce an executable Natural nucleus that supports the external functions.

Step 1 - Defining the Jump Table

A sample of a jump table - "jumptab.c" - can be found in the directory:

$NATDIR/$NATVERS/samples/sysexuex

Step 2 - Writing the External Functions

Each function has three parameters and returns a long integer. A function prototype should be as follows:

```
NATFCT myadd  (nparm, parmptr, parmdec)
```

```
WORD  nparm;
BYTE  **parmptr;
FINFO  *parmdec;
```

nparm	16 bit unsigned short value, containing the total number of transferred operands (operand2).
parmptr	Array of pointers, pointing to the transferred operands.
parmdec	Array of field information for each transferred operand.

The data type FINFO is defined as follows:

```
typedef struct {
  unsigned char     TypeVar;      /* type of variable                        */
  unsigned char     pb2;          /* if type == ('D', 'N', 'P' or 'T') ==>    */
                                  /*    total num of digits               */
                                  /* else                                     */
  union {                         /*    unused                            */
    unsigned char   pb[2];        /* if type == ('D', 'N', 'P' or 'T') ==>    */
    unsigned short  lfield;       /*    pb[0] = #dig before.dec.point     */
  } flen;                         /*    pb[1] = #dig  after.dec.point     */
                                  /* else                                 */
                                  /*    lfield = length of field          */
} FINFO;
```

Next, the module containing the external functions must be written. A sample function - "mycadd.c" - can be found in the directory:

$NATDIR/$NATVERS/samples/sysexuex

Step 3 - Compiling and Linking

The file "natuser.h", which is included by the sample program, is delivered with Natural. It contains declarations for the data types BYTE, WORD and the FINFO structure, that is, the description of the internal representation of each passed parameter.

- In the case of dynamically linked user exits, the shared library containing the user exits has to be rebuilt.
- In the case of statically linked user exits, the Natural nucleus has to be relinked.

For these purposes, it is strongly recommended to use the sample makefiles supplied by Software AG, as they already contain the necessary compiler and linker parameters. The sample makefiles can be found in the directory:

$NATDIR/$NATVERS/samples/sysexuex

For further information, see the following sections and the explanations in the makefiles themselves.

How to Build a Shared Library

1. From the example directory, which is contained in
   $NATDIR/$NATVERS/samples/sysexuex
   copy the following files into your work directory:
   Makedyn
   jumptab.c
   ncuxinit.c
2. Copy the C source files which contain your user exits into the same work directory.
3. Edit the file "jumptab.c" to include the names and function pointers for your user exits. To do so, you add in Section 2 the external declarations of your user exits, and in Section 3 you add the name/function-pointer pairs for your user exits. You might consider cutting and pasting the appropriate sections from your pre-2.2 version of "jumptab.c".
4. Edit the makefile as follows:
   Specify the names of the object files containing the user exits in the following line:
   USEROBJS =
   Specify the name of the resulting shared library in the following line:
   USERLIB =
   If you need to include private header files, specify the directories containing them in the following line:
   INCDIR =

 Copyright © Software AG 2002

5. To remove all unneeded files, issue the command:
   make -f Makedyn clean
6. To compile and link your shared library, issue the command:
   make -f Makedyn lib

Using the Shared Library

Set the environment variable NATUSER to the libraries you want to use. For example:

setenv NATUSER $NATDIR/$NATVERS/bin/<library-name>

You must specify a full qualified path name for the shared library.

You can specify more than one path if you delimit them with a colon (:) like the UNIX PATH variable.

Example:

See the sample user exit function in **$NATDIR/$NATVERS/samples/sysexuex**.

Note:
The libraries are searched in the order in which they are specified in NATUSER. This means that if two libraries contain a function of the same name, Natural always calls the function in the library which is specified first in NATUSER.

How to Generate a Static Nucleus

1. From the example directory, which is contained in **$NATDIR/$NATVERS/samples/sysexuex**
   copy the following files into your work directory:
   Makefile
   jumptab.c
2. Copy the C source files which contain your user exits into the same work directory.
3. Edit the file "jumptab.c" to include the names and function pointers for your user exits. To do so, you add in Section 2 the external declarations of your user exits, and in Section 3 you add the name/function-pointer pairs for your user exits. You might consider cutting and pasting the appropriate sections from your pre-2.2 version of "jumptab.c".
4. Edit the makefile as follows:
   Specify the names of the object files containing the user exits in the following line:
   USEROBJS =
   If you need to include private header files, specify the directories containing them in the following line:
   INCDIR =
5. Issue the command "make" to get information about further processing options.

Example:

See the sample user exit function in **$NATDIR/$NATVERS/samples/sysexuex**.

Example Programs:

After successful compilation and linking, the external programs can be invoked from a Natural program. Corresponding Natural example programs are provided in the library SYSEXUEX.

 Copyright © Software AG 2002

CALL FILE

Structured Mode Syntax

```
CALL FILE `program-name` operand1 operand2
    statement...
END-FILE
```

Reporting Mode Syntax

```
CALL FILE `program-name` operand1 operand2
    statement...
[LOOP]
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S	A			A	N	P	I	F	B	D	T	L	C			yes	yes
Operand2		S	A	G		A	N	P	I	F	B	D	T	L	C			yes	yes

Function

The CALL FILE statement is used to call a non-Natural program which reads a record from a non-Adabas file and returns the record to the Natural program for processing.

The CALL FILE statement initiates a processing loop which must be terminated with an ESCAPE or STOP statement. More than one ESCAPE statement may be specified to escape from a CALL FILE loop based on different conditions.

Restriction

The statements AT BREAK, AT START OF DATA and AT END OF DATA must not be used within a CALL FILE processing loop.

Copyright © Software AG 2002

Control Field - *operand1*

Operand1 is used to provide control information.

Record Area - *operand2*

Operand2 defines the record area.

The format of the record to be read can be described using field definitions (or FILLER nX) entries following the name of the first field in the record. The fields used to define the record format must not have been previously defined in the Natural program. This ensures that fields are allocated in the contiguous storage by Natural.

Example

Calling Program:

```
/* EXAMPLE 'CFIEX1': CALL FILE
/****************************
DEFINE DATA LOCAL
1 #CONTROL (A3)
1 #RECORD
 2 #A (A10)
 2 #B (N3.2)
 2 #FILL1 (A3)
 2 #C (P3.1)
END-DEFINE
/****************************
CALL FILE 'USER1' #CONTROL #RECORD
 IF #CONTROL = 'END'
  ESCAPE BOTTOM
 END-IF
END-FILE
/****************************
/* ... PROCESS RECORD ...
/****************************
END
```

The byte layout of the record passed by the called program to the Natural program in the above example is as follows:

```
CONTROL #A       #B   FILLER  #C
(A3)   (A10)   (N3.2)  3X   (P3.1)

xxx xxxxxxxxxx xxxxx   xxx   xxx
```

 Copyright © Software AG 2002

Called COBOL Program:

```
ID DIVISION.
PROGRAM-ID. USER1.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
     SELECT USRFILE ASSIGN UT-S-FILEUSR.
DATA DIVISION.
FILE SECTION.
FD   USRFILE RECORDING F LABEL RECORD OMITTED
     DATA RECORD DATA-IN.
01   DATA-IN         PIC X(80).
LINKAGE SECTION.
01   CONTROL-FIELD   PIC XXX.
01   RECORD-IN       PIC X(21).
PROCEDURE DIVISION USING CONTROL-FIELD RECORD-IN.
BEGIN.
     GO TO FILE-OPEN.
FILE-OPEN.
     OPEN INPUT USRFILE
     MOVE SPACES TO CONTROL-FIELD.
     ALTER BEGIN TO PROCEED TO FILE-READ.
FILE-READ.
     READ USRFILE INTO RECORD-IN
          AT END
          MOVE 'END' TO CONTROL-FIELD
          CLOSE USRFILE
          ALTER BEGIN TO PROCEED TO FILE-OPEN.
     GOBACK.
```

Copyright © Software AG 2002

CALL LOOP

Structured Mode Syntax

```
CALL LOOP operand1 [operand2]...40
    statement...
END-LOOP
```

Reporting Mode Syntax

```
CALL LOOP operand1 [operand2]...40
    statement...
[LOOP]
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S				A												yes	no
Operand2	C	S	A	G		A	N	P	I	F	B	D	T	L	C			yes	yes

Function

The CALL LOOP statement is used to generate a processing loop that contains a call to a non-Natural program.

Unlike the CALL statement, the CALL LOOP statement results in a processing loop which is used to repeatedly call the non-Natural program. See the CALL statement for detailed description of CALL processing.

Program Name - operand1

The name of the program to be called *(operand1)* can be specified as a constant or - if different programs are to be called dependent on program logic - as an alphanumeric variable of length 1 to 8. A program name must be placed left-justified in the variable.

 Copyright © Software AG 2002

Parameters - operand2

The CALL LOOP statement can have a maximum of 40 parameters. The parameter list is constructed as described for the CALL statement. Fields used in the parameter list may be initially defined in the CALL LOOP statement itself or may have been previously defined.

Loop Termination

The processing loop initiated with a CALL LOOP statement must be terminated with an ESCAPE statement.

Restriction

The statements AT BREAK, AT START OF DATA and AT END OF DATA must not be used within a CALL LOOP processing loop.

Example

```
DEFINE DATA LOCAL
1 PARAMETER1 (A10)
END-DEFINE
CALL LOOP 'ABC' PARAMETER1
   IF PARAMETER1 = 'END'
      ESCAPE BOTTOM
   END-IF
END-LOOP
END
```

CALLNAT

```
CALLNAT operand1 [[USING] { operand2 [(AD = { M } )] } ... ]
                          {          { O }   }
                          {          { A }   }
                          {   nX             }
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S				A												yes	no
Operand2	C	S	A	G		A	N	P	I	F	B	D	T	L	C	G	O	yes	yes

Related Statements: DEFINE DATA PARAMETER | FETCH | PERFORM

Function

The CALLNAT statement is used to invoke a Natural subprogram for execution.

When the CALLNAT statement is executed, the execution of the invoking object (that is, the object containing the CALLNAT statement) will be suspended and the invoked subprogram will be executed. The execution of the subprogram continues until either its END statement is reached or processing of the subprogram 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 statement.

Note:
A Natural subprogram can only be invoked via a CALLNAT statement; it cannot be executed by itself.

Subprogram Name - operand1

As *operand1*, you specify the name of the subprogram to be invoked. The name may be specified either as a constant of 1 to 8 characters, or - if different subprograms are to be called dependent on program logic - as an alphanumeric variable of length 1 to 8.

The subprogram name may contain an ampersand (&); at execution time, this character will be replaced by the current value of the system variable *LANGUAGE. This makes it possible, for example, to invoke different subprograms for the processing of input, depending on the language in which input is provided.

Parameters - operand2

If parameters are passed to the subprogram, the structure of the parameter list must be defined in a DEFINE DATA PARAMETER statement. The parameters specified with the CALLNAT statement are the only data available to the subprogram from the invoking object.

 Copyright © Software AG 2002

By default, the parameters are passed "by reference", that is, the data are transferred via address parameters, the parameter values themselves are not moved.
However, it is also possible to pass parameters "by value", that is, pass the actual parameter values. To do so, you define these fields in the DEFINE DATA PARAMETER statement of the subprogram with the option BY VALUE or BY VALUE RESULT as described under Parameter-Data-Definition in the section DEFINE DATA.

- If parameters are passed "by reference" the following applies: The sequence, format and length of the parameters in the invoking object must match exactly the sequence, format and length of the DEFINE DATA PARAMETER structure in the invoked subprogram. The names of the variables in the invoking object and the invoked subprogram may be different.
- If parameters are passed "by value" the following applies: The sequence of the parameters in the invoking object must match exactly the sequence in the DEFINE DATA PARAMETER structure of the invoked subprogram. Formats and lengths of the variables in the invoking object and the subprogram may be different; however, they have to be data transfer compatible (see the corresponding table in the Natural Reference documentation). The names of the variables in the invoking object and the subprogram may be different.
  If parameter values that have been modified in the subprogram are to be passed back to the invoking object, you have to define these fields with BY VALUE RESULT.
  With BY VALUE (without RESULT) it is not possible to pass modified parameter values back to the invoking object (regardless of the AD specification; see also below).

Note:
With BY VALUE, an internal copy of the parameter values is created. The subprogram accesses this copy and can modify it, but this will not affect the original parameter values in the invoking object.
With BY VALUE RESULT, an internal copy is likewise created; however, after termination of the subprogram, the original parameter values are overwritten by the (modified) values of the copy.

For both ways of passing parameters, the following applies:

If a group is specified as *operand2*, the individual fields contained in that group are passed to the subprogram; that is, for each of these fields a corresponding field must be defined in the subprogram’s parameter data area.

In the parameter data area of the invoked subprogram, a redefinition of groups is only permitted within a REDEFINE block.

If an array is passed, its number of dimensions and occurrences in the subprogram’s parameter data area must be the same as in the CALLNAT parameter list.

Note:
If multiple occurrences of an array that is defined as part of an indexed group are passed with the CALLNAT statement, the corresponding fields in the subprogram’s parameter data area must not be redefined, as this would lead to the wrong addresses being passed.

AD=

If operand2 is a variable, you can mark it in one of the following ways:

AD=O	non-modifiable
AD=M	modifiable
AD=A	input only

The default setting for AD= is AD=M.

If *operand2* is a constant, AD cannot be explicitly specified. For constants AD=O always applies.

AD=M

By default, the passed value of a parameter can be changed in the subprogram and the changed value passed back to the invoking object, where it overwrites the original value.

Exception: For a field defined with BY VALUE in the subprogram's parameter data area, no value is passed back.

AD=O

If you mark a parameter with AD=O, the passed value can be changed in the subprogram, but the changed value cannot be passed back to the invoking object; that is, the field in the invoking object retains its original value.

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=A

If you mark a parameter with AD=A, its value will not be passed `to` the subprogram, but it will receive a value `from` the subprogram. This may be useful for remote subprograms executed via Natural RPC in a client/server environment to reduce the load of data sent. If a subprogram is executed locally, AD=A fields will be reset to empty before the subprogram is invoked.

For a field defined with BY VALUE in the subprogram's parameter data area, the invoking object cannot receive a value. In this case, AD=A only causes the field to be reset to empty before the subprogram is invoked.

nX

Note:
This notation is not available on mainframe computers.

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 subprogram.
A parameter that is to be skipped must be defined with the keyword OPTIONAL in the subprogram's DEFINE DATA PARAMETER statement. OPTIONAL means that a value can - but need not - be passed from the invoking object to such a parameter.

Other Considerations

A subprogram can in turn invoke other subprograms.

A subprogram has no access to the global data area used by the invoking object.

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

 Copyright © Software AG 2002

Parameter Transfer with Dynamic Variables

Dynamic variables may be passed as parameters to a called program object (CALLNAT, PERFORM). Call-by-reference is possible because the value space of a dynamic variable is contiguous. 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. Call-by-value result causes additionally the movement in the opposite direction. In case of call-by-reference both definitions must be DYNAMIC. If only one of them is DYNAMIC, a runtime error is raised. In case of call-by-value (result) all combinations are possible. The following table illustrates the valid combinations of statically and dynamically defined variables of the caller and statically and dynamically defined parameters concerning the parameter transfer.

Call By Reference

Operand2 of Caller	**Parameter Definition**	
	Static	**Dynamic**
Static	YES	NO
Dynamic	NO	YES

The formats of the dynamic variables A or B must match.

Call by Value (Result)

Operand2 of Caller	**Parameter Definition**	
	Static	**Dynamic**
Static	YES	YES
Dynamic	YES	YES

Note:
In 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

Invoking Program:

```
/* EXAMPLE 'CNTEX1': CALLNAT
/*********************************************************************
/* MAIN PROGRAM 'MAINP1'
/*********************************************************************
DEFINE DATA LOCAL
1 #FIELD1 (N6)
1 #FIELD2 (A20)
1 #FIELD3 (A10)
END-DEFINE
/*********************************************************************
CALLNAT 'SUBP1' #FIELD1 (AD=M) #FIELD2 (AD=O) #FIELD3 'P4 TEXT'
/* ...
END
```

Invoked Subprogram:

```
/* SUBPROGRAM 'SUBP1'
/*********************************************************************
DEFINE DATA PARAMETER
1 #FIELDA (N6)
1 #FIELDB (A20)
1 #FIELDC (A10)
1 #FIELDD (A7)
END-DEFINE
/*********************************************************************
/* ...
END
```

 Copyright © Software AG 2002

Example 2

Invoking Program:

```
/* EXAMPLE 'CNTEX2': CALLNAT
/******************************
/* MAIN PROGRAM 'MAINP2'
/******************************
DEFINE DATA LOCAL
1 #ARRAY1  (A3/1:10,1:10)
END-DEFINE
CALLNAT 'SUBP2' #ARRAY1 (2:5,*)
/* ...
```

Invoked Subprogram:

```
/* SUBPROGRAM 'SUBP2'
/******************************
DEFINE DATA PARAMETER
1 #ARRAY (A3/1:4,1:10)
END-DEFINE
/******************************
/* ...
END
```

Copyright © Software AG 2002

CLOSE CONVERSATION

```
CLOSE CONVERSATION { operand1 ... }
                   { *CONVID      }
                   {     ALL      }
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S	A						I									yes	no

Related Statements: DEFINE DATA CONTEXT | OPEN CONVERSATION

Function

The statement CLOSE CONVERSATION is used in conjunction with Natural RPC. It allows the client to close conversations. You can close the current conversation, another open conversation, or all open conversations.

Note:
A logon to another library does not automatically close conversations.

Conversation to be Closed

operand1

To close a specific open conversation, you specify its ID as operand1. Operand1 must be a variable of format/length I4.

*CONVID

To close the current conversation, you specify *CONVID. The ID of the current conversation is determined by the value of the system variable *CONVID.

ALL

To close all open conversations, you specify ALL.

Further Information and Examples

See the Natural RPC documentation.

 Copyright © Software AG 2002

CLOSE DIALOG

Note:
This statement is only available under Windows and Windows NT.

```
CLOSE DIALOG [USING] [DIALOG-ID] { operand1 }
                                 { *DIALOG-ID }
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S							I									yes	no

Related Statement: OPEN DIALOG.

Function

This statement is used to close a dialog dynamically.

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.

Dialog to be Closed

operand1

Operand1 is the identifier of the dialog to be closed.

*DIALOG-ID

To close the current dialog, you specify *DIALOG-ID.

Further Information and Examples

See the section Event-Driven Programming Techniques in the Natural User's Guide for Windows.

CLOSE PC

This statement is only available with Natural Connection and is described in the Natural Connection documentation.

 Copyright © Software AG 2002

CLOSE PRINTER

```
CLOSE PRINTER { (logical-printer-name) }
              { (printer-number)       }
```

Function

The CLOSE PRINTER statement is used to close a specific printer. With the CLOSE PRINTER statement, you explicitly specify in a program that a printer is to be closed.

A printer is also closed automatically in one of the following cases:

- when a DEFINE PRINTER statement in which the same printer is defined again is executed;
- when command mode is reached.

Printer

With the logical-printer-name or printer-number you specify which printer is to be closed. The name and number are the same as in the corresponding DEFINE PRINTER statement in which you defined the printer. Naming conventions for the logical-printer-name are the same as for user-defined variables (see the general information for Reference). The printer-number may be a number in the range from 1 to 31.

Related Statement: DEFINE PRINTER.

Copyright © Software AG 2002

Example

```
* EXAMPLE 'CLPEX1': CLOSE PRINTER
*********************************
DEFINE DATA LOCAL
 1 EMP-VIEW VIEW OF EMPLOYEES
   2 PERSONNEL-ID
   2 NAME
   2 FIRST-NAME
   2 BIRTH
 1 I-NAME
END-DEFINE
*
DEFINE PRINTER (PRT01=1)
*
REPEAT
  INPUT 'SELECT PERSON' I-NAME
  IF I-NAME = ' '
    STOP
  FIND EMP-VIEW WITH NAME = I-NAME
    WRITE (PRT01) 'NAME         :' NAME ',' FIRST-NAME
            /      'PERSONNEL-ID :' PERSONNEL-ID
            /      'BIRTH        :' BIRTH (EM=YYYY-MM-DD)
   END-FIND
 *
   CLOSE PRINTER (PRT01)
 *
 END-REPEAT
 END
```

 Copyright © Software AG 2002

CLOSE WORK FILE

```
CLOSE WORK [FILE] work-file-number
```

Related Statements: READ WORK FILE | WRITE WORK FILE

Function

The CLOSE WORK FILE statement is used to close a specific work file. It allows you to explicitly specify in a program that a work file is to be closed.

A work file is also closed automatically when command mode is reached, or when an end-of-file condition occurs during the execution of a READ WORK FILE statement.

Work File

The work-file-number is the number of the work file (as defined to Natural) to be closed.

Automatic Closing

A work file is closed automatically:

- when command mode is reached,
- when an end-of-file condition occurs during the execution of a READ WORK FILE statement;
- before a DEFINE WORK FILE statement is executed which assigns another dataset to the work file number concerned.

Copyright © Software AG 2002

Example

```
/* EXAMPLE 'CWFEX1': CLOSE WORK FILE
DEFINE DATA LOCAL
01 W-DAT   (A20)
01 REC-NUM (N3)
01 I       (P3)
END-DEFINE
REPEAT
  READ WORK FILE 1 ONCE W-DAT  /* READ MASTER RECORD
  AT END OF FILE
    ESCAPE BOTTOM
  END-ENDFILE
  INPUT 'PROCESSING FILE' W-DAT (AD=O)
      / 'ENTER RECORDNUMBER TO DISPLAY' REC-NUM
  IF REC-NUM = 0
    STOP
  END-IF
  FOR I = 1 TO REC-NUM
    READ WORK FILE 1 ONCE W-DAT
    AT END OF FILE
      WRITE 'RECORD-NUMBER TOO HIGH, LAST RECORD IS'
      ESCAPE BOTTOM
    END-ENDFILE
  END-FOR
  I := I - 1
  WRITE 'RECORD' I ':' W-DAT
  CLOSE WORK FILE 1
END-REPEAT
END
```

 Copyright © Software AG 2002

COMPOSE

Note:
This statement can only be used if the Con-nect Office System (Version 2 or above) has been installed.

```
COMPOSE
    [RESETTING-clause]
    [MOVING-clause]
    [ASSIGNING-clause]
    [FORMATTING-clause]
    [EXTRACTING-clause]
```

If you specify more than one clause, they will be processed in the order shown above.

Function

The COMPOSE statement may be used to initiate text formatting by Con-form (the text formatter within Con-nect) directly from a Natural program.

The text to be formatted can either be supplied using variables or it may be retrieved from a Con-nect text block (a document containing Con-form formatting commands).

The contents of Natural variables can be passed to Con-form as variables for dynamic inclusion in the formatted text.

The values contained in a Con-form variable can also be returned to the Natural program from the text formatter.

When the Con-form instructions are completed (resulting in a formatted document), the output is passed to one of the following places:

- a Natural report,
- a document in the Con-nect system file,
- variables in the Natural program that executes the COMPOSE statement,
- a non-Natural program (on mainframe computers only).

Clauses

The RESETTING clause is used to delete information from the text format buffer area and to release memory from the COMPOSE buffer (on mainframe computers allocated by the CSIZE parameter in the Natural parameter module).

The MOVING clause is used to move text lines to the text formatter buffer area, or directly to the formatter, and to retrieve formatted text output from the work space of the formatter.

The ASSIGNING clause is used to assign the values of Natural variables to text variables.

The FORMATTING clause is used to create text in final formatted form, that is, with correct line and page breaks, using input which can be a combination of text and Con-form statements.

The EXTRACTING clause is used to assign the values of text variables to Natural variables.

Formatting Process

The formatting process begins when the FORMATTING clause of the COMPOSE statement is executed (even if text input via a MOVING clause is intended, but no such input has been provided yet).
While the formatting process is active, the text input resulting from the execution of the COMPOSE MOVING statement is fed directly into the formatter's work space (and cannot be re-used in a later formatting process).
If the formatting process is inactive, the text input is stored intermediately in the COMPOSE buffer (in the "DATAAREA"). Thus the input can be re-used for multiple formatting processes.

Since the formatter's buffer is not cleared at the end of the Natural program, the respective COMPOSE statements need not be executed within one Natural program; they can be issued in several successively invoked programs.

The execution of a RESETTING or FORMATTING clause, or a serious formatting error, causes the termination of an ongoing formatting pass.

End-of-input is specified by the LAST subclause of the MOVING clause.

When a Con-nect document is specified as the source of input, end-of-input is assumed when the end of that document is reached.

Note:
It is recommended to use the STATUS subclause of the FORMATTING or MOVING clause respectively, to make sure that the formatting process is always in the appropriate status for a given processing step.

Dialog Mode

Dialog Mode Processing is the set of interactions which are performed between a user program and the formatter while formatting input and producing output.

Dialog mode allows a user program to supply raw text as input to the formatter at any level of the input hierarchy. It also accepts formatted output directly in the current program environment.

The dialog is achieved by subdividing the formatting process into a series of steps, each of which is separately invoked by a COMPOSE statement.

- Dialog Mode for Input
- Dialog Mode for Output
- Dialog Mode for Input and Output
- Execution of COMPOSE Statements in Dialog Mode

Dialog Mode for Input

Dialog mode for input is entered if the source of the input text is DATAAREA, or if the formatting control statement ".TE ON" is encountered, and Con-form's data area does not contain any more text to be processed. Dialog mode for input is signalled by the word "TERM" in the first STATUS variable.

The user program should respond by supplying the required input by invoking the MOVING function in a subsequently-processed COMPOSE statement. The user program can terminate terminal input by specifying the LAST option of the MOVING clause, or ".TE OFF" if terminal input was invoked by ".TE ON", as text through the MOVING function. The formatter will signal the end of the formatting process with "END", or "ENDX" in the case of an error in the first status variable.

 Copyright © Software AG 2002

Dialog Mode for Output

Dialog mode for output is entered if the destination of the output is TO VARIABLES. The formatter passes control back to the Natural program environment as soon as the supplied Natural variables are filled or a page break is reached (whichever occurs first). Dialog mode for output is signalled with "STRG" in the first STATUS variable. The user program should respond by taking the formatted output just placed into the Natural variables and designate another set of Natural variables as the output destination in a subsequently processed COMPOSE MOVING statement. The end of the formatting process is indicated with "END", or "ENDX" in the case of an error.

Note:
When dialog mode is used (see the INPUT and OUTPUT subclauses), the formatting operation is usually spread across several executions of a COMPOSE statement.

Dialog Mode for Input and Output

Dialog mode can be entered for combined input and output processing. Therefore, when the formatter requests for further input (indicated by "TERM") or when the formatter provides output (indicated by "STRG"), the Natural program must take the appropriate action.

When dialog mode is entered for combined input and output processing, only one line of input is accepted by the formatter at a time. In the case of input mode only, multiple lines are accepted at one time.

Execution of COMPOSE Statements in Dialog Mode

While it has been pointed out that dialog mode is entered via a COMPOSE FORMATTING statement which encompasses a series of COMPOSE MOVING executions, please note the following:

- COMPOSE ASSIGNING and COMPOSE EXTRACTING statements are valid while dialog mode is active.
- COMPOSE RESETTING and FORMATTING will force the immediate termination of all formatting.

Copyright © Software AG 2002

Non-Natural Programs - only Mainframe

Depending on the parameters specified with the FORMATTING clause, input and output may be processed by non-Natural programs. Such programs are invoked by the same mechanism that is used within the CALL statement.

COMPOSE exchanges parameters with these programs using the standard linkage conventions (dynamic loading is not possible in a CICS environment).

Note:
Input/output processing by non-Natural programs is only possible on mainframe computers; on other platforms, the appropriate parts of the COMPOSE statement are ignored.

Depending on the status of the formatting process, two or three parameters are passed between the formatter and the non-Natural programs:

Parameter 1 (format/length A1)	Function code is passed from the formatter to non-Natural programs. Possible values: **I** - Initiate (input, output), **O** - Open document (input), **R** - Read one line of document (input), **W** - Write one line of output (output), **C** - Close document (input), **T** - Terminate (input, output).
Parameter 2 (format/length B1)	Response code is passed from non-Natural programs to the formatter. Possible values: **X'00'** - Function successfully completed. **X'01'** - In response to function "O": document could not be found. In response to function "R": end of document was reached. **X'FF'** - Function not completed.
Parameter 3 (format A1/256)	In the case of the functions "O" and "W", these parameters are passed from the formatter to non-Natural programs. However, the parameters from the function "R" are passed from non-Natural programs to the formatter. Bytes 1 - 2: Signify the length n of this parameter. Bytes 3 - 4: Empty. Bytes 5 - n: Function "O": Document name. Function "R": Line read by the non-Natural program. Function "W": Line of output from the formatter. Output is preceded by "N" if a form feed is required, otherwise by "1". Specific options for highlighting text such as boldface and italics are ignored if the output is passed to a non-Natural program.

 Copyright © Software AG 2002

RESETTING-clause

```
RESETTING [ DATAAREA  ]
          [ TEXTAREA  ]
          [ MACROAREA ]
          [ ALL       ]
```

This clause may be used to delete the following from the text format buffer area:

- DATAAREA deletes all active text variables.
- TEXTAREA deletes all text input data.
- MACROAREA deletes all text macros.
- ALL deletes all of the above.

Note:
For compatibility reasons, the keyword TEXTAREA refers to the formatter's "Data Area" as used in the MOVING clause.

MOVING-clause

Depending on the status of the dialog mode, one of the following forms of the MOVING clause may be used:

Syntax 1

```
MOVING [operand1]...37 [TO DATAAREA]
       [LAST] [STATUS [TO] operand2 [operand3 [operand4 [operand5]]]]
```

Syntax 2

```
MOVING {operand1 [TO DATAAREA]} [OUTPUT] TO VARIABLES operand6...20
       {LAST                  }
       [STATUS [TO] operand2 [operand3 [operand4 [operand5]]]]
```

Syntax 3

```
MOVING OUTPUT [TO VARIABLES] operand6...20
        [STATUS [TO] operand2 [operand3 [operand4 [operand5]]]]
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S	A			A	N	P									yes	no
Operand2		S				A											yes	yes
Operand3		S								B							yes	yes
Operand4		S								B							yes	yes
Operand5		S								B							yes	yes
Operand6		S	A			A											yes	no

Operand2 must be defined with format/length A4. Operand3, operand4, and operand5 must be defined with format/length B4.

This clause may be used to move one or more text values to the text format buffer area (Syntax 1). This area may be used as a source of input for formatting operations. If the text formatter is currently waiting for input (see Dialog Mode), the text will be passed directly to it without being stored in Con-form's text area (Syntax 1 and 2). The source input is terminated with the LAST option. If the formatted text is currently waiting for output (see Dialog Mode), Syntax 3 of the MOVING clause is used to pass control back from the Natural program to the formatter. For description of the status variables, see the FORMATTING clause.

Syntax 1

Syntax 1 of the MOVING clause is applicable when formatting has not begun or the formatter is in dialog mode for input and is waiting for input ("TERM" in the first status variable).

Syntax 2

Syntax 2 of the MOVING clause is applicable when the formatter is in dialog mode for both input and output, and is waiting for further input ("TERM" in the first status variable). The formatter will not accept more than one line of input in this mode.

The execution context may change between succession of executed COMPOSE statements. Therefore it is necessary to re-specify the output variables even when the formatter is waiting for input.

Syntax 3

Syntax 3 of the MOVING clause is applicable when the formatter is in dialog mode for output (and possibly for input at the same time), and is passing output to the Natural program ("STRG" in the first status variable).

 Copyright © Software AG 2002

ASSIGNING-clause

```
ASSIGNING [TEXTVARIABLE] {operand1 = operand2},...19
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S				A											yes	no
Operand2	C	S				A	N	P									yes	yes

This clause is used to assign values to Con-form text variables. These text variables may subsequently be referred to in formatting operations.

The text variable name(s) should be specified in upper case.

FORMATTING-clause

```
FORMATTING [ OUTPUT-subclause
             INPUT-subclause
             STATUS-subclause
             PROFILE-subclause
             MESSAGES-subclause
             ERRORS-subclause
             ENDING-subclause
             STARTING-subclause ]...
```

This clause causes Con-form to produce formatted output.

The formatting options are specified in one or more subclauses. If subclauses are omitted, Con-form will apply default formatting options. The status variable is used in dialog mode.

OUTPUT Subclause

The output medium. This can be a Natural report, a Con-nect cabinet, one or more Natural variables (or an array of Natural variables), or a non-Natural program.

INPUT Subclause

The input medium. This can be a Con-nect document, the COMPOSE data area (see the MOVING clause), the environment of the Natural program(s) executing the COMPOSE statement(s) (see the MOVING clause), a non-Natural program, or a mixture of these four possibilities.

STATUS Subclause

The status of the formatting operation. The formatting operation may involve multiple executions of a COMPOSE statement (in Dialog Mode). For example, the input is fed into the formatter's work space by a Natural program, and the output is passed from the formatter's work space into the environment of a Natural program (that is, one or more Natural variables). Therefore it is necessary to inform the Natural program of the formatting status. The following variables are passed to the Natural program during the formatting process:

- **State** - "TERM" when the dialog mode is ready for input. "STRG" when the dialog mode is ready for output. "END" if the formatting process was completed successfully. "ENDX" if the formatting process was completed unsuccessfully.
- **Position** - Page and line number of the document that is being formatted. The page and line numbers are kept separately in two variables (page position and line position).
- **Amount of Output Data** - The number of lines of formatted output which are being passed to the Natural program. The formatter uses this number as the pointer to the next output variable to be filled. The value is incremented by "1" before the output line is issued. If the current value is out of range, the value is set to "1".

PROFILE Subclause

Text block to be processed before input is processed.

 Copyright © Software AG 2002

MESSAGES/ERRORS Subclauses

Controls the output of warning messages and statistical information and error processing.

OUTPUT Subclause

```
                  (rep)
                SUPPRESSED
OUTPUT {       CALLING operand1                          }
       {TO VARIABLES [CONTROL operand2 operand3] operand4...20}
                DOCUMENT-option                           }
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S				A											yes	no
Operand2	C	S				A											yes	no
Operand3	C	S				A											yes	no
Operand4		S	A			A											yes	no

This subclause enables you to direct Con-form’s formatted text output to a specific destination.

If this subclause is omitted, Natural’s main printer will be used as the default output device.

OUTPUT - rep

If the output is directed to a printer (that is, the report number is not "0") and a Con-nect printer profile has been loaded (by the Con-nect API function Z-DRIVER), the settings of that profile will be used to control the text highlighting options of the formatted output text.

If a printer profile is active and the logical form feed controls were not specified, page ejects will be inserted by use of the appropriate internal Natural nucleus functions.

Any other highlighting text option which is not reflected in the currently active Con-nect printer profile will be ignored.

Note:
Executions of the COMPOSE RESETTING ALL or COMPOSE FORMATTING statement with non-report output destination will unload a printer profile from the formatter’s workspace.

If output is directed to report 0 or if a printer profile is not active, Con-nect will pass the responsibility of the output handling to the Natural nucleus routines. In this case, only the highlighting text options boldface, underline and italics will be recognized (applies only to mainframe computers; on other platforms, these text highlighting options will be ignored).

Note:
A report which is referred to in a DEFINE PRINTER (n) OUTPUT ’CONNECT’ statement must not be specified as output destination in a COMPOSE FORMATTING statement.

OUTPUT SUPPRESSED

This option causes the output to be SUPPRESSED.

OUTPUT CALLING

See the section Non-Natural Programs.

OUTPUT TO VARIABLES

Generally, the formatted text will be passed in final format to an array of Natural variables. Each line fills one variable (if necessary, the line may be truncated to fit into the variables). Text highlighting options will be ignored, with the exception of the CONTROL variables specified, which will be used to emphasize sections of the text (that is, boldface or underscore).

If the CONTROL variables, "I" and "N" are specified, the formatted text will be produced in an intermediate format (that is, with interspersed logical control sequences).

Operand2 and *operand3* must be of format/length A1.

For further information, see the section Dialog Mode.

DOCUMENT-option

```
DOCUMENT [INTO [ FINAL ] [CABINET] operand1 [PASSW = operand2] ]
                [ INTERMEDIATE ]
   [ [GIVING] { operand3 [operand 4] } ]
              { operand4 [operand 3] }
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S				A												yes	no
Operand2		S				A												yes	no
Operand3		S									B							yes	yes
Operand4		S									B							yes	yes

OUTPUT DOCUMENT

Operand3 (format/length B10) is used by the formatter to pass a unique key from the document back to the Natural program. It is supported for compatibility reasons only.

Operand4 (format/length B4) is used by the formatter to pass an ISN which points to the formatted output document back to the Natural program. This ISN can be useful when referencing the document in successive calls to Con-nect APIs.

 Copyright © Software AG 2002

If *operand1* (which may be up to 8 characters long) is not specified, the document will be added to the current user's cabinet (that is, to the cabinet whose ID is identical to the currently active Natural user ID).

A password (up to 8 characters) must be specified if storing the document in a cabinet to which the currently assumed user ID has no access.

Con-form enforces adherence to Con-nect access restrictions and only accepts cabinet IDs which have been defined to Con-nect.

Note:
Cabinet IDs must be specified in upper case.

The document will be added to the folder "COMPOSE" without a document name. The subject line will be filled with the name of the program executing the COMPOSE FORMATTING statement along with the date and time of execution.

If the keyword INTERMEDIATE has been omitted, the document will be created in final form text. In this case, specific text highlighting options such as boldface or italics will be ignored.

INPUT-subclause

```
INPUT { DATAAREA [FROM { EXIT operand2
                       { CABINETS operand2 [PASSW = operand3] }...9 ] }
      { operand1 FROM { EXIT operand2
                      { CABINETS operand2 [PASSW = operand3] }...10 }
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S				A											yes	no
Operand2	C	S				A											yes	no
Operand3		S				A											yes	no

This subclause may be used to specify the sources which will supply input for the text formatter. The input may be taken from Con-form's data area (a mixture of text from the data area and from the dialog mode is also possible) which must be filled by one or more MOVING operations, or from a text block (specified by *operand1*). The text block may be contained in a Con-nect cabinet, or it may be supplied by a non-Natural program. It will be invoked using the same conventions which apply to the CALL statement. A hierarchy of Con-nect cabinets or non-Natural programs may be specified, each of which will be scanned in turn for the text block specified in *operand1*.

A password must be specified if the document is stored in a cabinet to which the currently assumed user ID has no access.

Con-form enforces adherence to Con-nect access restrictions and only accepts cabinet IDs which have been defined to Con-nect.

If this subclause is omitted, the Con-form data area will be processed.

Note:
Cabinet and text block IDs must be specified in upper case.

STATUS-subclause

[**STATUS** *operand1* [*operand 2* [*operand 3* [*operand 4*]]]]

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1		S				A											yes	no
Operand2		S									B						yes	no
Operand3		S									B						yes	no
Operand4		S									B						yes	no

Operand 1 contains the Status variable "State".
Operand 2 contains the Status variable "Position (page number)".
Operand 3 contains the Status variable "Position (line number)".
Operand 4 contains the Status variable "Amount of Output Data".

 Copyright © Software AG 2002

PROFILE-subclause

PROFILE *operand1*

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S				A											yes	no

This subclause causes the content of the specified text block to be processed prior to any input which has been specified with the *INPUT-subclause* (by default, a text block will not be processed as a profile).

Copyright © Software AG 2002

MESSAGES-subclause

```
MESSAGES { [LISTED] [ON] (rep) }
         {   SUPPRESSED        }
```

Warning messages and statistical information are to be displayed upon completion of formatting. SUPPRESSED indicates that no messages are to be displayed and errors are to be ignored.

ERRORS-subclause

```
ERRORS { [LISTED] [ON] (rep) }
       {   INTERCEPTED       }
```

This subclause may be used to specify the actions to be performed when a formatting error occurs. The error may be simply ignored, it may be processed by Natural's standard error-processing routine, or it may be listed on a specified Natural report (*rep)*.

Note:
Errors and messages are mutually exclusive. Some errors may cause the standard Natural error-process routine to be invoked, even if a different option was specified.
Errors or messages must not be directed to a report which is directed to the Con-nect system by a DEFINE PRINTER (n) OUTPUT 'CONNECT' statement.

 Copyright © Software AG 2002

ENDING-subclause

```
ENDING { [AT] [PAGE] operand1      }
       { AFTER operand1 [PAGES]    }
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S					N	P									yes	no

This subclause causes output of formatted text to be suppressed following a page with a specified number, or alternatively, it limits the amount of formatted output to a specified number of pages.

STARTING-subclause

```
STARTING [FROM] [PAGE] operand1
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S					N	P									yes	no

This subclause causes output of formatted text to be suppressed until the page with the specified number (operand1) is reached.

EXTRACTING-clause

```
EXTRACTING [TEXTVARIABLE] {operand1 = operand2},...19
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1		S				A	N	P									yes	yes
Operand2	C	S				A											yes	no

This clause may be used to assign the values of text variables to Natural variables. The current text variable settings may be the result of previous formatting operations.

The text variable name(s) must be specified in upper case.

 Copyright © Software AG 2002

Example 1

```
COMPOSE RESETTING ALL
        FORMATTING INPUT 'TEXT' FROM CABINET 'TLIB'
        OUTPUT (1)
        MESSAGES LISTED ON (0)
```

The above COMPOSE statement results in a formatted output of the text block TEXT within the Con-nect cabinet TLIB which is produced on report 1. Errors and statistical messages are displayed on report 0 (the default printer).

Example 2

```
COMPOSE RESETTING ALL
COMPOSE MOVING '.FI ON' 'This is an example'
COMPOSE MOVING 'for use of Con-form from'
               'within Natural applications' LAST
COMPOSE FORMATTING
```

The above COMPOSE statements result in a formatted output of text on report 0 (default printer).

Example 3

```
COMPOSE ASSIGNING 'VAR1' = 'Text1', 'VAR2' = 540
```

The above COMPOSE statement results in the assignment of values to Con-form text variables &VAR1 and &VAR2 in a Con-nect procedure.

Example 4

Text Block "XYZ" in "XYLIB":

```
.FI ON
Dear Mr &name.,
.IL
I am pleased to invite you to a presentation of our new product &prod..
```

Natural Program:

```
...
INPUT #NAME (A32) #PROD (A32)
COMPOSE ASSIGNING 'NAME' = #NAME, 'PROD' = #PROD
        FORMATTING INPUT 'XYZ' FROM CABINET 'XYLIB'
        OUTPUT (1) MESSAGES SUPPRESSED
...
```

Input Map produced by Program:

```
#NAME Davenport
  #PROD NATURAL 2.2
```

Resulting Output:

```
Dear Mr Davenport,

  I am pleased to invite you to a presentation of our new product NATURAL 2.2.
```

Example 5

This is an example of formatting in dialog mode with combined input/output handling. The example program initiates the line-oriented formatting mode of Con-form, passes some commands/variables to Con-form, and performs a subroutine which displays status information and formatted output lines on the screen.

```
DEFINE DATA LOCAL
01 #LINES_PER_PERFORM(P5)   /* counts repeat-loops per PERFORM CNF_OUT
01 #TRACE(A1) INIT<'N'>     /* if 'Y' displays additional trace-infos
01 #OUT_FORM(A1) INIT<'F'>  /* output-format
01 #START_PAGE (P3) INIT<1> /* beginning of display
01 #CNTR (P5)               /* Loop-Counter
01 #STATI                   /* Status-Information
 02 #STATUS   (A4)          /* can be STRG TERM END or ENDX
 02 #PAGE     (B4)          /* actual page-number
 02 #LINE     (B4)          /* actual line-number on page (not .tt/.bt)
 02 #NO_LINES (B4)          /* number of lines returned
 02 REDEFINE #NO_LINES
  03 #NO_LINES_I (I4)
01 #OUTPUT(A30/4)           /* output of formatted line
01 #INDEX (P3)              /* index as pointer to out line
END-DEFINE
*
SET KEY ALL
```

 Copyright © Software AG 2002

```
SET CONTROL 'M9'
INPUT
   008/008 'Demonstration of formatted output to Variable'(I)
   /   08X 'Enter page to start display    :' #START_PAGE(AD=MIL)
   /   08X 'Display additional trace-data ?:' #TRACE(AD=MIT)
   /   08X 'Please select the output-format:' #OUT_FORM(AD=MIT)
           '(F=Final without BP/US-marks'
   /   44X 'M=Final with BP/US-marks "<>"'
   /   09X '(only, if CMF-Zap 2056 applied =>) I=Intermediate)'
   /   50X 'PF3=Exit'(I)
*
IF *PF-KEY EQ 'PF3'
   SET CONTROL 'MB'
   STOP
END-IF
*
IF NOT (#OUT_FORM EQ 'F' OR EQ 'M' OR EQ 'I')
   REINPUT ' Please enter valid code!' MARK *#OUT_FORM ALARM
END-IF
*
WRITE TITLE LEFT
     'Stat * Page     * Line     * No.Lines >> Formatted Output'(I)
   / '-'(79)(I)
*
SET CONTROL 'MB'
COMPOSE RESETTING ALL /* clear all con-form buffers
RESET #NO_LINES
*
* start line-oriented formatting-mode here
* starting from 0
DECIDE ON FIRST VALUE OF #OUT_FORM
   VALUE 'F'
      COMPOSE FORMATTING
           OUTPUT TO VARIABLES #OUTPUT (1:4)      /* to Output
           STATUS #STATUS #PAGE #LINE #NO_LINES   /* get Status
   VALUE 'M'
      COMPOSE FORMATTING
           OUTPUT TO VARIABLES CONTROL '<' '>'
                               #OUTPUT (1:4)      /* to output
           STATUS #STATUS #PAGE #LINE #NO_LINES   /* get Status
   VALUE 'I'
      COMPOSE FORMATTING
           OUTPUT TO VARIABLES CONTROL 'I' 'N'
                               #OUTPUT (1:4)      /* to output
           STATUS #STATUS #PAGE #LINE #NO_LINES   /* get Status
   NONE
      STOP
END-DECIDE
*
RESET #NO_LINES
*
* put some commands to con-form to see something
*
COMPOSE MOVING
     '.pl 16;.hs 2;.tt 1Formatierung in Variable//;.tt 2//' /* Cmd
     OUTPUT TO VARIABLES #OUTPUT (1:4)           /* to Output
     STATUS #STATUS #PAGE #LINE #NO_LINES        /* get Status
*
COMPOSE MOVING
     '.fs 1;.bt Ende Seite #//;.fi on;.tb *=15' /* Commands
     OUTPUT TO VARIABLES #OUTPUT (1:4)           /* to Output
     STATUS #STATUS #PAGE #LINE #NO_LINES        /* get Status
```

```
*
*
* loop 40-times, send commands to con-form and display output
*
COMPOSE ASSIGNING 'Wert' = '1-20' /* Assign value to variable &Wert
*
FOR #CNTR 1 40                      /* Loop some time
   IF #STATUS NE 'TERM' /* no wait-for-input => error!!!!
      IF #STATUS EQ 'STRG'
         IGNORE
      ELSE
         WRITE 'Unexpected Status-code' #STATUS(AD=OI) 'found!'
             / 'Execution has stopped....'
         STOP
      END-IF
   END-IF
*
   IF #CNTR EQ 21
      COMPOSE ASSIGNING 'Wert' = '21-40' /* Assign a variable-value
   END-IF
   COMPOSE ASSIGNING 'CNTR' = #CNTR /* Again assignment
   COMPOSE MOVING
        '.BP;&Wert *Durchlauf &CNTR;.BR'      /* Commands
        OUTPUT TO VARIABLES #OUTPUT (1:4)     /* to Output
        STATUS #STATUS #PAGE #LINE #NO_LINES  /* get Status
   PERFORM CNF-OUT                            /* show result
END-FOR
COMPOSE MOVING
     LAST                                 /* End-Of-Processing
     OUTPUT TO VARIABLES #OUTPUT (1:4)    /* to Output
     STATUS #STATUS #PAGE #LINE #NO_LINES /* get Status
*
IF #TRACE EQ 'Y'
   WRITE 'End of processing...'(I)
END-IF
*
* Subroutines
*
PERFORM CNF-OUT
*
* Subroutine to display any waiting output from con-form
*



DEFINE SUBROUTINE CNF-OUT
   RESET #LINES_PER_PERFORM
   REPEAT UNTIL #STATUS EQ 'TERM' /* TERM = input waiting
      PERFORM BREAK               /* do some break-processing
      AT BREAK OF #PAGE
         IF #PAGE GT #START_PAGE
            WRITE '-'(79)(I)
         END-IF
         IF #TRACE EQ 'Y'
            WRITE 'End of this page...'(I)
         END-IF
         NEWPAGE
      END-BREAK
      IF #PAGE GE #START_PAGE     /* show line of output
         IF #NO_LINES_I GT 0
            FOR #INDEX 1 #NO_LINES_I
               ADD 1 TO #LINES_PER_PERFORM /* count loops
```

 Copyright © Software AG 2002

```
                WRITE NOTIT NOHDR #STATUS '*' #PAGE '*' #LINE
                                          '*' #NO_LINES
                                          '>>' #OUTPUT (#INDEX)
             END-FOR
          END-IF
       END-IF
       IF #STATUS NE 'STRG'         /* if no wait on out
          ESCAPE BOTTOM
       END-IF
       RESET #NO_LINES
       COMPOSE MOVING
            OUTPUT TO VARIABLES #OUTPUT (1:4)    /* get Output
            STATUS #STATUS #PAGE #LINE #NO_LINES /* Status
   END-REPEAT
*
   IF #TRACE EQ 'Y'
    WRITE 'Count of Lines per PERFORM was'(I) #LINES_PER_PERFORM(AD=OI)
   END-IF
*
END-SUBROUTINE
   SET CONTROL 'MB'
END
```

COMPRESS

```
                COMPRESS [NUMERIC] [FULL]

{                   operand1  [(parameter)]                    }
{SUBSTRING (operand1,operand3,operand4)  [(parameter)]}...

     INTO {                operand2                  }
          {SUBSTRING (operand2,operand5,operand6)}

          [                LEAVING SPACE                 ]
          [              LEAVING NO [SPACE]              ]
          [WITH [ALL] [DELIMITERS] [operand7]            ]
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S	A	G	N	A	N	P	I	F	B	D	T			G	O	yes	no
Operand2		S				A												yes	yes
Operand3	C	S					N	P	I									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
Operand7	C	S				A												yes	no

Related Statements: EXAMINE | SEPARATE

Function

The COMPRESS statement is used to transfer (combine) the contents of two or more operands into a single alphanumeric field.

 Copyright © Software AG 2002

Source Fields - operand1

As *operand1*, you specify the fields whose contents are to be transferred.

Note:
If operand1 is a time variable (format T), only the time component of the variable content is transferred, but not the date component.

Target Field - operand2

As *operand2*, you specify the field which is to receive the values of the source fields.

If you use the COMPRESS statement without any further options, or if you specify LEAVING SPACE (which also applies by default), the values in the target field will be separated from one another by a blank.

If you specify LEAVING NO SPACE, the values in the target field will `not` be separated from one another by a blank or any other character.

FULL

Without FULL, leading zeros (in numeric fields) and trailing blanks (in alphanumeric fields) are removed from the source fields before the values are transferred. For a numeric source field containing all zeros, one "0" will be transferred.

With FULL, the values of the source fields in their actual lengths - that is, including leading zeros and trailing blanks - will be transferred to the target field.

Examples:

```
1.    COMPRESS 'ABC '  001 INTO #TARGET WITH DELIMITER '*'
    Content of #TARGET is: ABC*1
2.    COMPRESS FULL 'ABC '  001 INTO #TARGET WITH DELIMITER '*'
    Content of #TARGET is: ABC *001
```

NUMERIC

This option determines how sign characters and decimal characters are to be handled:

- Without NUMERIC, decimal points and signs in numeric source values are suppressed before the values are transferred.
- With NUMERIC, decimal points and signs in numeric source values are also transferred to the target field.

Examples:

```
1.    COMPRESS -123  1.23 INTO #TARGET WITH DELIMITER '*'
    Content of #TARGET is: 123*123
2.    COMPRESS NUMERIC -123  1.23 INTO #TARGET WITH DELIMITER '*'
    Content of #TARGET is: -123*1.23
```

parameter

As parameter, you can specify the option "PM=I" or the session parameter DF:

PM=I

In order to support languages whose writing direction is from right to left, you can specify "PM=I" so as to transfer the value of *operand1* in inverse (right-to-left) direction to *operand2*.

For example, as a result of the following statements, the content of #B would be "ZYXABC":

```
MOVE 'XYZ' TO #A
COMPRESS #A (PM=I) 'ABC' INTO #B LEAVING NO SPACE
```

Any trailing blanks in *operand1* will be removed (except if FULL is specified), then the value is reversed character by character and transferred to *operand2*.

DF

If *operand1* is a date variable, you can specify the session parameter DF as *parameter* for this variable.

SUBSTRING

If *operand1* is of alphanumeric format, you can use the SUBSTRING option to transfer only a certain part of a source field.

Also, you can use the SUBSTRING option in the INTO clause to transfer source values into a certain part of the target field.

 Copyright © Software AG 2002

In both cases, the use of the SUBSTRING option in a COMPRESS statement corresponds to that in a MOVE statement. See the MOVE statement for details on the SUBSTRING option.

WITH DELIMITER - operand7

If you wish the values in the target field to be separated from one another by a specific character, you use the DELIMITER option:

- If you specify WITH DELIMITER *operand7*, the values will be separated by the character specified with *operand7*. *Operand7* must be a single character. If *operand7* is a variable, it must be of format/length A1.
- If you specify WITH DELIMITERS without *operand7*, the values will be separated by the input delimiter character (as defined with the session parameter ID).

ALL

Without ALL, a delimiter is placed in the target field only between values actually transferred.

With ALL, a delimiter is also placed in the target field for each blank value that is not actually transferred. This means that the number of delimiters in the target field corresponds to the number of source fields minus 1. This may be useful, for example, if the content of the target field is to be separated again with a subsequent SEPARATE statement.

Examples:

```
1.    COMPRESS 'A' ' ' 'C' ' ' INTO #TARGET WITH DELIMITER '*'
    Content of #TARGET is: A*C
```

```
2.    COMPRESS 'A' ' ' 'C' ' ' INTO #TARGET WITH ALL DELIMITERS '*'
    Content of #TARGET is: A**C*
```

Processing

The COMPRESS operation terminates when either all operands have been processed or the target field (*operand2*) is filled.

If the target field contains more positions than all operands combined, all remaining positions of operand2 will be filled with blanks. If the target field is shorter, the value will be truncated.

If *operand2* is a DYNAMIC variable, the COMPRESS operation terminates when all source operands have been processed. No truncation will be performed. The length of *operand2* after the COMPRESS operation will correspond to the combined length of the source operands. The current length of a DYNAMIC variable can be ascertained by using the system variable *LENGTH. For general information on DYNAMIC variables, see your Natural User's Guide.

Example 1

```
/* EXAMPLE 'CMPEX1S:' COMPRESS (STRUCTURED MODE)
/************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 MIDDLE-I
1 #COMPRESSED-NAME (A20)
END-DEFINE
/************************************************************
LIMIT 4
READ EMPLOY-VIEW BY NAME
 COMPRESS FIRST-NAME MIDDLE-I NAME INTO #COMPRESSED-NAME
 DISPLAY NOTITLE FIRST-NAME MIDDLE-I NAME 5X #COMPRESSED-NAME
END-READ
/************************************************************
END
```

```
   FIRST-NAME      MIDDLE-I         NAME                #COMPRESSED-NAME
-------------------- -------- --------------------     --------------------

KEPA                          ABELLAN                  KEPA ABELLAN
ROBERT               W        ACHIESON                 ROBERT W ACHIESON
SIMONE                        ADAM                     SIMONE ADAM
TIMMIE               D        ADKINSON                 TIMMIE D ADKINSON
```

Equivalent reporting-mode example: See program CMPEX1R in library SYSEXRM.

Example 2

```
/* EXAMPLE 'CMPEX2': COMPRESS LEAVING NO SPACE
/************************************************************
LIMIT 4
READ EMPLOYEES BY NAME
 COMPRESS CURR-CODE (1) SALARY (1) INTO #CCSALARY (A20)
          LEAVING NO SPACE
 DISPLAY NOTITLE NAME CURR-CODE (1) SALARY (1) 5X #CCSALARY
/************************************************************
END
```

```
      NAME          CURRENCY   ANNUAL            #CCSALARY
                       CODE     SALARY
-------------------- -------- ----------     --------------------

ABELLAN               PTA         1450000     PTA1450000
ACHIESON              UKL           10500     UKL10500
ADAM                  FRA          159980     FRA159980
ADKINSON              USD           36000     USD36000
```

 Copyright © Software AG 2002

Example 3

```
/* EXAMPLE 'CMPEX3': COMPRESS WITH DELIMITER
/*************************************************************
LIMIT 4
READ EMPLOYEES BY NAME
 COMPRESS CURR-CODE (1) SALARY (1) INTO #CCSALARY (A20)
          WITH DELIMITER '*'
 DISPLAY NOTITLE NAME CURR-CODE (1) SALARY (1) 5X #CCSALARY
/*************************************************************
END
```

```
      NAME          CURRENCY   ANNUAL            #CCSALARY
                       CODE     SALARY
-------------------- -------- ----------     --------------------

ABELLAN                PTA        1450000     PTA*1450000
ACHIESON               UKL          10500     UKL*10500
ADAM                   FRA         159980     FRA*159980
ADKINSON               USD          36000     USD*36000
```

COMPUTE

Structured Mode Syntax

```
{ {COMPUTE} [ROUNDED] {operand1[:]=}...{arithmetic-expression}
  {ASSIGN}            {operand1:=}...{operand2              }
```

Reporting Mode Syntax

```
[ {COMPUTE} [ROUNDED] ] {operand1[:]=}...{arithmetic-expression}
  {ASSIGN}              {operand1:=}...{operand2              }
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S	A		M	A	N	P	I	F	B	D	T	L	C	G	O	yes	yes
Operand2	C	S	A		N	A	N	P	I	F	B	D	T	L	C	G	O	yes	no

Related Statements: ADD | SUBTRACT | MULTIPLY | DIVIDE | MOVE

Function

The COMPUTE statement is used to perform an arithmetic or assignment operation.

This statement may be issued in short form by omitting the statement keyword COMPUTE (or ASSIGN). In structured mode, when the statement keyword is omitted, the equal sign (=) must be preceded by a colon (:). However, when the ROUNDED option is used, the statement keyword COMPUTE or ASSIGN must be specified.

For arithmetic operations involving arrays, see also the section Arithmetic Operations with Arrays in the Reference part of the documentation.

Result Field - operand1

Operand1 will contain the result of the arithmetic/assignment operation.

For the precision of the result, see the section Rules for Arithmetic Assignment in the Reference part of the documentation.

 Copyright © Software AG 2002

If *operand1* is a database field, the field in the database is not updated.

If *operand1* is a DYNAMIC variable, it will be filled up to the length of *operand2* or the length of the result of the *arithmetic-operation*, and the length of *operand1* will then be adjusted accordingly. The current length of a DYNAMIC variable can be ascertained by using the system variable *LENGTH. For general information on DYNAMIC variables, see your Natural User's Guide.

ROUNDED

If you specify the keyword ROUNDED, the value will be rounded before it is assigned to *operand1*. For information on rounding, see the section Rules for Arithmetic Assignment in the Reference part of the documentation.

arithmetic-expression

An arithmetic expression consists of one or more constants, database fields, and user-defined variables.

Natural mathematical functions (which are described in the section System Functions in the Reference part of the documentation) may also be used as arithmetic operands.

Operands used in an arithmetic expression must be defined with format N, P, I, F, D, or T.

As for the formats of the operands, see also the section Performance Considerations for Mixed Formats in the Reference part of the documentation.

The following connecting operators may be used:

Operator	Symbol
Parentheses	( )
Exponentiation	**
Multiplication	*
Division	/
Addition	+
Subtraction	-

Each operator should be preceded and followed by at least one blank so as to avoid any conflict with a variable name that contains any of the above characters.

The processing order of arithmetic operations is:

1. Parentheses
2. Exponentiation
3. Multiplication/division (left to right as detected)
4. Addition/subtraction (left to right as detected).

Result Precision of a Division

The precision (number of decimal positions) of the result of a division in a COMPUTE statement is determined by the precision of either the first operand (dividend) or the first result field, whichever is greater.

For a division of integer operands, however, the following applies: For a division of two integer constants, the precision of the result is determined by the precision of the first result field; however, if at least one of the two integer operands is a variable, the result is also of integer format (that is, without decimal positions, regardless of the precision of the result field).

SUBSTRING

If the operands are of alphanumeric format, you may use the SUBSTRING option in the same manner as described for the MOVE statement to assign a part of operand2 to operand1.

Example 1

```
/* EXAMPLE 'ASGEX1S': ASSIGN (STRUCTURED MODE)
DEFINE DATA LOCAL
  1 #A (N3)
  1 #B (A6)
  1 #C (N0.3)
  1 #D (N0.5)
  1 #E (N1.3)
  1 #F (N5)
  1 #G (A25)
  1 #H (A3/1:3)
END-DEFINE
/*********************************************************************
ASSIGN #A = 5                                  WRITE NOTITLE '=' #A
ASSIGN #B = 'ABC'                              WRITE '=' #B
ASSIGN #C = .45                                WRITE '=' #C
ASSIGN #D = #E = -0.12345                      WRITE '=' #D / '=' #E
ASSIGN ROUNDED #F = 199.999                    WRITE '=' #F
#G     := 'HELLO'                              WRITE '=' #G
#H (1) := 'UVW'
#H (3) := 'XYZ'                                WRITE '=' #H (1:3)
END
```

```
#A:    5
  #B: ABC
  #C:  .450
  #D: -.12345
  #E: -0.123
  #F:    200
  #G: HELLO
  #H: UVW     XYZ
```

Equivalent reporting-mode example: See program ASGEX1R in library SYSEXRM.

Example 2

 Copyright © Software AG 2002

```
/* EXAMPLE 'CPTEX1S': COMPUTE (STRUCTURED MODE)
/************************************************************
DEFINE DATA LOCAL
1 #I (P2)
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 SALARY (1:2)
1 #A (P4)
1 #B (N3.4)
1 #C (N3.4)
1 #CUM-SALARY (P10)
END-DEFINE
/************************************************************
COMPUTE #A = 3 * 2 + 4 / 2 - 1
WRITE NOTITLE 'COMPUTE #A = 3 * 2 + 4 / 2 - 1' 10X '=' #A
/************************************************************
COMPUTE ROUNDED #B = 3 - 4 / 2 * .89
WRITE   'COMPUTE ROUNDED #B = 3 -4 / 2 * .89' 5X '=' #B
/************************************************************
COMPUTE #C = SQRT (#B)
WRITE 'COMPUTE #C = SQRT (#B)' 18X '=' #C
/************************************************************
LIMIT 1
READ EMPLOY-VIEW BY PERSONNEL-ID STARTING FROM '20017000'
 WRITE / 'CURRENT SALARY: '  4X SALARY (1)
       / 'PREVIOUS SALARY:'  4X SALARY (2)
 FOR #I = 1 TO 2
  COMPUTE #CUM-SALARY = #CUM-SALARY + SALARY (#I)
 END-FOR
 WRITE 'CUMULATIVE SALARY:' #CUM-SALARY
END-READ
/************************************************************
END
```

```
COMPUTE #A = 3 * 2 + 4 / 2 - 1          #A:     7
  COMPUTE ROUNDED #B = 3 -4 / 2 * .89     #B:    1.2200
  COMPUTE #C = SQRT (#B)                  #C:    1.1045

  CURRENT SALARY:          34000
  PREVIOUS SALARY:         32300
  CUMULATIVE SALARY:       66300
```

Equivalent reporting-mode example: See program CPTEX1R in library SYSEXRM.

CREATE OBJECT

```
CREATE  OBJECT  operand1 OF  [CLASS ] operand2
                             [ON  [NODE ] operand3]
                             [GIVING  operand4]
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1		S														O	no	no
Operand2	C	S				A											yes	no
Operand3	C	S				A											yes	no
Operand4		S			N				I								yes	no

Function

The CREATE OBJECT statement is used to create an instance of a class. When a CREATE OBJECT statement is executed, NaturalX checks if the name of the class specified in the statement is registered. If this is the case, it creates the object using DCOM. If this is not the case, it searches for a class with that name in the current Natural library or in the steplibs and creates the object locally.

Object Handle - operand1

Operand1 must be defined as an object handle (HANDLE OF OBJECT).

The object handle is filled when the object is successfully created. When not successfully returned, *operand1* contains the value NULL-HANDLE.

Class-Name - operand2

Operand2 is the name of the class of which the object is to be created. For classes that are not registered, it must contain the class name defined in the DEFINE CLASS statement. For classes that are registered, it must contain either the ProgID of the class or the class GUID. For Natural classes, the ProgID corresponds to the class name specified in the DEFINE CLASS statement. For further information, see the section Registration with Natural.

```
CREATE OBJECT #O1 OF CLASS "Employee" or
  CREATE OBJECT #O1 OF CLASS "653BCFE0-84DA-11D0-BEB3-10005A66D231"
```

 Copyright © Software AG 2002

Node - operand3

As *operand3* you specify the node where the object is created. This is only possible if the class is registered. If the node clause is specified, an attempt is made to create the object on that node. If the node clause is not specified or contains a blank value, the object is created on the node that is specified in the system registry under the key "RemoteServerName" for that class. If this registry key is not specified, the object is created in the local Natural session. For example

```
CREATE OBJECT #O1 OF CLASS "Employee" ON NODE "volcano.iceland.com"
```

GIVING - operand4

If the GIVING clause is specified, *operand4* contains either the Natural message number if an error occurred, or zero on success.

If the GIVING clause is not specified, Natural run time error processing is triggered if an error occurs.

DECIDE FOR

```
DECIDE FOR {FIRST} CONDITION
           {EVERY}
      {WHEN logical-condition statement..}...
      [WHEN ANY statement...]
      [WHEN ALL statement...]
      WHEN NONE statement...
END-DECIDE
```

Related Statements: DECIDE ON | IF

Function

The DECIDE FOR statement is used to decide for one or more actions depending on multiple conditions (cases).

Note:
If **no** action is to be performed under a certain condition, you specify the statement IGNORE in the corresponding clause of the DECIDE FOR statement.

FIRST/EVERY

With the keyword FIRST or EVERY, you indicate whether only the first or every true condition is to be processed.

WHEN logical-condition

With this clause, you specify the *logical condition(s)* to be processed. See the section Logical Condition Criteria in the Natural Reference documentation.

WHEN ANY

With WHEN ANY, you can specify the *statement(s)* to be executed when *any* of the logical conditions are true.

WHEN ALL

With WHEN ALL, you can specify the *statement(s)* to be executed when *all* logical conditions are true. This clause is applicable only if EVERY has been specified.

 Copyright © Software AG 2002

WHEN NONE

With WHEN NONE, you specify the *statement(s)* to be executed when *none* of the logical conditions are true.

Example 1

```
/* EXAMPLE 'DECEX1:' DECIDE FOR (USING FIRST OPTION)
/*****************************************************
/* IF FUNCTION = A AND PARM = X
/*    ROUTINE-A IS TO BE EXECUTED.
/* IF FUNCTION = B AND PARM = X
/*    ROUTINE-B IS TO BE EXECUTED.
/* IF FUNCTION = C THRU D
/*    ROUTINE-CD IS TO BE EXECUTED.
/* FOR ALL OTHER CASES,
/*    REINPUT STATEMENT IS TO BE EXECUTED.
/*****************************************************
DEFINE DATA LOCAL
1 #FUNCTION (A1)
1 #PARM (A1)
END-DEFINE
/*****************************************************
INPUT #FUNCTION #PARM
/*****************************************************
DECIDE FOR FIRST CONDITION
           WHEN #FUNCTION = 'A' AND #PARM = 'X'
             PERFORM ROUTINE-A
           WHEN #FUNCTION = 'B' AND #PARM = 'X'
             PERFORM ROUTINE-B
           WHEN #FUNCTION = 'C' THRU 'D'
             PERFORM ROUTINE-CD
           WHEN NONE
             REINPUT 'PLEASE ENTER A VALID FUNCTION'
                     MARK *#FUNCTION
END-DECIDE
/*****************************************************
END
```

```
#FUNCTION A #PARM Y
```

```
PLEASE ENTER A VALID FUNCTION
#FUNCTION A #PARM Y
```

 Copyright © Software AG 2002

Example 2

```
/* EXAMPLE 'DECEX1E:' DECIDE FOR (EVERY OPTION)
/*******************************************************************
DEFINE DATA LOCAL
1 #FIELD1 (N5.4)
END-DEFINE
/*******************************************************************
INPUT #FIELD1
/*******************************************************************
DECIDE FOR EVERY CONDITION
           WHEN #FIELD1 >= 0
             WRITE '#FIELD1 is positive or zero.'
           WHEN #FIELD1 <= 0
             WRITE '#FIELD1 is negative or zero.'
           WHEN FRAC(#FIELD1) = 0
             WRITE '#FIELD1 has no decimal digits.'
           WHEN ANY
             WRITE 'Any of the above conditions is true.'
           WHEN ALL
             WRITE '#FIELD1 is zero.'
           WHEN NONE
             IGNORE
END-DECIDE
/*******************************************************************
END
```

```
#FIELD1 42
```

```
Page      1                                                 90-10-29  12:24:33

  #FIELD1 is positive or zero.
  #FIELD1 has no decimal digits.
  Any of the above conditions is true.
```

DECIDE ON

```
DECIDE ON { FIRST } [VALUES] [OF] operand1
          { EVERY }

    { VALUES { operand2                                  } statement... }
             { operand2,...                              }
             { [operand2,]...operand2 : operand2         }
    }...

    [ANY [VALUES] statement...]

    [ALL [VALUES] statement...]

    NONE [VALUES] statement...

END-DECIDE
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S	A		N	A	N	P	I	F	B	D	T	L		G	O	yes	no
Operand2	C	S	A			A	N	P	I	F	B	D	T	L		G	O	yes	no

Related Statements: DECIDE FOR | IF

Function

The DECIDE ON statement is used to specify multiple actions to be performed depending on the value (or values) contained in a variable.

Note:
If **no** action is to be performed under a certain condition, you specify the statement IGNORE in the corresponding clause of the DECIDE ON statement.

FIRST/EVERY

With one of these keywords, you indicate whether only the first or every value that is found is to be processed.

 Copyright © Software AG 2002

Selection Field - operand1

As *operand1* you specify the field whose contents is to be checked.

VALUES Clause

With this clause, you specify the value *(operand2)* of the selection field, as well as the *statement(s)* which are to be executed if the field contains that value.

You can specify one value, multiple values, or a range of values optionally preceded by one or more values.

Multiple values must be separated from one another either by the input delimiter character (as specified with the session parameter ID) or by a comma. A comma must not be used for this purpose, however, if the comma is defined as decimal character (with the session parameter DC).

For a range of values, you specify the starting value and ending value of the range, separated from each other by a colon.

ANY

With ANY, you specify the *statement(s)* which are to be executed if *any* of the values in the VALUE clause are found. These statements are to be executed *in addition to* the statement specified in the VALUE clause.

ALL

With ALL, you specify the *statement(s)* which are to be executed if *all* of the values in the VALUE clause are found. These statements are to be executed *in addition* to the statement specified in the VALUE clause.

The ALL clause applies only if the keyword EVERY is specified.

NONE

With NONE, you specify the *statement(s)* which are to be executed if *none* of the specified values are found.

Example 1

```
/* EXAMPLE 'DECEX2': DECIDE ON (FIRST OPTION)
/**********************************************************************
SET KEY ALL
INPUT 'TO UPDATE A RECORD, USE PF1 KEY' /
      'TO ADD    A RECORD, USE PF2 KEY' /
/**********************************************************************
/* ROUTINE-UPD IS TO BE EXECUTED IF PF1 IS USED,
/* ROUTINE-ADD IS TO BE EXECUTED IF PF2 IS USED,
/* IF EITHER PF1 OR PF2 USED, END TRANSACTION IS TO BE EXECUTED,
/* IF NEITHER PF1 NOR PF2 ARE USED, NO STATEMENTS ARE TO BE EXECUTED.
/**********************************************************************
DECIDE ON FIRST VALUE OF *PF-KEY
   VALUE 'PF1'
     PERFORM ROUTINE-UPD
   VALUE 'PF2'
     PERFORM ROUTINE-ADD
   ANY VALUE
     END TRANSACTION
     WRITE 'RECORD HAS BEEN MODIFIED'
   NONE VALUE
     IGNORE
END-DECIDE
/**********************************************************************
END
```

 Copyright © Software AG 2002

Example 2

```
/* EXAMPLE 'DECEX2E': DECIDE ON (EVERY OPTION)
/* THIS EXAMPLE SHOWS THE EFFECT OF USING THE EVERY CLAUSE
/*****************************************************************
INPUT 'ENTER ANY VALUE BETWEEN 1 AND 9:' FIELD1(N1) (SG=OFF)
DECIDE ON EVERY VALUE OF FIELD1
  VALUE 1 : 4
    WRITE 'CONTENT OF FIELD1 IS 1-4'
  VALUE 2 : 5
    WRITE 'CONTENT OF FIELD1 IS 2-5'
  ANY VALUE
    WRITE 'CONTENT OF FIELD1 IS 1-5'
  ALL VALUE
    WRITE 'CONTENT OF FIELD1 IS 2-4'
  NONE VALUE
    WRITE 'CONTENT OF FIELD1 IS NOT 1-5'
END-DECIDE
/*******************************************
END
```

```
ENTER ANY VALUE BETWEEN 1 AND 9: 4
```

```
Page      1                                                  94-06-16  12:47:24

  CONTENT OF FIELD1 IS 1-4
  CONTENT OF FIELD1 IS 2-5
  CONTENT OF FIELD1 IS 1-5
  CONTENT OF FIELD1 IS 2-4
```

DEFINE...

The following is a list of the DEFINE... statements:

- DEFINE CLASS
- DEFINE DATA
- DEFINE PRINTER
- DEFINE SUBROUTINE
- DEFINE WINDOW
- DEFINE WORK FILE

 Copyright © Software AG 2002

DEFINE CLASS

```
DEFINE CLASS class-name
          [ [WITH] ACTIVATION [POLICY] { EM } ]
                                       { ES }
                                       { IM }
          [ OBJECT { USING { local-data-area      } } ]...
                   {       { parameter-data-area } }
                   {       data-definition...    }
          [ LOCAL  { USING { local-data-area      } } ]...
                   {       { parameter-data-area } }
                   {       data-definition       }
          [ID class-GUID]

          [INTERFACE USING copycode]
          [    INTERFACE statement ]...
          [PROPERTY statement]...
          [METHOD statement]...
END-CLASS
```

Function

The DEFINE CLASS statement is used to specify a class from within a Natural class module.

A Natural class module consists of one DEFINE CLASS statement followed by an END statement.

class-name

This is the name that is used by clients to create objects of this class. The name can be up to a maximum of 32 characters long. The name may contain periods: this can be used to construct class names such as <company-name>.<application-name>.<class-name>. Each part between the periods (...) must conform to the Natural naming conventions for user variables (please refer to the Natural Reference documentation for further information).

If the class is planned to be used by clients written in different programming languages, the class name should be chosen in a way that it does not conflict with the naming conventions that apply in these languages. Bolero for example uses the Java naming convention. So, a class that is planned to be used in a Bolero client should also follow the Java naming conventions.

WITH ACTIVATION POLICY Clause

The WITH ACTIVATION POLICY clause is used to define explicitly the activation policy which is registered for the current class.

You can set the following parameters:

Parameter	Description
EM	Sets activation policy *ExternalMultiple*
ES	Sets activation policy *ExternalSingle*
IM	Sets activation policy *InternalMultiple*

When the class is stowed and registered, the setting in the WITH ACTIVATION POLICY clause overrides the ACTPOLICY=*activation-policy* (for OS/390, DCOM=(ACTPOL=*activation-policy*)) profile parameter, but is in turn overridden by manual registration using the REGISTER command with an explicit activation policy definition. For further information, see the section Activation Policies.

OBJECT Clause

The OBJECT clause is used to define the object data. The syntax of the OBJECT clause is the same as for the LOCAL clause of the DEFINE DATA statement. For further information, see the description of the LOCAL clause of the DEFINE DATA statement.

LOCAL Clause

The LOCAL clause is only used to include globally unique IDs (GUIDs) in the class definition. GUIDs need only be defined if a class is to be registered with DCOM. GUIDs are mostly defined in a local data area. For further information, see the section Globally Unique Identifiers (GUIDs).

The syntax of the LOCAL clause is the same as for the LOCAL clause of the DEFINE DATA statement. For further information, see the description of the LOCAL clause of the DEFINE DATA statement.

ID Clause

The ID clause is used to assign a globally unique ID for the class. The class GUID is the name of a GUID defined in the data area that is included by the LOCAL clause. The class GUID is a (named) alphanumeric constant. A GUID must be assigned to a class if it is to be registered with DCOM.

INTERFACE USING Clause

The INTERFACE USING clause is used to include copycode that contains INTERFACE statements.

copycode

The copycode used by the INTERFACE USING clause may contain one or more INTERFACE statements.

See the following statements for further information:

 Copyright © Software AG 2002

- INTERFACE
- PROPERTY
- METHOD

DEFINE DATA

General Syntax

```
DEFINE DATA

    [GLOBAL USING global-data-area [WITH block [,block]...]]
    [PARAMETER {USING parameter-data-area
                parameter-data-definition...}]...
    [OBJECT {USING {local-data-area
                    parameter-data-area}
             data-definition...}]...
    [LOCAL {USING {local-data-area
                   parameter-data-area}
            data-definition...}]...
    [INDEPENDENT AIV-data-definition...]
    [CONTEXT {USING {local-data-area
                     parameter-data-area}
              context-data-definition...}]

END-DEFINE
```

Note:
If more than one clause is used, the GLOBAL, PARAMETER, OBJECT, LOCAL, INDEPENDENT and CONTEXT clauses must be specified in the order shown above.
An "empty" DEFINE DATA statement is not allowed; in other words, at least one clause (GLOBAL, PARAMETER, OBJECT, LOCAL, INDEPENDENT or CONTEXT) must be specified and at least one field must be defined.

Function

The DEFINE DATA statement is used to define the data areas which are to be used within a Natural program. When a DEFINE DATA statement is used, it must be the first statement of the program/routine.

DEFINE DATA in Structured Mode

In structured mode, all variables to be used must be defined in the DEFINE DATA statement; they must not be defined elsewhere in the program.

 Copyright © Software AG 2002

DEFINE DATA in Reporting Mode

In reporting mode, the DEFINE DATA statement is not mandatory since variables may be defined in the body of the program. However, if a DEFINE DATA LOCAL statement is used in reporting mode, variables (except AIVs) must not be defined elsewhere in the program; and if a DEFINE DATA INDEPENDENT statement is used in reporting mode, application-independent variables (AIVs) must not be defined elsewhere in the program.

data areas

Natural supports three types of data areas:

- global data area
- parameter data area
- local data area

global data area

A global data area contains data elements which can be referenced by more than one programming object (as described in section Object Types of the Natural Programming Guide). The global data area and the objects which reference it must be contained in the same library (or in a steplib). No more than one global data area is allowed per DEFINE DATA statement.

parameter data area

A parameter data area contains data elements which are used as parameters in a subprogram, external subroutine or dialog. Parameter data elements must not be assigned initial or constant values, and they must not have EM, HD or PM definitions. Parameter data elements can also be defined within the subprogram/subroutine itself. Parameters can also be defined within a helproutine.

local data area

A local data area contains data elements which are to be used in a single Natural module. (Local data can also be defined directly within a program or routine.) A data area defined using DEFINE DATA LOCAL may be a parameter data area.

All three types of data areas can be created and maintained by using the data area editor.

block

Data blocks can overlay one another during program execution, thereby saving storage space.

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

.block

.block notations(s) specify the block(s) which are used in the program.

For further information on data blocks, see the section Data Blocks in the Natural Programming Guide.

data-definition

```
{ level { group-name [(array–definition)] }
        { view–definition                 }
        { redefinition                    }
        { variable–definition             }
        { handle–definition               }
```

level	Level number is a 1- or 2-digit number in the range from 01 to 99 (the leading "0" is optional) used in conjunction with field grouping. Fields assigned a level number of 02 or greater are considered to be a part of the immediately preceding group which has been assigned a lower level number. The definition of a group enables reference to a series of fields (may also be only 1 field) by using the group name. This provides a convenient and efficient method of referencing a series of consecutive fields. A group may consist of other groups. When assigning the level numbers for a group, no level numbers may be skipped.
group-name	The name of a group. The name must adhere to the rules for defining a Natural variable name.

 Copyright © Software AG 2002

parameter-data-definition

```
{ level { group-name [(array-definition)]
          redefinition
          variable-name { (format-length)
                        { (format-length/array-definition) } [BY VALUE [RESULT] [OPTIONAL]]
          parameter - handle - definition [BY VALUE [RESULT] [OPTIONAL]] } }
```

level	This is the same as under data-definition.
group-name	This is the same as under data-definition.
variable-name	This is the same as under variable-definition.
format-length	This is the same as under variable-definition.
DYNAMIC	A parameter may be defined as DYNAMIC. For more information on processing dynamic variables, see Large and Dynamic Variables/Fields. Depending on whether call-by-reference or call-by-value is used, the appropriate transfer mechanism is applicable. For further information, see the CALLNAT statement. Note: DYNAMIC is not available on mainframe computers.
BY VALUE	Without BY VALUE (default), a parameter is passed to a subprogram/subroutine by reference (that is, via its address); therefore a field specified as parameter in a CALLNAT/PERFORM statement must have the same format/length as the corresponding field in the invoked subprogram/subroutine. With BY VALUE, a parameter is passed to a subprogram/subroutine by value; that is, the actual parameter value (instead of its address) is passed. Consequently, the field in the subprogram/subroutine need not have the same format/length as the CALLNAT/PERFORM parameter. The formats/lengths must only be data transfer compatible. For data transfer compatibility, the Rules for Arithmetic Assignment/Data Transfer apply (see the Natural Reference documentation). BY VALUE allows you, for example, to increase the length of a field in a subprogram/subroutine (if this should become necessary due to an enhancement of the subprogram/subroutine) without having to adjust any of the objects that invoke the subprogram/subroutine. For parameter definitions for dialogs (under Windows and Windows NT), the following applies: - Without BY VALUE, a parameter, as specified in the inline definition of a dialog's parameter data area, is transferred via its address (by reference); the format and length of the parameter in an OPEN DIALOG or SEND EVENT statement, for example, must match the format and length of the parameter in the inline parameter data definition of the dialog. You can use a parameter by reference in the before open and after open event handlers and in all other events if the used parameters are transferred in the SEND EVENT statement triggering this event. - With BY VALUE, a parameter is transferred via its value; format and length do not have to match; the parameter in the OPEN DIALOG or SEND EVENT statement must be data transfer compatible with the parameter of the dialog.

BY VALUE RESULT	While BY VALUE applies to a parameter passed to a subprogram/subroutine, BY VALUE RESULT causes the parameter to be passed by value in both directions; that is, the actual parameter value is passed from the invoking object to the subprogram/subroutine and, on return to the invoking object, the actual parameter value is passed from the subprogram/subroutine back to the invoking object. With BY VALUE RESULT, the formats/lengths of the fields concerned must be data transfer compatible in both directions. Note: BY VALUE RESULT cannot be used in dialogs.
OPTIONAL	For a parameter defined without OPTIONAL (default), a value *must* be passed from the invoking object. For a parameter defined with OPTIONAL, a value can but need not be passed from the invoking object to this parameter. In the invoking object, the notation *n*X is used to indicate parameters which are skipped, that is, for which no values are passed. Note: OPTIONAL is not available on mainframe computers. With the SPECIFIED Option you can find out at run time whether an optional parameter has been defined or not.

 Copyright © Software AG 2002

Example of BY VALUE:

```
* Program
DEFINE DATA LOCAL
  1 #FIELDA (P5)
  ...
END-DEFINE
...
CALLNAT 'SUBR01' #FIELDA
...
```

```
* Subroutine SUBR01
DEFINE DATA PARAMETER
  1 #FIELDB (P9) BY VALUE
END-DEFINE
...
```

Example of BY VALUE for Dialog:

```
/*Example of three parameters not passed BY VALUE:
1 #A (A10)   /* Parameter Data Definition
1 #B (A20)   /* of the called Dialog
1 #C (A30)   /*
OPEN DIALOG 'MYDIALOG' #DLG$WINDOW WITH #X #Y #Z /* #X has to be A10,#Y has to
                                                 /* be A20,and #Z has to be A30
/*Example of three parameters passed BY VALUE:
1 #A (A10) BY VALUE   /* Parameter Data Definition
1 #B (A20) BY VALUE   /* of the called Dialog
1 #C (A30) BY VALUE   /*
OPEN DIALOG 'MYDIALOG' #DLG$WINDOW WITH #X #Y #Z  /* #X may be A1, #Y may be
                                                  /* A100,and #Z may be A253
```

parameter-handle-definition

```
handle-name { HANDLE OF { dialog-element-type | OBJECT } |
              (array-definition) HANDLE OF { dialog-element-type | OBJECT } }
```

handle-definition

```
handle-name { HANDLE OF { dialog-element-type | OBJECT } [ { CONSTANT | INIT } init-definition ] |
              (array-definition) HANDLE OF { dialog-element-type | OBJECT } [ { CONSTANT | INIT } array-init-definition ] }
```

The use of "*handle-definition*" with "*dialog-element-type*" is only possible under Windows.

handle-name	The name to be assigned to the handle; the naming conventions for user-defined variables apply (see the Natural Reference documentation).
dialog-element-type	The type of dialog element (only possible under Windows). Its possible values are the values of the TYPE attribute. For details, see the sections **Dialog Elements** and **Attributes** of the Natural Dialog Components documentation for Windows.
OBJECT	Is used in conjunction with NaturalX as described in the NaturalX documentation.

The HANDLE definition in the DEFINE DATA statement is generated automatically on the creation of a dialog element or dialog.

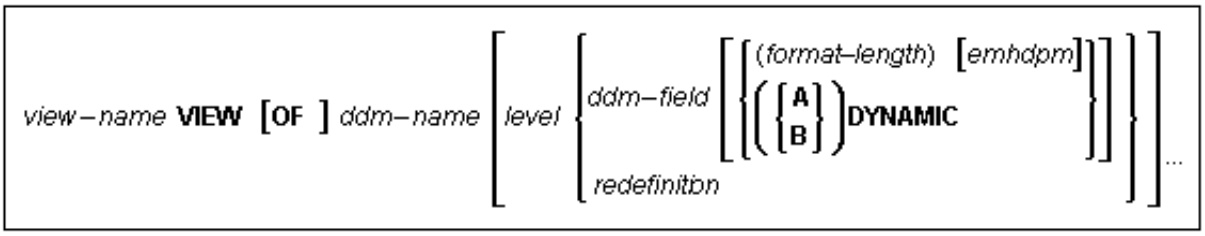
After having defined a handle, you can use the handle-name in any statement to query, set or modify attribute values for the defined dialog-element-type (see the section **Event-Driven Programming Techniques** in the *Natural User's Guide for Windows*).

Examples of handle-definitions:

```
1 #SAVEAS-MENUITEM HANDLE OF MENUITEM
1 #OK-BUTTON (1:10) HANDLE OF PUSHBUTTON
```

Note:
If you use "block" structures, a HANDLE OF OBJECT may only be defined in the master block, but not in any subordinate blocks.

view-definition



 Copyright © Software AG 2002

A *view-definition* is used to define a view as derived from a DDM.

In a parameter data area, *view-definition* is not permitted.

view-name	The name to be assigned to the view. Rules for Natural variable names apply.
ddm-name	The name of the DDM from which the view is to be taken.
level	Level number is a 1- or 2-digit number in the range from 01 to 99 (the leading "0" is optional) used in conjunction with field grouping. Fields assigned a level number of 02 or greater are considered to be a part of the immediately preceding group which has been assigned a lower level number. The definition of a group enables reference to a series of fields (may also be only one field) by using the group name. This provides a convenient and efficient method of referencing a series of consecutive fields. A group may consist of other groups. When assigning the level numbers for a group, no level numbers may be skipped.
ddm-field	The name of a field to be taken from the DDM. When you define a view for a HISTOGRAM statement, the view must contain only the descriptor for which the HISTOGRAM is to be executed.
format-length	Format and length of the field. If omitted these are taken from the DDM. In structured mode, the definition of format and length is not allowed. You may, however, specify the same format/length as in the DDM (for documentation purposes).
DYNAMIC	Defines a view field as DYNAMIC. For more information on processing dynamic variables, see the section Large and Dynamic Variables/Fields in the Natural User's Guide documentation. Note: DYNAMIC is not available on mainframe computers.

redefinition

```
REDEFINE field-name { level { rfield (format-length) } }...
                    { level {      FILLER nX         } }
```

A redefinition may be used to redefine a group, a view or a single field/variable.

field-name	The name of the group or single field that is being redefined.
level	This is the same as under view-definition.
rfield	The name of the field resulting from the redefinition. **Note:** In a "redefinition" within a "view-definition", the name of "rfield" must be different from any field name in the underlying DDM.
format-length	The format and length of the rfield.
FILLER*n***X**	With this notation, you 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.

Notes:
A handle cannot be redefined. A group that contains a handle can only be redefined up to - but not including or beyond - the handle. In a "parameter-data-definition", a "redefinition" of groups is only permitted within a REDEFINE block; otherwise internal errors might occur when passing parameters between the calling program and the called subprogam.

REDEFINE - Example 1:

```
DEFINE DATA LOCAL
  01 #VAR1 (A15)
  01 #VAR2
     02 #VAR2A (N4.1) INIT <0>
     02 #VAR2B (P6.2) INIT <0>
  01 REDEFINE #VAR2
     02 #VAR2RD (A10)
END-DEFINE
...
```

REDEFINE - Example 2:

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

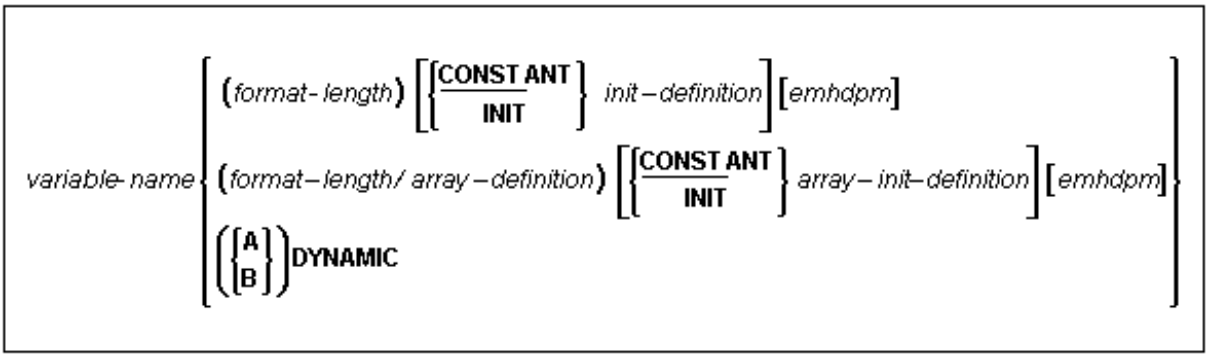
 Copyright © Software AG 2002

REDEFINE - Example 3:

```
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
...
```

variable-definition

- Default_Initial_Values



variable-name	The name to be assigned to the variable. Rules for Natural variable names apply. For information on naming conventions for user-defined variables, see the Natural Reference documentation.
format-length	The format and length of the field. For information on format/length definition of user-defined variables, see the Natural Reference documentation.
DYNAMIC	A field may be defined as DYNAMIC. For more information on processing dynamic variables, see the section Large and Dynamic Variables/Fields in the Natural User's Guide documentation. Note: DYNAMIC is not available on mainframe computers.
CONSTANT	The variable/array is to be treated as a named constant. 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. **Note:** For reasons of internal handling, it is not allowed to mix variable definitions and constant definitions within one group definition; that is, a group may contain either variables only or constants only.
INIT	The variable/array is to be assigned an initial value. This value will also be used when this variable/array is referenced in a RESET INITIAL statement.

 Copyright © Software AG 2002

The following restrictions apply to a dynamic variable:

- The INIT and the CONST clause are not allowed.
- A dynamic variable may not be defined as an array.
- A redefinition of a dynamic variable is not allowed.
- A dynamic variable may not be contained in a redefinition.

Default Initial Values

If no INIT or CONSTANT specification is supplied, a field will be initialised with a default initial value depending on its format:

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

init-definition

```
{ < constant >
  < system-variable >
  FULL LENGTH < character-s >
  LENGTH n < character-s > }
```

constant	The constant value with which the variable is to be initialized; or the constant value to be assigned to the field. See the Natural Reference documentation for further information on constants.
system-variable	The initial value for a variable may also be the value of a Natural system variable. **Note:** When the variable is referenced in a RESET INITIAL statement, the system variable is evaluated again; that is, it will be reset not to the value it contained when program execution started but to the value it contains when the RESET INITIAL statement is executed.
FULL LENGTH **LENGTH** *n*	As initial value, a variable can be filled, entirely or partially, with a specific single character or string of characters (only possible for alphanumeric variables). With the "FULL LENGTH" option, the entire field will be filled with the specified *character* or *characters*. With the "LENGTH *n*" option, the first *n* positions of the field will be filled with the specified *character* or *characters*. *n* must be a numeric constant.

Example of System Variable as Initial Value:

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

Example of FULL LENGTH:

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

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

Example of LENGTH *n*:

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
```

array-definition

- Variable_Arrays_in_a_Parameter_Data Area

{*index* [: *index*] }$_{,...3}$

 Copyright © Software AG 2002

With an array-definition, you define the dimensions (indices) of an array.

index	This can be a numeric integer constant, or a previously defined named constant, or (for database arrays) a previously defined user-defined variable.

For information on defining arrays, see also the section Index Notation in the Natural Reference documentation.

Examples of Array Definitions:

```
#ARRAY1(A5/3)                    /* a one-dimensional array
#ARRAY2(A5/1:5,1:5)              /* a two-dimensional array
#ARRAY3(A5/1:10,1:10,1:10)       /* a three-dimensional array
```

Variable Arrays in a Parameter Data Area

In a parameter data area, you may specify an array with a variable number of occurrences. This is done with the index notation "`1:V`". For example:

```
#ARRAYX (A5/1:V)
```

```
#ARRAYY (I2/1:V,1:V)
```

An array that is defined with index "1:V" must not be redefined or be the result of a redefinition. As the number of occurrences is not known at compilation time, it must not be referenced with the index notation (*) in any statement, except ADD, COMPRESS, COMPUTE, DISPLAY, DIVIDE, EXAMINE, IF, MULTIPLY, RESET, SUBTRACT.

A variable array can only be referenced either in its entirety (that is, all its occurrences) or as a scalar value (that is, a single occurrence). For example:

```
#ARRAYX (*)
#ARRAYY (*,*)
#ARRAYX (1)
#ARRAYY (5,#FIELDX)
```

A partial range of a variable array must not be referenced:

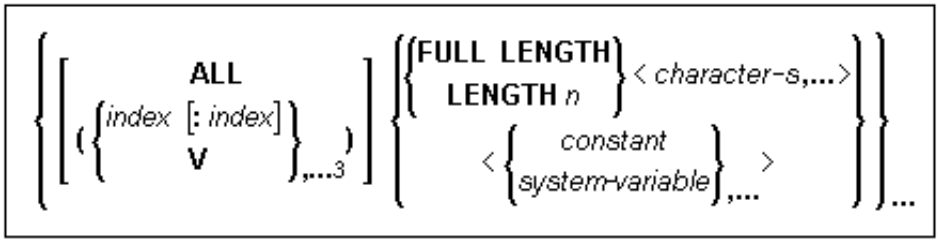
```
#ARRAYY (1,*)  /* not allowed
```

To avoid runtime errors, the maximum number of occurrences of such an array should be passed to the subprogram/subroutine via another parameter.

Notes:
If a parameter data area that contains an index "1:V" is used as a local data area (that is, specified in a DEFINE DATA LOCAL statement), a variable named "V" must have been defined as CONSTANT.
In a dialog, an index "1:V" cannot be used in conjunction with BY VALUE.

See also the system variable *OCCURRENCE in the Natural Reference documentation.

array-init-definition



 Copyright © Software AG 2002

With this clause, you define the initial/constant values for an array.

For a redefined field, an *array-init-definition* is not permitted.

ALL	All occurrences in all dimensions of the array are initialized with the same value.
index	Only the array occurrences specified by the *index* are initialized. If you specify *index*, you can only specify one value with *constant*; that is, all specified occurrences are initialized with the same value.
V	This notation is only relevant for multidimensional arrays if the occurrences of one dimension are to be initialized with *different* values. "V" indicates an index range that comprises all occurrences of the dimension specified with "V"; that is, all occurrences in that dimension are initialized. Only *one* dimension per array may be specified with "V". The occurrences are initialized occurrence by occurrence with the values specified for that dimension. The number of values must not exceed the number of occurrences of the dimension specified with "V".
constant	The constant (value) with which the array is to be initialized (INIT), or the constant to be assigned to the array (CONSTANT). See the Natural Reference documentation for further information on defining constants. **Note:** Occurrences for which no values are specified, are initialized with a default value.
system-variable	The initial value for an array may also be the value of a Natural system variable. **Note:** Multiple constant values/system variables must be separated either by the input delimiter character (as specified with the session parameter ID) or by a comma. A comma must not be used for this purpose, however, if the comma is defined as decimal character (with the session parameter DC).
FULL LENGTH LENGTH*n*	As initial value, it is also possible to have an array filled, entirely or partially, with a specific single character or string of characters (only possible for alphanumeric arrays). With "FULL LENGTH", the entire array occurrence(s) are filled with the specified *character* or *characters*. With "LENGTH *n*", the first *n* positions of the array occurrence(s) are filled with the specified *character* or *characters*. A *system-variable* must not be specified with "FULL LENGTH" or "LENGTH *n*". Within one *array-init-definition*, only either "FULL LENGTH" or "LENGTH *n*" may be specified; both notations must not be mixed.

Example of LENGTH n for Array:

In this example, the first 5 positions of each occurrence of the array will be filled with "NONON".

```
DEFINE DATA LOCAL
1 #FIELD (A25/1:3) INIT ALL LENGTH 5 <'NO'>
...
END-DEFINE
```

Numerous examples of assigning initial values to arrays are provided in the Natural Programming Guide.

emhdpm

([**EM** = *value*] [**HD** = *'value'*] [**PM** = *value*])

 Copyright © Software AG 2002

With this option, additional parameters to be in effect for the field/variable may be defined.

EM=value	This parameter may be used to define an edit mask. See the session parameter EM in the Natural Reference documentation.
HD='value'	This parameter may be used to define the header to be used as the default header for the field (see the DISPLAY statement).
PM=value	This parameter may be used to set the print mode, which indicates how fields are to be output. See the session parameter PM in the Natural Reference documentation.

If for a database field you specify neither an edit mask (EM=) nor a header (HD=), the default edit mask and default header as defined in the DDM will be used.
However, if you specify one of the two, the other's default from the DDM will `not` be used.

AIV-data-definition

```
level { AIV-definition      }
      {REDEFINE  field-name}
```


Copyright © Software AG 2002

DEFINE DATA INDEPENDENT is used to define application-independent variables (AIVs).

level	Level number is a 1- or 2-digit number (the leading "0" is optional) used in conjunction with field grouping. An application-independent variable must be defined at level 01. Other levels are only used in a redefinition.
REDEFINE	REDEFINE may be used to redefine an application-independent variable. The fields resulting from the redefinition must not be application-independent variables.

AIV-definition

```
variable-name { (format-length) [INIT init-definition]
              { (format-length/array-definition) [INIT array-init-definition] } [emhdpm]
```

variable-name	The name to be assigned to the application-independent variable. The first character of the name must be a "+". Rules for Natural variable names apply. For information on naming conventions for user-defined variables, see the Natural Reference documentation.
format-length	Format and length of the field. For information on format/length definition of user-defined variables, see the Natural Reference documentation.
INIT	The application-independent variable is to be assigned an initial value. This value will also be used when the variable is referenced in a RESET INITIAL statement.
init-definition	This is the same as under variable-definition.
array-definition	This is the same as under variable-definition.
array-init-definition	This is the same as under variable-definition.
emhdpm	This is the same as under variable-definition.

context-data-definition

```
level { variable-definition }
      { redefinition        }
      { handle-definition   }
```

 Copyright © Software AG 2002

DEFINE DATA CONTEXT is used in conjunction with Natural RPC. It is used to define so-called "context variables", that is, variables which are to be available to multiple remote subprograms within one conversation, without having to explicitly pass the variables as parameters with the corresponding CALLNAT statements. It suffices to define the context variables in a DEFINE DATA CONTEXT statement in each subprogram in which they are to be available.

A context variable is referenced by its name, and its content is shared by all subprograms referring to that name within one conversation.

For further information, see Defining a Conversation Context in the Natural RPC documentation.

level
variable-definition
redefinition
handle-definition

DEFINE DATA OBJECT

DEFINE DATA OBJECT is used in conjunction with NaturalX. It is described in the NaturalX documentation.

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 (see first example below).

The qualifier must be a level-1 data element (see second example below).

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
...
```

Example:

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

Note:
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; because if you do not, the user-defined variable will be referenced instead.

 Copyright © Software AG 2002

Example 1

```
/* EXAMPLE 'DDAEX1': DEFINE DATA
/*****************************************************************
DEFINE DATA LOCAL
01 #VAR1 (A15)
01 #VAR2
   02 #VAR2A (N4.1) INIT <1111>
   02 #VAR2B (N6.2) INIT <22222>
01 REDEFINE #VAR2
   02 #VAR2C (A2)
   02 #VAR2D (A2)
   02 #VAR2E (A6)
END-DEFINE
/*****************************************************************
WRITE NOTITLE '=' #VAR2A / '=' #VAR2B /
              '=' #VAR2C / '=' #VAR2D / '=' #VAR2E
/*****************************************************************
END
```

```
#VAR2A:  1111.0
  #VAR2B:  222222.00
  #VAR2C: 11
  #VAR2D: 11
  #VAR2E: 022222
```

Example 2

```
/* EXAMPLE 'DDAEX2': DEFINE DATA (ARRAY DEFINITION/INITIALIZATION)
/*****************************************************************
DEFINE DATA LOCAL
01 #VAR1 (A1/1:2,1:2) INIT (1,V) <'A','B'>
01 #VAR2 (N5/1:2,1:3) INIT (1,2) <200>
01 #VAR3 (A1/1:4,1:3) INIT (V,2:3) <'W','X','Y','Z'>
END-DEFINE
/*****************************************************************
WRITE NOTITLE '=' #VAR1 (1,1) '=' #VAR1 (1,2)
            / '=' #VAR1 (2,1) '=' #VAR1 (2,2)
/*****************************************************************
WRITE ///     '=' #VAR2 (1,1) '=' #VAR2 (1,2)
            / '=' #VAR2 (2,1) '=' #VAR2 (2,2)
/*****************************************************************
WRITE ///     '=' #VAR3 (1,1) '=' #VAR3 (1,2) '=' #VAR3 (1,3)
WRITE       / '=' #VAR3 (2,1) '=' #VAR3 (2,2) '=' #VAR3 (2,3)
WRITE       / '=' #VAR3 (3,1) '=' #VAR3 (3,2) '=' #VAR3 (3,3)
WRITE       / '=' #VAR3 (4,1) '=' #VAR3 (4,2) '=' #VAR3 (4,3)
/*****************************************************************
END
```

Copyright © Software AG 2002

```
#VAR1: A #VAR1: B
  #VAR1:   #VAR1:



  #VAR2:      0 #VAR2:    200
  #VAR2:      0 #VAR2:      0



  #VAR3:   #VAR3: W #VAR3: W

  #VAR3:   #VAR3: X #VAR3: X

  #VAR3:   #VAR3: Y #VAR3: Y

  #VAR3:   #VAR3: Z #VAR3: Z
```

Example 3

```
/* EXAMPLE 'DDAEX3': DEFINE DATA (VIEW DEFINITION, REDEFINE ARRAY)
/*********************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 ADDRESS-LINE (A20/2)
  2 PHONE
1 #ARRAY (A75/1:4)
1 REDEFINE #ARRAY
  2 #ALINE (A25/1:4,1:3)
1 #X (N2) INIT <1>
1 #Y (N2) INIT <1>
END-DEFINE
/*********************************************************************
FORMAT PS=20
LIMIT 5
FIND EMPLOY-VIEW WITH NAME = 'SMITH'
 MOVE NAME            TO #ALINE (#X,#Y)
 MOVE ADDRESS-LINE(1) TO #ALINE (#X+1,#Y)
 MOVE ADDRESS-LINE(2) TO #ALINE (#X+2,#Y)
 MOVE PHONE           TO #ALINE (#X+3,#Y)
 IF  #Y = 3
  RESET INITIAL #Y
  PERFORM PRINT
 ELSE
  ADD 1 TO #Y
 END-IF
 AT END OF DATA
  PERFORM PRINT
 END-ENDDATA
END-FIND
/*********************************************************************
DEFINE SUBROUTINE PRINT
WRITE NOTITLE (AD=OI) #ARRAY(*)
RESET #ARRAY(*)
SKIP 1
END-SUBROUTINE
/*********************************************************************
END
```

 Copyright © Software AG 2002

```
SMITH                     SMITH                     SMITH
  ENGLANDSVEJ 222           3152 SHETLAND ROAD        14100 ESWORTHY RD.
                            MILWAUKEE                 MONTERREY
  554349                    (414)877-4563             (408)994-2260

  SMITH                     SMITH
  5 HAWTHORN                2307 DARIUS LANE
  OAK BROOK                 TAMPA
  (312)150-9351             (813)131-4010
```

Example 4

```
/* EXAMPLE 'DDAEX4': DEFINE DATA (GLOBAL, PARAMETER AND LOCAL AREAS)
/*********************************************************************
/* MAIN PROGRAM
/*********************************************************************
DEFINE DATA GLOBAL USING GLOBAL-1
            LOCAL
            1 #FIELD1 (A10)
            1 #FIELD2 (N5)
END-DEFINE
/*********************************************************************
/* ...
CALLNAT 'SUBP1' #FIELD1 #FIELD2
/* ...
END
```

```
/* SUBPROGRAM 'SUBP1'
DEFINE DATA PARAMETER
1 #FIELDA (A10)
1 #FIELDB (N5)
END-DEFINE
/*********************************************************************
/* ...
END
```

Example 5

```
* EXAMPLE 'DDAEX5': DEFINE DATA (INITIALIZATION)
**********************************************
DEFINE DATA LOCAL
 1 #START-DATE (D)   INIT <*DATX>
 1 #UNDERLINE  (A50) INIT FULL LENGTH <'_'>
 1 #SCALE      (A65) INIT LENGTH 65 <'....+..../'>
END-DEFINE
*
WRITE NOTITLE #START-DATE (DF=L)
    / #UNDERLINE
    / #SCALE
END
```

```
1999-01-19
  ____________________________________________________
  ....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+
```

Example 6

```
   /* EXAMPLE 'DDAEX6': DEFINE DATA (VARIABLE ARRAY)
   /*****************************************************************
   DEFINE DATA
      PARAMETER
        01 #STRING (A1/1:V)
        01 #MAX (P3)
      LOCAL
         01 #I (P3)
   END-DEFINE
   /*****************************************************************
   F0R #I = 1 TO #MAX
     IF #STRING (#I) < H'40'
       MOVE H'40' TO #STRING (#I)
     END-IF
   END-FOR
   END
```

Example 7

```
DEFINE DATA LOCAL
  1 #MyHomePage (A4096)        /* alpha variable with max. 4096 characters
  1 #MyStream (B1000000/1:10)  /* binary array with 10 occurrences and max. 1000000 bytes per occ.
  1 #MyDynHomePage (A) DYNAMIC /* dynamic alpha variable
  1 #MyDynStream (B) DYNAMIC   /* dynamic binary variable
  END-DEFINE
```

 Copyright © Software AG 2002

DEFINE PRINTER

```
DEFINE PRINTER ( [logical-printer-name =] n)
               [OUTPUT operand1]
               [ PROFILE operand2 ]
               [  FORMS  operand2 ]
               [  NAME   operand2 ]
               [  DISP   operand2 ]
               [  CLASS  operand2 ]
               [ COPIES  operand3 ]
               [  PRTY   operand4 ]...7
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S				A											yes	no
Operand2	C	S				A											yes	no
Operand3	C	S					N										yes	no
Operand4	C	S					N	P	I								yes	no

Related Statements: CLOSE PRINTER | DISPLAY | WRITE

Function

The DEFINE PRINTER statement is used to assign a symbolic name to a report number and to control the allocation of a report to a logical destination. This provides you with additional flexibility when creating output for various logical print queues.

When this statement is executed and the specified printer is already open, the statement will implicitly cause that printer to be closed. To explicitly close a printer, however, you should use the CLOSE PRINTER statement.

Printer

As *logical-printer-name*, you specify the logical name which is to be allocated to printer *n*. This is the name which will be used for the "*rep*" notation in a DISPLAY/WRITE statement. Naming conventions for the *logical-printer-name* are the same as for user-defined variables (see "General Information" in the Natural Reference documentation).

The printer number *n* may be a value in the range of 1 - 31. For Natural Advanced Facilities, the valid range is also 1 - 31.

Multiple logical names may be assigned to the same printer number.

Unlike the output destination (see below), *logical-printer-name=n* is evaluated at *compilation time* and therefore independent of the program control flow.

OUTPUT *operand1*

- Printers under OS/390 with Access Method AM=STD - Standard Batch
- Printers under VM/CMS with Access Method AM=STD - Standard Batch
- Printers under BS2000/OSD with Access Method AM=STD - Standard Batch
- Printers under Com-plete
- Printers under Natural Advanced Facilities
- Additional Reports

With *operand1*, you specify the destination within the online spooling system. The destination is evaluated at *runtime*.

If *operand1* is a variable, its length must be at least 8.

Any logical printer name may be assigned, provided that it has been defined in the parameter file/parameter module used, or via parameters or JCL during startup of Natural.

Under OpenVMS, UNIX and Windows, the name must be specified as LPT*nn*, where *nn* may be 1 to 31.

Examples:

LPT12 (OpenVMS, UNIX or Windows printer name)
CMPRT01 (OS/JCL DDNAME)
P01 (BS2000/OSD file command)
TID111 (Com-plete printer name)
PCPRNT01 (printer defined for Natural Connection)

If the OUTPUT clause is omitted, the destination depends on the "Device Assignment" in the parameter file used; on mainframe computers, it depends on the profile parameter PRINT or macro NTPRINT respectively (see the corresponding Natural Installation and Operations documentation or Natural Operations documentation).

Printers under OS/390 with Access Method AM=STD - Standard Batch

- Logical Dataset Names
- Physical Dataset Names
- HFS File
- JES Spool File Class
- NULLFILE
- Allocation and De-Allocation of Datasets
- Print Files in Server Environments

Under OS/390, for a printer number that is defined with access method AM=STD (either automatically in the JCL or in the NTPRINT macro of the Natural parameter module or with the dynamic profile parameter PRINT), you can use *operand1* to specify a logical or a physical dataset name to be assigned to that printer number.

In this case, *operand1* can be 1 to 253 characters long and can be one of the following:

- a logical dataset name (DD name, 1 to 8 characters);
- a physical dataset name of a cataloged dataset (1 to 44 characters), or a physical dataset member name (1 to 44 characters for the dataset name, plus 1 to 8 characters in parentheses for the member name);
- a path name and member name of an HFS file (1 to 253 characters) in an MVS UNIX Services environment;
- a JES spool file class;

 Copyright © Software AG 2002

- "NULLFILE" (to indicate a dummy dataset).

Logical Dataset Names

For example:

```
DEFINE PRINTER (21) OUTPUT 'SYSPRINT'
```

The specified dataset must have been allocated before the DEFINE PRINTER statement is executed.

The allocation can be done via JCL, CLIST or dynamic allocation (SVC 99). For dynamic allocation you can use the user exit USR2021 in library SYSEXT.

The dataset name specified in the DEFINE PRINTER statement overrides the name specified with the subparameter DEST of the NTPRINT macro or PRINT profile parameter.

Optionally, the dataset name may be prefixed by "DDN=" to indicate that it is a DD name and to avoid name conflicts with additional reports. For example:

```
DEFINE PRINTER (22) OUTPUT 'DDN=SOURCE'
```

Physical Dataset Names

For example:

```
DEFINE PRINTER (23) OUTPUT 'TEST.PRINT.FILE'
```

The specified dataset must exist in cataloged form. When the DEFINE PRINTER statement is executed, the dataset is allocated dynamically by SVC 99 with the current DD name and option DISP=SHR.

If the dataset name is 8 characters or shorter and does not contain a period ".", it might be misinterpreted as a DD name. To avoid this, prefix the name with "DSN=". For example:

```
DEFINE PRINTER (22) OUTPUT 'DSN=PRINTXYZ'
```

If the dataset is a PDS member, you specify the PDS member name (1 to 8 characters) in parentheses after the dataset name (1 to 44 characters). For example:

```
DEFINE PRINTER (4) OUTPUT 'TEST.PRINT.PDS(TEST1)'
```

If the specified member does not exist, a new member of that name will be created.

HFS File

For example:

```
DEFINE PRINTER (14) OUTPUT '/u/nat/rec/test.txt'
```

The specified path name must exist. When the DEFINE PRINTER statement is executed, the HFS file is allocated dynamically. If the specified member does not exist, a new member of that name will be created.

For the dynamic allocation of the dataset, the following OS/390 path options are used:

```
PATHOPTS=(OCREAT,OTRUNC,ORDWR)
  PATHMODE=(SIRUSR,SIWUSR,SIRGRP,SIWGRP)
  FILEDATA=TEXT
```

When an HFS file is closed, it is automatically de-allocated by OS/390 (regardless of the setting of the subparameter FREE in the NTPRINT macro or PRINT profile parameter).

JES Spool File Class

To create a JES spool dataset, you specify SYSOUT=x (where x is the desired spool file class). For the default spool file class, you specify SYSOUT=*.

Examples:

```
DEFINE PRINTER (10) OUTPUT 'SYSOUT=A'
  DEFINE PRINTER (12) OUTPUT 'SYSOUT=*'
```

To specify additional parameters for the dynamic allocation, use the user exit USR2021 in library SYSEXT instead of the DEFINE PRINTER statement.

 Copyright © Software AG 2002

NULLFILE

To allocate a dummy dataset, you specify NULLFILE as *operand1*:

```
DEFINE PRINTER (n) OUTPUT 'NULLFILE'
```

This corresponds to the JCL definition:

```
// DD-name DD DUMMY
```

Allocation and De-Allocation of Datasets

When the DEFINE PRINTER statement is executed and a physical dataset name, HFS file, spool file class or dummy dataset has been specified, the corresponding dataset is allocated dynamically. If the logical print file is already open, it will be closed automatically, except when the profile parameter CLOSE=FIN has been specified, in which case an error will be issued. Moreover, an existing dataset allocated with the same current DD name is automatically de-allocated before the new dataset is allocated. Print files that are to be allocated dynamically have to be predefined in the Natural parameter module with AM=STD.

To avoid unnecessary overhead by unsuccessful premature opening of print files not yet allocated at the start of the program, print files should be defined with the subparameter OPEN=ACC (open at first access) in the NTPRINT macro or PRINT profile parameter.

In the case of an HFS file, or a print file defined with the subparameter FREE=ON in the NTPRINT macro or PRINT profile parameter, the print file is automatically de-allocated as soon as it has been closed.

As an alternative for the dynamic allocation and de-allocation of datasets, the user exit USR2021 in library SYSEXT is provided. This user exit also allows you to specify additional parameters for dynamic allocation.

Print Files in Server Environments

In server environments, errors may occur if multiple Natural sessions attempt to allocate or open a dataset with the same DD name. To avoid this, you either specify the print file with subparameter DEST=* in the NTPRINT macro or PRINT profile parameter, or you specify OUTPUT '*' in the DEFINE PRINTER statement; Natural then generates a unique DD name at the physical dataset allocation when the first DEFINE PRINTER statement for that print file is executed.

All print files whose DD names begin with "CM" are shared by all sessions in a server environment. A shared print file is opened by the first session, and is physically closed when the server is terminated. For further information, see the section Natural as a Server in the Natural Operations for Mainframes documentation.

Printers under VM/CMS with Access Method AM=STD - Standard Batch

Under VM/CMS, for a printer number that is defined with the access method AM=STD (either automatically in the JCL or in the NTPRINT macro of the Natural parameter module or with the dynamic profile parameter PRINT), you can use *operand1* to specify a logical or a physical dataset name to be assigned to that printer number.

For this, the same applies as under OS/390 (see Printers under OS/390 with Access Method AM=STD - Standard Batch), but with the following differences:

- Instead of dynamic allocation via MVS SVC 99, the CMS command FILEDEF is used to define a dataset.
- HFS files are not supported.
- JES spool classes are not supported.
- In addition, the following syntax is used:

  ```
  DEFINE PRINTER (n) OUTPUT ('fname ftype fmode (options)')
  ```

  This generates the CMS command:

  ```
  FILEDEF ddname-n DISK fname ftype fmode (options)
  ```

- Moreover, the following syntax is allowed:

  ```
  DEFINE PRINTER (n) OUTPUT ('FILEDEF=filedef-parameters')
  ```

  This generates the CMS command:

  ```
  FILEDEF ddname-n =filedef-parameters
  ```

 Copyright © Software AG 2002

Printers under BS2000/OSD with Access Method AM=STD - Standard Batch

Under BS2000/OSD, for a printer number that is defined with the access method AM=STD (whether automatically in the JCL, in the NTPRINT macro of the Natural parameter module or dynamically using the profile parameter PRINT), you can use *operand1* to specify a file name, link name or system file that is allocated to this printer number.

In this case, *operand1* can have a length of from 1 to 253 characters and one of the following meanings:

- a BS2000/OSD link name (1 to 8 characters)
- a BS2000/OSD file name (9 to 54 characters)
- a generic BS2000/OSD file name (wildcard)
- a BS2000/OSD file name and link name
- a generic BS2000/OSD file name and link name (wildcard)
- the logical BS2000/OSD system file **SYSOUT**
- the logical BS2000/OSD system file **SYSLST** (or **SYSLSTnn**, nn=01-99)
- the logical BS2000/OSD system file **SYSLST** (**SYSLSTnn**) with allocation to a file name
- the logical BS2000/OSD system file **SYSLST** (**SYSLSTnn**) with allocation to a generic file name (wildcard)
- *DUMMY

The following rules apply:

1. File name and link name can be specified as positional parameters or keyword parameters. The corresponding keywords are **FILE=** and **LINK=**. Mixing positional and keyword parameters is allowed but not recommended.
2. A string with a length of 1 to 8 characters without commas is interpreted as a link name. This notation is compatible with earlier versions of Natural.

   Example:

   ```
   DEFINE PRINTER (1) OUTPUT 'P01'
   ```

   The corresponding definition with a keyword parameter is:

   ```
   DEFINE PRINTER (1) OUTPUT 'LINK=P01'
   ```

3. A string of of 9 to 54 characters without commas is interpreted as a file name.

   Example:

   ```
   DEFINE PRINTER (2) OUTPUT 'NATURAL31.TEST.PRINTER02'
   ```

   The corresponding definition with a keyword parameter is:

   ```
   DEFINE PRINTER (2) OUTPUT 'FILE=NATURAL31.TEST.PRINTER02'
   ```

4. The following input is interpreted without considering the length and therefore forms exceptions to Rules 2 and 3:
   - keyword input: LINK=, FILE=
   - *DUMMY
   - NULLFILE (equivalent to *DUMMY)
   - *
   - *,*
   - SYSOUT
   - SYSLST or SYSLST(*nn*)

   Example: DEFINE PRINTER (7) OUTPUT ’FILE=Y’ is a valid file allocation and not a link name, although the string of characters contains fewer than 9 characters.

5. Generic file names are formed as follows:
   pnn.userid.tsn.date.time.number
   where

nn	is a report number
userid	is a Natural user-ID, 8 characters
tsn	is the BS2000/OSD TSN of the current task, 4 digits
date	is DDMMYYYY
time	is HHIISS
number	is a sequential number, 5 digits

6. Generic link names are formed as follows:

   NPFnnnnn

   nnnnn is a 5-digit number that is increased by one after every generation of a dynamic link name.

7. Changing the file allocation for a printer number causes an implicit CLOSE of the print file allocated so far.

You are strongly recommended, in all cases except when you only specify a link name (for example: P01), to work with keyword parameters. This avoids conflicts of names with additional reports and is essential for file names with fewer than 9 characters.

Examples:

```
DEFINE PRINTER (1) OUTPUT ’LINK=SOURCE’
  DEFINE PRINTER (1) OUTPUT ’FILE=SOURCE’
  DEFINE PRINTER (1) OUTPUT ’SOURCE’
```

 Copyright © Software AG 2002

Link Name

Example:

```
DEFINE PRINTER (1) OUTPUT 'LINKP01'
```

means the same as

```
DEFINE PRINTER (1) OUTPUT 'LINK=LINKP01'
```

A file with the LINK 'LINKP01' must exist at runtime. This can be created either using JCL before starting Natural or by dynamic allocation from the current application. For dynamic allocation, the user exit USR2029 in the library SYSEXT can be used. If, before execution, the link was active as a destination to another file, for example: 'P01', this will be released or retained depending on the value of the profile parameter FREE (possible values are ON and OFF). Release is done via an explicit RELEASE call to the BS2000/OSD command processor.

File Name

Example:

```
DEFINE PRINTER (2) OUTPUT 'NATURAL31.TEST.PRINTER02'
```

means the same as

```
DEFINE PRINTER (2) OUTPUT 'FILE=NATURAL31.TEST.PRINTER02'
```

The file specified in *operand1* is set up using a FILE macro call and inherits the link name that was valid for the corresponding print file before execution of the DEFINE PRINTER statement.

Generic File Name

Example:

```
DEFINE PRINTER (21) OUTPUT '*'
```

means the same as

```
DEFINE PRINTER (21) OUTPUT 'FILE=*'
```

A file with a name created according to Rule 4 is set up using a FILE macro call and inherits the link name that was valid for the corresponding print file before execution of the DEFINE PRINTER statement.

```
DEFINE PRINTER (22) OUTPUT 'FILE=*,LINK=GENFLK22'
```

A file with a name created according to Rule 4 is set up with the specified link name using a FILE macro call.

File Name and Link Name

Example:

```
DEFINE PRINTER (11) OUTPUT 'NATURAL31.TEST.PRINTER11,LNKP11'
```

means the same as

```
DEFINE PRINTER (11) OUTPUT 'FILE=NATURAL31.TEST.PRINTER11,LINK=LNKP11'
```

which means the same as

```
DEFINE PRINTER (11) OUTPUT 'FILE=NATURAL31.TEST.PRINTER11,LNKP11'
```

The file specified in *operand1* is set up with the specified link name using a FILE macro call and allocated to the corresponding printer number.

Generic File Name and Link Name

Example:

```
DEFINE PRINTER (27) OUTPUT '*,*'
```

means the same as

```
DEFINE PRINTER (27) OUTPUT 'FILE=*,LINK=*'
```

A file with a file name and link name created according to Rule 4 and Rule 5 is set up using a FILE macro call and allocated to the specified printer number (27).

Note:
When file name and link name are specified, the previous link name is not released, regardless of the value of the profile parameter FREE.

System File SYSOUT

Example:

```
DEFINE PRINTER (14) OUTPUT 'SYSOUT'
```

Report 14 is written to SYSOUT.

Note:
Under TIAM, SYSOUT is by default output on the screen.

System File SYSLST

Example:

```
DEFINE PRINTER (15) OUTPUT 'SYSLST'
```

Report 15 is written to the system file SYSLST.

 Copyright © Software AG 2002

System File SYSLSTnn - nn=01,...,99

Example:

```
DEFINE PRINTER (16) OUTPUT 'SYSLST16'
```

Report 16 is written to the system file SYSLST16.

System File SYSLST - nn - with Implicit Allocation to a BS2000/OSD File

Examples:

```
DEFINE PRINTER (11) OUTPUT 'SYSLST=LST.PRINTER11'
```

The system file SYSLST is allocated to the file LST.PRINTER11; Report 11 is written to the system file SYSLST.

```
DEFINE PRINTER (13) OUTPUT 'SYSLST13=LST.PRINTER13'
```

The system file SYSLST13 is allocated to the file LST.PRINTER13; Report 13 is written to the system file SYSLST13.

```
DEFINE PRINTER (19) OUTPUT 'SYSLST19=*'
```

The system file SYSLST19 is allocated to a file with a name generated according to Rule 4; Report 19 is written to the system file SYSLST19.

Printers under Com-plete

For Natural users under Com-plete, any printer name can be assigned, even if it has not been defined to Natural.

Assignment Algorithm on Mainframes

To assign the OUTPUT destination name to a report number, an algorithm is executed which works identical for all access methods (as defined with the AM=xxx parameter of the NTPRINT macro).

This algorithm scans the list of printers defined to NATURAL (as it is displayed by the SYSFILE Print File Information screen) to find a name which matches the OUTPUT destination name. During this scan, the access method of the related logical printer is **not** taken into account.

If a matching name is found, the logical printer of this destination is used to spool the report. The SYSFILE output, however, will not be changed, i.e., this routing is internal only and not visible to the user.

If a matching name is **not** found, the logical printer of the LAST entry in the list of defined printers is used to spool the report. In this case, the logical printer name will physically be overwritten. The SYSFILE output reflects this change.

Printers under Natural Advanced Facilities

For Natural Advanced Facilities users, the name of any predefined logical printer profile can be specified. This logical printer profile need not belong to the currently active user profile. It may be any logical printer profile defined on the NATSPOOL file. It will be active only for the duration of the Natural program which contains the DEFINE PRINTER statement. For more information, see the Natural Advanced Facilities documentation.

Additional Reports

Additional reports can be assigned for default with the following names:

SOURCE	Output into the Natural source area.
CONNECT	Output into a Con-nect folder (only on mainframe computers). **Note for Natural installation:** the NATPCNT module of Natural must be linked to the Natural nucleus.
DUMMY	Output to be deleted.
HARDCOPY	Output to the current hardcopy device (only on mainframe computers).
INFOLINE	Output to the Natural infoline. For details on the infoline, see the Natural terminal command %X in the Natural Reference documentation.
WORKPOOL	Output into the Natural ISPF workpool (only on mainframe computers).
CCONTROL	In mainframe environments only: CCONTROL is the name of a special printer control table associated to the printer "*n*-1"; it must not be modified. For further information, refer to Printer-Advance Control Characters in the *Natural Operations for Mainframe documentation*.

 Copyright © Software AG 2002

PROFILE/FORMS/NAME/DISP/CLASS/COPIES/PRTY

Note:
The clauses FORMS, NAME, DISP, CLASS, COPIES and PRTY can only be used on mainframe computers.

With these clauses, you can provide printing control information to be interpreted by the spooling system of the TP monitor or operating system respectively.

You can specify one or more of these clauses, but each of them only once.

With the PROFILE clause, you specify as *operand2* the name of a printer control characters table. Such a table is defined in the configuration file NATCONF.CFG, or for mainframe computers in the NTCCTAB macro respectively (as described in your Natural Installation or Operations documentation).

Note:
With Natural Advanced Facilities, the NTCC table can be maintained online (as described in the Natural Advanced Facilities documentation).

With the other clauses, you can provide values for parameters of the TP monitor's spooling system:

FORMS	Form
NAME	Listname
DISP	Disposition
CLASS	Spool class
COPIES	Number of copies
PRTY	Listing priority (1 - 255)

Those clauses will only use the default values for the first execution.

If one of the clauses listed above was defined once for a certain output, a subsequent DEFINE PRINTER statement with the same output and without this clause will use this definition. If the previous definitions are not clear in a Natural environment, Software AG recommends to set them in each module using DEFINE PRINTER statement.

For the PROFILE, FORMS and NAME clauses, the maximum length allowed for *operand2* is 8; for the DISP clause, its maximum length is 4; for the CLASS clause, its length has to be 1.

For the DISP clause, the possible values for *operand2* are "DEL", "HOLD", "KEEP" and "LEAV". If the DISP clause is omitted (or incorrectly specified), "DEL" applies by default.

Operand3 and *operand4* must be integer values.

Default values can be set with the corresponding subparameters of the profile parameter PRINT.

Example 1

```
DEFINE PRINTER (1) OUTPUT 'TID100'
WRITE (1) 'PRINTED ON PRINTER TID100'
END
```

Example 2

```
DEFINE PRINTER (REPORT1 = 1) OUTPUT 'LPT1'
WRITE (REPORT1) 'REPORT1 PRINTED ON PRINTER LPT1'
END
```

Example 3

```
DEFINE PRINTER (REPORT1 = 1) /* NO 'OUTPUT'
WRITE (REPORT1) 'DEPENDS ON NATPARM SETTING OR JCL IN BATCH'
                'OR ''PRINTER PARAMETER'' UNDER COM-PLETE OR A/F'
END
```

Example 4

```
/* EXAMPLE 'DPIEX1': DEFINE PRINTER INFOLINE
*
SET CONTROL 'XT'  /* INFOLINE TOP
SET CONTROL 'XI'  /* SWITCH INFOLINE MODE
DEFINE PRINTER (1) OUTPUT 'INFOLINE'
WRITE (1) 'EXECUTING' *PROGRAM 'BY' *INIT-USER
WRITE 'TEST OUTPUT'
SET CONTROL 'XI'  /* SWITCH BACK TO NORMAL
END
```

 Copyright © Software AG 2002

```
Page      1                                                  97-06-18  13:31:25

  TEST OUTPUT



  IO=814,AI =650,L=0 C= ,LS=80,P =3,PLS=80,PCS=24,FLD=90,CLS=5,ADA=22
```

DEFINE SUBROUTINE

```
DEFINE [SUBROUTINE] subroutine-name
          statement...
{      END-SUBROUTINE          }
{RETURN (reporting mode only)  }
```

Related Statements: PERFORM | DEFINE DATA PARAMETER

Function

The DEFINE SUBROUTINE statement is used to define a Natural subroutine. A subroutine is invoked with a PERFORM statement.

Inline/External Subroutines

A subroutine may be defined within the object which contains the PERFORM statement that invokes the subroutine (inline subroutine); or it may be defined external to the object that contains the PERFORM statement (external subroutine). An inline subroutine may be defined before or after the first PERFORM statement which references it.

Note:
Although the structuring of a program function into multiple external subroutines is recommended for achieving a clear program structure, please note that a subroutine should always contain a larger function block because the invocation of the external subroutine represents an additional overhead as compared with inline code or subroutines.

subroutine-name

For a subroutine name (maximum 32 characters), the same naming conventions apply as for user-defined variables (see the Natural Reference documentation).

The subroutine name is independent of the name of the module in which the subroutine is defined (it may but need not be the same).

Subroutine Termination

The subroutine definition is terminated with END-SUBROUTINE. In reporting mode, RETURN may also be used to terminate a subroutine.

Restrictions

Any processing loop initiated within a subroutine must be closed before END-SUBROUTINE is issued.

 Copyright © Software AG 2002

An inline subroutine must not contain another DEFINE SUBROUTINE statement (see Example 1 below).

An external subroutine (that is, an object of type subroutine) must not contain more than one DEFINE SUBROUTINE statement block (see Example 2 below). However, an external DEFINE SUBROUTINE block may contain further inline subroutines (see Example 1 below).

Example 1

The following construction is possible in an object of type subroutine, but not in any other object (where SUBR01 would be considered an inline subroutine):

```
...
DEFINE SUBROUTINE SUBR01
   ...
   PERFORM SUBROUTINE SUBR02
   PERFORM SUBROUTINE SUBR03
   ...
   DEFINE SUBROUTINE SUBR02
   /* inline subroutine...
   END-SUBROUTINE
 ...
   DEFINE SUBROUTINE SUBR03
   /* inline subroutine...
   END-SUBROUTINE
END-SUBROUTINE
END
```

Example 2 (invalid):

The following construction is `not` allowed in an object of type subroutine:

```
...
DEFINE SUBROUTINE SUBR01
...
END-SUBROUTINE
DEFINE SUBROUTINE SUBR02
...
END-SUBROUTINE
END
```

Data Available in a Subroutine

- Inline Subroutines
- External Subroutines

Inline Subroutines

No explicit parameters can be passed from the invoking program via the PERFORM statement to an internal subroutine.

An inline subroutine has access to the currently established global data area as well as to the local data area used by the invoking program.

External Subroutines

An external subroutine has access to the currently established global data area. Moreover parameters can be passed directly with the PERFORM statement from the invoking object to the external subroutine; thus, you may reduce the size of the global data area.

An external subroutine has no access to the local data area defined in the calling program; however, an external subroutine may have its own local data area.

 Copyright © Software AG 2002

Example 1

```
/* EXAMPLE 'DSREX1S': DEFINE SUBROUTINE (STRUCTURED MODE)
/*************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 ADDRESS-LINE  (A20/2)
  2 PHONE
1 #ARRAY (A75/1:4)
1 REDEFINE #ARRAY
  2 #ALINE  (A25/1:4,1:3)
1 #X (N2) INIT <1>
1 #Y (N2) INIT <1>
END-DEFINE
/*************************************************************
FORMAT PS=20
LIMIT 5
FIND EMPLOY-VIEW WITH NAME = 'SMITH'
 MOVE NAME            TO #ALINE (#X,#Y)
 MOVE ADDRESS-LINE(1) TO #ALINE (#X+1,#Y)
 MOVE ADDRESS-LINE(2) TO #ALINE (#X+2,#Y)
 MOVE PHONE           TO #ALINE (#X+3,#Y)
 IF  #Y = 3
  RESET INITIAL #Y
  PERFORM PRINT
 ELSE
  ADD 1 TO #Y
 END-IF
 AT END OF DATA
  PERFORM PRINT
 END-ENDDATA
END-FIND
/*************************************************************
DEFINE SUBROUTINE PRINT
WRITE NOTITLE (AD=OI) #ARRAY(*)
RESET #ARRAY(*)
SKIP 1
END-SUBROUTINE
/*************************************************************
END
```

```
SMITH                     SMITH                     SMITH
  ENGLANDSVEJ 222           3152 SHETLAND ROAD        14100 ESWORTHY RD.
                            MILWAUKEE                 MONTERREY
  554349                    (414)877-4563             (408)994-2260

  SMITH                     SMITH
  5 HAWTHORN                2307 DARIUS LANE
  OAK BROOK                 TAMPA
  (312)150-9351             (813)131-4010
```

Example 2

```
/* EXAMPLE 'DSREX2'
/* SAMPLE STRUCTURE FOR EXTERNAL SUBROUTINE USING GLOBAL DATA
/************************************************************
/* PROGRAM CONTAINING SUBROUTINE
DEFINE DATA GLOBAL USING GLOBAL-1
            LOCAL  1 FIELD (N7)
END-DEFINE
/* ...
/* ...
/* ...
/*************************************************************
/*  SUBROUTINE 'SUBROUT1'
DEFINE SUBROUTINE SUBROUT1
/* ...
WRITE 'IN SUBROUTINE:' FIELD
/* ...
END-SUBROUTINE
/* ************************************************************
END
```

 Copyright © Software AG 2002

DEFINE WINDOW

```
DEFINE WINDOW window-name
                [ SIZE { AUTO                    } ]
                       { QUARTER                 }
                       { operand1 * operand2     }
                [ BASE { CURSOR                  } ]
                       { { TOP    } { LEFT  }    }
                       { { BOTTOM } { RIGHT }    }
                       { operand3 / operand4    }
                [REVERSED [(CD = background-color)] ]
                [TITLE operand5]
                [ CONTROL { WINDOW } ]
                          { SCREEN }
                [ FRAMED { [ON] [(CD = frame-color)] [position-clause] } ]
                         {                   OFF                       }
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S					N	P	I								yes	no
Operand2	C	S					N	P	I								yes	no
Operand3	C	S					N	P	I								yes	no
Operand4	C	S					N	P	I								yes	no
Operand5	C	S				A											yes	no

Related Statements: SET WINDOW | INPUT WINDOW=’*window-name*’ | SET CONTROL ’W’

Function

The DEFINE WINDOW statement is used to specify the size, position and attributes of a window.

A window is that segment of a logical page, built by a program, which is displayed on the terminal screen. There is always a window present, although you may not be aware of its existence: unless specified differently, the size of the window is identical to the physical size of your terminal 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.

Note:
There is always only **one** Natural window, 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.

Control of Full Screen

Even if a window is active, Natural maintains control of the full screen. This does not affect single-session systems like CICS or TSO; when Natural is running under a multiple-session system like Com-plete or Multi-pass, however, the complete Natural screen - that is, not only the currently active window, but also the Natural screen "underneath" it - will be displayed when a suspended Natural session is resumed; at the same time, the attributes of the full-screen fields partially overlaid by the window will not be influenced by the window.

window-name

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 (see the Natural Reference documentation).

 Copyright © Software AG 2002

SIZE

With the SIZE clause, you specify the size of the window.

- **SIZE AUTO** - The size of the window is determined automatically by Natural at runtime. The size is determined by the data generated into the window as follows:
  The number of window lines will be the number of INPUT lines generated (plus possibly the PF-key lines, message line, and infoline/statistics line).
  The number of window columns is determined by the longest INPUT line: Natural scans, starting from the ends of the lines, for the rightmost significant byte in a line. This may cause an input-only or modifiable field (AD=A or AD=M) to be truncated; to avoid this, you either put a single-character text string after such a field or explicitly set the window size with "SIZE *operand1 * operand2*".
  Note: The title is not part of the window data. Therefore, if the window size has been determined as described above **and** the title is longer than the window, it will be truncated.
- **SIZE QUARTER** - The size of the window will be one quarter of the physical screen.
- **SIZE** *operand1 * operand2* - The size of the window will be *n* lines by *n* columns. The number of lines is determined by *operand1*, the number of columns by *operand2*. Neither of the two operands must contain decimal digits.
  If the window is FRAMED, the specified size will be inclusive of the frame.
  The minimum possible window size is:
  - without frame: 2 lines by 10 columns,
  - with frame: 4 lines by 13 columns.
  The maximum possible window size is the size of the physical screen.

If you omit the SIZE clause, SIZE AUTO applies by default.

Note:
On mainframe computers, Natural requires additional columns for so-called attribute bytes to be able to display data on the screen (on other platforms, such attribute bytes are not needed). Consequently, on mainframe computers the screen area overlaid by a window is wider, and the size of the page segment visible inside a window is smaller than on other platforms. Example: Assume a window whose size is defined as "SIZE 5 * 15" (that is, with a width of 15 columns):

- On mainframe computers, the screen area overlaid by the window is 16 columns; the size of what is visible inside the window is 14 columns without frame, and 10 columns with frame respectively.
- On other platforms, the screen area overlaid by the window is 15 columns; the size of what is visible inside the window is 15 columns without frame, and 13 columns with frame respectively.

BASE

With the BASE clause, you determine the position of the window on the physical screen.

- **BASE CURSOR** places the top left corner of the window at the current cursor position. The cursor position is the *physical* position of the cursor on the screen.
  If the size of the window makes it impossible to place the window at the cursor position, Natural automatically places the window as close as possible to the desired position.
- **BASE TOP/BOTTOM LEFT/RIGHT** places the window at the top-left, bottom-left, top-right, or bottom-right corner respectively of the physical screen.
- **BASE** *operand3/operand4* - This places the top left corner of the window at the specified line/column of the physical screen. The line number is determined by *operand3*, the column number by *operand4*. Neither of the two operands must contain decimal digits.
  If the size of the window makes it impossible to place the window at the specified position, you will get an error message.

If you omit the BASE clause, BASE CURSOR applies by default.

REVERSED

REVERSED will cause the window to be displayed in reverse video (if the screen used supports this feature; if it does not, REVERSED will be ignored).

REVERSED - CD=background-color

This will cause the window to be displayed in reverse video and the background of the window in the specified color (if the screen used supports these features; if it does not, the respective specification will be ignored).

For information on valid color codes, see the session parameter CD in the Natural Reference documentation.

TITLE operand5

With the TITLE clause, you may specify a heading for the window. The specified title (`operand5`) will be displayed centered in the top frame-line of the window. The title can be specified either as a text constant (in apostrophes) or as the content of a user-defined variable. If the title is longer than the window, it will be truncated. The title is only displayed if the window is FRAMED; if FRAMED OFF is specified for the window, the TITLE clause will be ignored.

Note:
If the title contains trailing blanks, these will be removed.
If the first character of the title is a blank, one blank will automatically be appended to the title.

CONTROL

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.

CONTROL WINDOW

CONTROL WINDOW causes the lines to be displayed inside the window.

CONTROL SCREEN

CONTROL SCREEN causes the lines to be displayed on the full physical screen outside the window.

If you omit the CONTROL clause, CONTROL WINDOW applies by default.

 Copyright © Software AG 2002

FRAMED

By default, i.e. if you omit the FRAMED clause, the window is framed. If you specify FRAMED OFF, the framing and everything attached to the frame (window title and position information) will be switched off.

The top and bottom frame lines are 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* below) and then pressing ENTER. 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.

Note:
If the window size is smaller than 4 lines by 12 (or 13 on mainframe computers) columns, the frame will not be visible.

FRAMED - CD=frame-color

This causes the frame of the window to be displayed in the specified color (if the screen used is a color screen; if it is not, the color specification will be ignored).

For information on valid color codes, see the session parameter CD in the Natural Reference documentation.

position-clause

Note:
The POSITION clause is only evaluated on mainframe computers: on all other platforms it is ignored.

```
                 SYMBOL [ TOP    ] [AUTO] [SHORT] [ LEFT  ]
                        [ BOTTOM ]                [ RIGHT ]
POSITION {
                 TEXT [MORE] [ LEFT  ]
                             [ RIGHT ]

                 OFF                                       }
```

The POSITION clause causes information on the position of the window on the logical page 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

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.

AUTO is only applicable if the logical page is fully visible in the window as far as its horizontal size is concerned, that is, if only "-" and/or "+" are to be displayed. In this case, AUTO automatically switches from the symbols to the words "Top", "Bottom" and "More" respectively.

SHORT causes the word "More:" before the symbols "< - + >" to be suppressed.

LEFT/RIGHT determines whether the position information is displayed in the left or right part of the frame line.

POSITION TEXT

POSITION TEXT causes the position information to be displayed in text form. The information is displayed in the top and/or bottom frame line with the words "More", "Top" and "Bottom". The text is language-dependent and may also be displayed in another language if the language code is set accordingly.

POSITION TEXT MORE suppresses the words "Top" and "Bottom" and only displays the word "More" where applicable, i.e., in the top or bottom frame line or both.

LEFT/RIGHT determines whether the position information is displayed in the left or right part of the top frame line.

POSITION OFF

POSITION OFF causes the position information to be suppressed; no position information will be displayed.

Protection of Input Fields in a Window

The following rules apply to input fields (AD=A or AD=M) which are not entirely within the window:

- Input fields whose beginning is not inside the window are always made protected.
- Input fields which begin inside and end outside the window are only made protected if the values they contain cannot be displayed completely in the window. Please note that in this case it is decisive whether the *value length*, not the *field length*, exceeds the window size. Filler characters (as specified with the profile parameter FC) do not count as part of the value.

If you wish to access input fields thus protected, you have to adjust the window size accordingly so that the beginning of the field/end of the value is within the window.

 Copyright © Software AG 2002

Invoking Different Windows

A DEFINE WINDOW statement must not be placed within a logical condition statement block. To invoke different windows depending on a condition, use different SET WINDOW statements (or INPUT statements with a WINDOW clause respectively) in a condition.

Example

```
/* EXAMPLE 'DWDEX1': DEFINE WINDOW
DEFINE DATA LOCAL
01 #I(P3)
END-DEFINE
*
SET KEY PF1='%W<<' PF2='%W>>' PF4='%W--' PF5='%W++'
*
DEFINE WINDOW WIND1
  SIZE QUARTER
  BASE TOP RIGHT
  FRAMED ON POSITION SYMBOL AUTO
*
SET WINDOW 'WIND1'
FOR #I = 1 TO 10
  WRITE 25X #I 'THIS IS SOME LONG TEXT' #I
LOOP
*
END
```

```
> r                                    +-----------------------More:     + >+
  All    ....+....1....+....2....+....3.. ! Page      1                        !
    0010 /* EXAMPLE 'DWDEX1': DEFINE WIND !                                    !
    0020 DEFINE DATA LOCAL               !                             1 THIS  !
    0030 01 #I(P3)                       !                             2 THIS  !
    0040 END-DEFINE                      !                             3 THIS  !
    0050 *                               !                             4 THIS  !
    0060 SET KEY PF1='%W<<' PF2='%W>>' PF !                             5 THIS  !
    0070 *                               !                             6 THIS  !
    0080 DEFINE WINDOW WIND1             !                             7 THIS  !
    0090   SIZE QUARTER                  ! MORE                                !
    0100   BASE TOP RIGHT                +-------------------------------------+
    0110   FRAMED ON POSITION SYMBOL AUTO
    0120 *
    0130 SET WINDOW 'WIND1'
    0140 FOR #I = 1 TO 10
    0150   WRITE 25X #I 'THIS IS SOME LONG TEXT' #I
    0160 END-FOR
    0170 *
    0180 END
    0190
    0200
         ....+....1....+....2....+....3....+....4....+....5....+... S 18   L 1
```

DEFINE WORK FILE

DEFINE WORK FILE *n operand1* **[TYPE** *operand2* **]**

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S				A											yes	no
Operand2	C	S				A											yes	no

Related Statements: WRITE WORK FILE | READ WORK FILE | CLOSE WORK FILE

Function

The DEFINE WORK FILE statement is used to assign a file name to a Natural work file number within a Natural application.

This allows you to make or change work file assignments dynamically within a Natural session or overwrite work file assignments made at another level.

When this statement is executed and the specified work file is already open, the statement will implicitly cause that work file to be closed.

Work File Number - *n*

n is the work file number (1 to 32). This is the number to be used in a WRITE WORK FILE, READ WORK FILE or CLOSE WORK FILE statement.

On mainframe computers, the specified work file must be one that is defined with access method AM=STD (in the NTWORK macro of the Natural parameter module or with the dynamic profile parameter WORK). For work files defined with any other access method, the DEFINE WORK FILE statement is ignored.

Work File Name - *operand1*

operand1 is the name of the work file.

- Work File Name on OpenVMS, UNIX and Windows
- Work File Name on Mainframe Computers

Work File Name on OpenVMS, UNIX and Windows

The file name (*operand1*) may contain environment variables or logicals.

If a file with the specified name does not exist, it will be created.

 Copyright © Software AG 2002

Work File Name on Mainframe Computers

As *operand1* you specify the name of the dataset to be assigned to the work file number.

Operand1 can be 1 to 253 characters long. You can specify either a logical or a physical dataset name.

Work File Type - *operand2*

Operand2 specifies the type of work file.

Operand2 cannot be specified on mainframe computers.

Possible values for *operand2* are:

Work File Type	Usage
DEFAULT	Determine file type from extension for upward compatibility.
TRANSFER	transfer to/from PC with Entire Connection
SAG	binary format
ASCII	"text" files with records terminated by [carriage return] linefeed
ASCII-COMPRESSED	as ASCII, but trailing blanks are removed
ENTIRECONNECTION	READ/WRITE directly in Entire Connection format
UNFORMATTED	no formatting information is written (neither for fields nor for records)
PORTABLE	files which can handle dynamic variables exactly and can also be transported: e.g., from a little endian machine to a big endian machine, and vice versa

The value of *operand2* is handled in a case insensitive way and must be enclosed in quotes or provided in an alphanumeric variable.
Examples:

```
DEFINE WORK FILE 17 #FILE TYPE 'unformatted'
  #TYPE := 'SAG'
  DEFINE WORK FILE 18 #FILE TYPE #TYPE
```

Work File Name under OS/390

Under OS/390, for a work-file number that is defined with the access method AM=STD (whether automatically in the JCL, in the NTWORK macro of the Natural parameter module or dynamically using the profile parameter WORK), *operand1* can be:

- a logical dataset name (DD name, 1 to 8 characters);
- a physical dataset name of a cataloged dataset (1 to 44 characters) or a physical dataset member name;
- a path name and member name of an HFS file (1 to 253 characters) in an MVS UNIX Services environment;
- a JES spool file class;
- "NULLFILE" (to indicate a dummy dataset).

- Logical Dataset Names
- Physical Dataset Names
- HFS Files
- JES Spool File Class

- NULLFILE
- Allocation and De-Allocation of Datasets
- Work Files in Server Environments
- Further Information

Logical Dataset Names

For example:

```
DEFINE WORK FILE 21 'SYSOUT'
```

The specified dataset must have been allocated before the DEFINE WORK FILE statement is executed.

The allocation can be done via JCL, CLIST or dynamic allocation (SVC 99). For dynamic allocation you can use the user exit USR2021 in library SYSEXT.

The dataset name specified in the DEFINE WORK FILE statement overrides the name specified with the subparameter DEST of the NTWORK macro or WORK profile parameter.

Optionally, the dataset name may be prefixed by "DDN=" to indicate that it is a DD name. For example:

```
DEFINE WORK FILE 22 'DDN=XYZ'
```

Physical Dataset Names

For example:

```
DEFINE WORK FILE 23 'TEST.WORK.FILE'
```

The specified dataset must exist in cataloged form. When the DEFINE WORK FILE statement is executed, the dataset is allocated dynamically by SVC 99 with the current DD name and option DISP=SHR.

If the dataset name is 8 characters or shorter and does not contain a period ".", it might be misinterpreted as a DD name. To avoid this, prefix the name with "DSN=". For example:

```
DEFINE WORK FILE 22 'DSN=WORKXYZ'
```

If the dataset is a PDS member, you specify the PDS member name (1 to 8 characters) in parentheses after the dataset name (1 to 44 characters). For example:

```
DEFINE WORK FILE 4 'TEST.WORK.PDS(TEST1)'
```

If the specified member does not exist, a new member of that name will be created.

HFS Files

For example:

```
DEFINE WORK FILE 14 '/u/nat/rec/test.txt'
```

The specified path name must exist. When the DEFINE WORK FILE statement is executed, the HFS file is allocated dynamically. If the specified member does not exist, a new member of that name will be created.

For the dynamic allocation of the dataset, the following OS/390 path options are used:

```
PATHOPTS=(OCREAT,OTRUNC,ORDWR)
  PATHMODE=(SIRUSR,SIWUSR,SIRGRP,SIWGRP)
  FILEDATA=TEXT
```

 Copyright © Software AG 2002

When an HFS file is closed, it is automatically de-allocated by OS/390 (regardless of the setting of the subparameter FREE in the NTWORK macro or WORK profile parameter).

To read an HFS file, you have to use the user exit USR2021 instead of the DEFINE WORK FILE statement, because of the OTRUNC option. This option will reset the HFS file at the first read access and result in an empty file.

JES Spool File Class

To create a JES spool dataset, you specify SYSOUT=x (where x is the desired spool file class). For the default spool file class, you specify SYSOUT=*.

Examples:

```
DEFINE WORK FILE 10 'SYSOUT=A'
  DEFINE WORK FILE 12 'SYSOUT=*'
```

To specify additional parameters for the dynamic allocation, use the user exit USR2021 in the library SYSEXT instead of the DEFINE WORK FILE statement.

NULLFILE

To allocate a dummy dataset, you specify NULLFILE as *operand1*:

```
DEFINE WORK FILE n 'NULLFILE'
```

This corresponds to the JCL definition:

```
// DD-name DD DUMMY
```

Allocation and De-Allocation of Datasets

When the DEFINE WORK FILE statement is executed and a physical dataset name, HFS file, spool file class or dummy dataset has been specified, the corresponding dataset is allocated automatically. If the logical file is already open, it will be closed automatically, except when the profile parameter CLOSE=FIN has been specified, in which case an error will be issued. Moreover, an existing dataset allocated with the same current DD name is automatically de-allocated before the new dataset is allocated. Work files that are to be allocated dynamically have to be predefined in the Natural parameter module with AM=STD.

To avoid unneccessary overhead by unsuccessful premature opening of work files not yet allocated at the start of the program, work files should be defined with the subparameter OPEN=ACC (open at first access) in the NTWORK macro or WORK profile parameter.

In the case of an HFS file, or a work file defined with the subparameter FREE=ON in the NTWORK macro or WORK profile parameter, the work file is automatically de-allocated as soon as it has been closed.

As an alternative for the dynamic allocation and de-allocation of datasets, the user exit USR2021 in the library SYSEXT is provided. This user exit also allows you to specify additional parameters for dynamic allocation.

Work Files in Server Environments

In server environments, errors may occur if multiple Natural sessions attempt to allocate or open a dataset with the same DD name. To avoid this, you either specify the work file with the subparameter DEST=* in the NTWORK macro or WORK profile parameter, or you specify DEFINE WORK FILE '*' in your program before the actual DEFINE WORK FILE statement; Natural then generates a unique DD name at the physical dataset allocation when the first DEFINE WORK FILE statement for that work file is executed.

All work files whose DD names begin with "CM" are shared by all sessions in a server environment. A shared work file opened for output by the first session is physically closed when the server is terminated. A shared work file opened for input is physically closed when the last session closes it, that is, when it receives an end-of-file condition. When a work file is read concurrently, one file record is supplied to one READ WORK FILE statement only.

Further Information

For information on work files, see also Operations for Mainframes.

Work File Name under VM/CMS

Under VM/CMS, the same applies to *operand1* as under OS/390 (see above) but with the following differences:

- Instead of dynamic allocation via MVS SVC 99, the CMS command FILEDEF is used to define a file.
- HFS files are not supported.
- JES spool classes are not supported.
- In addition, the following syntax is used:

  ```
  DEFINE WORK FILE n 'fname ftype fmode (options)'
  ```

  This generates the CMS command:

  ```
  FILEDEF ddname-n DISK fname ftype fmode (options)
  ```

 Copyright © Software AG 2002

- Moreover, the following syntax is allowed:

  `DEFINE WORK FILE n 'FILEDEF=filedef-parameters'`

  This generates the CMS command:

  `FILEDEF ddname-n =filedef-parameters`

  For example:

  `DEFINE WORK FILE 5 'FILEDEF=TAP1 SL 2 VOLID BKUP08 (BLKSIZE 20000)'`

  This generates the CMS command:

  `FILEDEF CMWKF05 TAP1 SL 2 VOLID BKUP08`

Work File Name under BS2000/OSD

Under BS2000/OSD, for a work-file number that is defined with the access method AM=STD (whether automatically in the JCL, in the NTWORK macro of the Natural parameter module or dynamically using the profile parameter WORK), you can use *operand1* to specify a file name or a link name that is allocated to this work file.

In this case, *operand1* can have a length of 1 to 253 characters and one of the following meanings:

- a BS2000/OSD link name (1 to 8 characters)
- a BS2000/OSD file name (9 to 54 characters)
- a generic BS2000/OSD file name (wildcard)
- a BS2000/OSD file name and link name
- a generic BS2000/OSD file name and link name (wildcard)
- *DUMMY

The following rules apply.

1. File name and link name can be specified as positional parameters or keyword parameters. The corresponding keywords are **FILE=** and **LINK=**. Mixing positional and keyword parameters is allowed but not recommended.
2. A string with a length of 1 to 8 characters without commas is interpreted as a link name. This notation is compatible with earlier versions of Natural.

   Example:

   ```
   DEFINE WORK FILE 1 'W01'
   ```

   The corresponding definition with a keyword parameter is:

   ```
   DEFINE WORK FILE 1 'LINK=W01'
   ```

3. A string of 9 to 54 characters without commas is interpreted as a file name.

   Example:

   ```
   DEFINE WORK FILE 2 'NATURAL31.TEST.WORKFILE02'
   ```

   The corresponding definition with a keyword parameter is:

   ```
   DEFINE WORK FILE 2 'FILE=NATURAL31.TEST.WORKFILE02'
   ```

4. The following input is interpreted without considering the length and therefore forms exceptions to Rules 2 and 3:
   - keyword input: LINK=, FILE=
   - *DUMMY
   - NULLFILE (equivalent to *DUMMY)
   - *
   - *,*

   Example: DEFINE WORK FILE 7 'FILE=Y' is a valid file allocation and not a link name, although the string of characters contains fewer than 9 characters.

5. Generic file names are formed as follows:

   Wnn.userid.tsn.date.time.number

 Copyright © Software AG 2002

where
nn is a work-file number
userid is a Natural user-ID, 8 characters
tsn is the BS2000/OSD TSN of the current task, 4 digits
date is DDMMYYYY
time is HHIISS
number is a number, 5 digits

6. Generic link names are formed as follows:

   NWFnnnnn

   nnnnn is a 5-digit number that is increased by one after every generation of a dynamic link name.

7.

   Changing the file allocation for a work-file number causes an implicit CLOSE of the work file allocated so far.

You are strongly recommended, in all cases except when you only specify a link name (for example: W01), to work with keyword parameters. This avoids conflicts of interpretation with additional reports and is essential for file names with fewer than 9 characters.

Example:

```
DEFINE WORK FILE 3 'LINK=#W03'
  DEFINE WORK FILE 3 'FILE=#W03'
```

Link Name

Example:

```
DEFINE WORK FILE 1 'LINKW01'
```

means the same as

```
DEFINE WORK FILE 1 'LINK=LINKW01'
```

A file with the LINK 'LINKW01' must exist at runtime. This can be created either using JCL before starting Natural or by dynamic allocation from the current application. For dynamic allocation, the user exit USR2029 in the library SYSEXT can be used. If, before execution, the link was active on another file, for example: 'W01', this will be released or retained depending on the value of the profile parameter FREE (possible values are ON and OFF). Release is done via an explicit RELEASE call to the BS2000/OSD command processor.

File Name

Example:

```
DEFINE WORK FILE 2 'NATURAL31.TEST.WORK02'
```

means the same as

```
DEFINE WORK FILE 2 'FILE=NATURAL31.TEST.WORK02'
```

The file specified in *operand1* is set up using a FILE macro call and inherits the link name that was valid for the corresponding work file before execution of the DEFINE WORK FILE statement.

Generic File Name

Example:

```
DEFINE WORK FILE 21 '*'
```

means the same as

```
DEFINE WORK FILE 21 'FILE=*'
```

A file with a name created according to Rule 4 is set up using a FILE macro call and inherits the link name that was valid for the corresponding work file before execution of the DEFINE WORK FILE statement.

```
DEFINE WORK FILE 22 'FILE=*,LINK=WFLK22'
```

A file with a name created according to Rule 4 is set up with the specified link name, using a FILE macro call.

File Name and Link Name

Example:

```
DEFINE WORK FILE 11 'NATURAL31.TEST.WORKF11,LNKW11'
```

means the same as

```
DEFINE WORK FILE 11 'FILE=NATURAL31.TEST.WORKF11,LINK=LNKW11'
```

which means the same as

```
DEFINE WORK FILE 11 'FILE=NATURAL31.TEST.WORKF11,LNKW11'
```

The file given in *operand1* is set up with the specified link name, using a FILE macro call and allocated to the corresponding work-file number.

Generic File Name and Link Name

Example:

```
DEFINE WORK FILE 27 '*,*'
```

means the same as

```
DEFINE WORK FILE 27 'FILE=*,LINK=*'
```

A file with a file name and link name created according to Rule 4 and Rule 5 is set up using a FILE macro call and allocated to the specified work file 27.

Note:
When file name and link name are specified, the previous link name is not released, regardless of the value of the profile parameter FREE.

 Copyright © Software AG 2002

DELETE

```
DELETE [RECORD] [IN] [STATEMENT] [(r)]
```

Related Statements: END TRANSACTION | BACKOUT TRANSACTION | STORE | UPDATE

Function

The DELETE statement is used to delete a record from a database.

Considerations for DL/I Databases

The DELETE statement is used to delete a segment from a DL/I database, which also results in the deletion of all descendants of the segment.

Due to GSAM restrictions, the UPDATE statement cannot be used for GSAM databases.

Considerations for SQL Databases

The DELETE statement is used to delete a row from the database table. It corresponds with the SQL statement DELETE WHERE CURRENT OF CURSOR-NAME, that is, only the row which was read last can be deleted.

With most SQL databases, a row that was read with a FIND SORTED BY or READ LOGICAL statement cannot be deleted.

Considerations for VSAM Databases

The DELETE statement is not valid for VSAM entry-sequenced datasets (ESDS).

Statement Reference - r

The "(r)" notation is used to reference the statement which was used to select/read the record to be deleted.

If no statement reference is specified, the DELETE statement will reference the innermost active processing loop in which a database record was selected/read.

Note:
The DELETE statement must be placed within the READ or FIND loop it references.

Restriction

A DELETE statement cannot be specified in the same statement line as a FIND, READ, or GET statement.

Copyright © Software AG 2002

Hold Status

The use of the DELETE statement causes each record selected in the corresponding FIND or READ statement to be placed in hold status.

Record hold logic is explained in the section Database Access of the Natural Programming Guide.

Example 1

In this example, all records with the name = 'ALDEN' are deleted.

```
/* EXAMPLE 'DELEX1S': DELETE (STRUCTURED MODE)
/*****************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
END-DEFINE
/*****************************************************
FIND EMPLOY-VIEW WITH NAME = 'ALDEN'
 DELETE
 END TRANSACTION
/*****************************************************
 AT END OF DATA
  WRITE NOTITLE *NUMBER 'RECORDS DELETED'
 END-ENDDATA
/*****************************************************
END-FIND
END
```

Equivalent reporting-mode example: See the program DELEX1R in the library SYSEXRM.

Example 2

If no records are found in the VEHICLES file for the person named ALDEN, the EMPLOYEE record for ALDEN is deleted.

 Copyright © Software AG 2002

```
/* EXAMPLE 'DELEX2S:' DELETE (STRUCTURED MODE)
/*************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
1 VEHIC-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
END-DEFINE
/*************************************************************
EMPL.  FIND EMPLOY-VIEW WITH NAME = 'ALDEN'
/*************************************************************
VEHC.   FIND VEHIC-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (EMPL.)
         IF NO RECORDS
          DELETE (EMPL.)
          END TRANSACTION
         END-NOREC
        END-FIND
/*************************************************************
       END-FIND
/*************************************************************
END
```

Equivalent reporting-mode example: See the program DELEX2R in the library SYSEXRM.

DISPLAY

```
DISPLAY [(rep)] [options] { [/...] [output-format] output-element }...
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S	A	G	N	A	N	P	I	F	B	D	T	L		G	O	yes	no

Related Statements: WRITE | WRITE TITLE | WRITE TRAILER

Function

The DISPLAY statement is used to specify the fields to be output on a report in column format. A column is created for each field and a field header is placed over the column.

Report Specification - rep

The notation (*rep*) may be used to specify the identification of the report for which the DISPLAY statement is applicable. A value in the range 0 - 31 or a logical name which has been assigned using the DEFINE PRINTER statement may be specified. If *(rep)* is not specified, the DISPLAY statement will apply to the first report (report 0).

options

```
[NOTITLE] [NOHDR] [ [AND] [GIVE] [SYSTEM] FUNCTIONS ] [(statement-parameters)]
```

Page Title/NOTITLE

By default, Natural generates a single title line for each page resulting from a DISPLAY statement. This title contains the page number, the time of day, and the date. Time of day is set at the beginning of the program execution (TP mode) or at the beginning of the job (batch mode).

The default title line may be overridden by using a WRITE TITLE statement, or it may be suppressed by specifying the keyword NOTITLE in the DISPLAY statement.

Default title will be produced:

```
DISPLAY NAME
```

 Copyright © Software AG 2002

User title will be produced:

```
DISPLAY NAME
WRITE TITLE 'USER TITLE'
```

No title will be produced:

```
DISPLAY NOTITLE NAME
```

If the NOTITLE option is used, it applies to all DISPLAY, PRINT and WRITE statements within the same object which write data to the same report.

Column Headers/NOHDR

Column headers are produced for each field specified in the DISPLAY statement using the following rules:

The header text may be explicitly specified in the DISPLAY statement before the field name. For example:

```
DISPLAY 'EMPLOYEE' NAME 'SALARY' SALARY
```

If you do not specify an explicit header for a field, the header as defined in the DEFINE DATA statement will be used. If for a database field no header is defined in the DEFINE DATA statement, the default header as defined in the DDM will be used; if no default header is defined in the DDM, the field name will be used as header. If for a user-defined variable no header is defined in the DEFINE DATA statement, the variable name will be used as header. See also the DEFINE DATA statement for header definition.

```
DISPLAY NAME SALARY #NEW-SALARY
```

Natural always underlines column headings and generates one blank line between the underlining and the data being displayed.

If there are multiple DISPLAY statements in a program, the first DISPLAY statement determines the column header(s) to be used; this is evaluated at compilation time.

Suppressing Column Headers

To suppress the column header for a single field, specify the characters '/' (apostrophe-slash-apostrophe) before the field name. For example:

```
DISPLAY '/' NAME 'SALARY' SALARY
```

To suppress all column headers, specify the keyword NOHDR:

```
DISPLAY NOHDR NAME SALARY
```

NOHDR only takes effect for the first DISPLAY, as subsequent DISPLAY statements cannot create column headers anyhow.

If both NOTITLE and NOHDR are used, they must be specified in the following order:

```
DISPLAY NOTITLE NOHDR NAME SALARY
```

 Copyright © Software AG 2002

GIVE SYSTEM FUNCTIONS

The GIVE SYSTEM FUNCTIONS clause is used to make available the Natural system functions AVER, COUNT, MAX, MIN, NAVER, NCOUNT, NMIN, SUM, TOTAL. These are evaluated when the DISPLAY statement containing the GIVE SYSTEM FUNCTIONS clause is executed.

These functions may then be referred to in a statement executed as a result of an end-of-page condition.

Only one DISPLAY statement per report may contain a GIVE SYSTEM FUNCTIONS clause. When system functions are evaluated from a DISPLAY statement, they are evaluated on a page basis, which means that all functions (except TOTAL) are reset to zero when a new page is initiated.

When system functions are used within a DISPLAY statement within a subroutine, the end-of-page processing must occur within the same routine.

statement-parameters

One or more parameters, enclosed within parentheses, may be specified.

Each parameter specified will override any previous parameter specified in a GLOBALS command, SET GLOBALS or FORMAT statement. If more than one parameter is specified, they must be separated by one or more blanks from one another. Each parameter specification must not be split between two statement lines.

The parameter settings appiled here will only be regarded for variable fields, but they have no effect on text-constants. If you would like to set field attributes for a text-constant, they have to be set explicitly for this element.

Example:

```
DEFINE DATA LOCAL
1 VARI (A4)     INIT <'1234'>                            /*     Output
END-DEFINE                                               /*    Produced
*                                                        /*    ---------
DISPLAY NOHDR           'Text'         '='   VARI        /*    Text 1234
DISPLAY NOHDR (PM=I)    'Text'         '='   VARI        /*    Text 4321
DISPLAY NOHDR           'Text' (PM=I)  '='   VARI (PM=I) /*    txeT 4321
DISPLAY NOHDR           'Text' (PM=I)  '='   VARI        /*    txeT 1234
END
```

For information on which parameters you may specify and a description of each parameter, see the section Session Parameters in the Natural Reference documentation.

Line Advance - Slash

When specified within a text element, a slash "/" causes a line advance for the text displayed.

When specified between output elements, it causes the output element specified by "/" to be placed vertically within the same column. The header for this column will be constructed by placing the headers of the vertically displayed elements vertically above the column.

output-format

```
[ nX          ]
[ nT          ]
[ x/y         ]
[ T*field-name ]
[ P*field-name ]
[ 'text' [(attributes)] ]
[ 'c' (n) [(attributes)] ]...

[ VERTICALLY [AS { 'text' [(attributes)] [CAPTIONED] } ] [/...] ]
[                { [CAPTIONED]                       }
[ HORIZONTALLY ]
```


Copyright © Software AG 2002

Field Positioning Notations

*n***X**	Example: DISPLAY NAME **5X** SALARY **Note:** (for Mainframes Only) This notation inserts n spaces between columns. n must not be "0".
*n***T**	This notation causes positioning (tabulation) to display position n. Backward positioning is not permitted. In the following example, NAME is displayed beginning in position 25, and SALARY beginning in position 50: DISPLAY **25T** NAME **50T** SALARY
x/y	This notation causes the next element to be placed *x* lines below the output of the last statement, beginning in column *y*. *y* must not be "0". Backward positioning is not permitted.
T***field-name*	This notation is used to position to a specific print position of a *field* used in a previous DISPLAY statement. Backward positioning is not permitted.
P***field-name*	This notation is used to position to a specific print position *and line* of a *field* used in a previous DISPLAY statement. It is most often used in conjunction with vertical display mode. Backward positioning is not permitted.

Override Column Heading Assignment

'='	If placed immediately before the field, '=' indicates that either the default header specified for the field in the DDM is to be used as header, or, if no default header is specified, the field name is to be used as header.
'*text*'	If placed immediately before a field, text overrides the column heading. The character '/' before a field causes the header for the field to be suppressed. DISPLAY **'EMPLOYEE'** NAME **'MARITAL/STATUS'** MAR-STAT If multiple text elements are specified before a field name, the last text element will be used as the column header and the other text elements will be placed before the value of the field within the column.
'*c*'*(n)*	The character c is displayed n times immediately before the field value. DISPLAY **'*' (5)** '=' NAME

attributes

Indicates the display and color attributes to be used for text display. Attributes may be:

1	2
B	BL
C	GR
D	NE
I	PI
N	RE
U	TU
V	YE

Copyright © Software AG 2002

1. Display attributes (see the session parameter AD in the Natural Reference documentation).
2. Color attributes (see the session parameter CD in the Natural Reference documentation).

Vertical/Horizontal Display

The VERT clause may be used to cause multiple field values to be positioned underneath one another in the same column. In vertical mode, a new column may be initiated by specifying the keyword VERT or HORIZ.

The column heading in vertical mode is controlled using the entry or entries specified with the AS clause as described below.

- No column heading is produced if the AS clause is omitted.
  DISPLAY **VERT** NAME SALARY
- If AS 'text' is specified, text is used as the column heading. The character "/" in the character string of text will cause multiple lines of column headings.
  DISPLAY **VERT AS 'LAST/NAME'** NAME
- If AS 'text' CAPTIONED is specified, text is used as the column heading and the standard heading text or field name is inserted immediately before the field value in each detail display line.
  DISPLAY **VERT AS 'PERSONS/SELECTED' CAPTIONED** NAME FIRST-NAME
- If AS CAPTIONED is specified, the standard heading text for the field (either heading text or the field name) will be used as the column heading.
  DISPLAY **VERT AS CAPTIONED** NAME FIRST-NAME

Vertical and horizontal column orientation may be intermixed by using the respective keyword.

To suspend vertical display for a single output element, you may place a dash "-" in front of the element. For example:

```
DISPLAY VERT NAME - FIRST-NAME SALARY
```

In the above example, FIRST-NAME will be output horizontally next to NAME, while SALARY will be output vertically again, i.e. below NAME.

The standard display mode is horizontal. A column is constructed for each field to be displayed.

Column headings are obtained and used by Natural according to the following priority:

1. heading 'text' supplied in the DISPLAY statement;
2. the default heading defined in the DDM (database fields), or the name of a user-defined variable;
3. the field name as defined in the DDM
   (if no heading text was defined for the database field).

For group names, a group heading is produced for the entire group. When specifying a group, only the heading for the entire group may be overridden by a user-specified heading.

The maximum number of column header lines is 15.

Line size overflow is not permitted for output resulting from a DISPLAY statement. If a line overflow occurs, an error message is issued.

output-element

```
[ { 'text' [(attributes)] }
  { 'c' (n) [(attributes)] }...
            nX
            nT
            x/y              ] ['=' {operand1 [(parameters)]}
```

 Copyright © Software AG 2002

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S	A	G	N	A	N	P	I	F	B	D	T	L		G	O	yes	no

*n***X**	This is the same as under *output-format* (see above).
*n***T**	This is the same as under *output-format* (see above).
x/y	This is the same as under *output-format* (see above).
’*text*’	This is the same as under *output-format* (see above).
’*c*’ *(n)*	This is the same as under *output-format* (see above).
’*text*’ ’=’	If ’*text*’ ’=’ is placed immediately before the field, *text* is output immediately before the field value. DISPLAY **’*****’ ’=’** NAME
attributes	This is the same as under *output-format* (see above).
operand1	The field to be displayed. *Note for DL/I databases:* *The DL/I AIX fields can be displayed only if a PCB is used with the AIX specified in the parameter PROCSEQ. If not, an error message is returned by Natural at runtime.*
parameters	One or more parameters, enclosed within parentheses, may be specified immediately after *operand1*. Each parameter specified in this manner will override any previous parameter specified in a GLOBALS command, SET GLOBALS or FORMAT statement. If more than one parameter is specified, one or more blanks must be placed between each entry. An entry must not be split between two statement lines.

Defaults

The following defaults are applicable for a DISPLAY statement:

1. The width of the report defaults to the value set when Natural is installed. This default value is normally 132 in batch mode or the line length of the terminal in TP mode. It may be overridden with the session parameter LS. In TP mode, line size (LS) and page size (PS) parameters are set by Natural based on the physical characteristics of the terminal type in use.
2. When the DISPLAY output is displayed on a terminal screen, the output begins in physical column 2 (because column 1 must be reserved for possible use as an attribute position on a 3270-type terminal). When the DISPLAY output is printed on paper, the printout begins in the leftmost column (column 1).
3. The default spacing factor between elements is one position. There is a minimum of one space between columns (reserved for terminal attributes). This default may be overridden with the session parameter SF.
4. The length of the field or the field heading, whichever is greater, determines the column width for the report (unless the HW parameter is used). If the field is longer than the heading, the heading will be centered over the column unless the HC=L or HC=R parameter is used to produce a left-justified or right-justified heading. If the heading is longer than the field, the field will be left-justified under the heading. The values contained in the field are left-justified for alphanumeric fields and right-justified for numeric fields. Numeric fields may be displayed left-justified by specifying AD=L. Alphanumeric fields may be displayed right-justified by specifying AD=R. In a vertical display, the longest data value or heading among all fields determines the column width (unless the HW parameter is used).

5. One extra high-order print position is reserved for a sign when printing a numeric field. The session parameter SG may be used to suppress the sign position.
6. Page overflow is checked before execution of a DISPLAY statement. No new page title or trailer information is generated during the execution of a DISPLAY statement.

Example 1

```
/* EXAMPLE 'DISEX1:' DISPLAY (USING NX, NT NOTATION)
/**************************************************
LIMIT 4
READ EMPLOYEES BY NAME
 DISPLAY NOTITLE 5X NAME 50T JOB-TITLE
/**************************************************
END
```

```
            NAME                                        CURRENT
                                                         POSITION
     --------------------                        -------------------------

     ABELLAN                                     MAQUINISTA
     ACHIESON                                    DATA BASE ADMINISTRATOR
     ADAM                                        CHEF DE SERVICE
     ADKINSON                                    SALES PERSON
```

 Copyright © Software AG 2002

Example 2

```
/* EXAMPLE 'DISEX2' DISPLAY (GIVE SYSTEM FUNCTIONS)
/************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
  2 SALARY (1)
  2 CURR-CODE (1)
END-DEFINE
/************************************************************
LIMIT 15
FORMAT PS=15
READ EMPLOY-VIEW
 DISPLAY GIVE SYSTEM FUNCTIONS
         PERSONNEL-ID NAME FIRST-NAME SALARY (1) CURR-CODE (1)
   AT END OF PAGE
      WRITE / 'SALARY STATISTICS:'
            / 7X 'MAXIMUM:' MAX(SALARY(1)) CURR-CODE (1)
            / 7X 'MINIMUM:' MIN(SALARY(1)) CURR-CODE (1)
            / 7X 'AVERAGE:' AVER(SALARY(1)) CURR-CODE (1)
    END-ENDPAGE
END-READ
/************************************************************
END
```

```
PAGE      1                                                    97-04-02  14:29:16


  PERSONNEL         NAME              FIRST-NAME        ANNUAL   CURRENCY
     ID                                                   SALARY     CODE
  --------- -------------------- -------------------- ---------- --------

  50005600  MORENO               HUMBERTO                 165810 FRA
  50005500  BLOND                ALEXANDRE                172000 FRA
  50005300  MAIZIERE             ELISABETH                166900 FRA
  50004900  CAOUDAL              ALBERT                   167350 FRA
  50004600  VERDIE               BERNARD                  170100 FRA
  50004300  GUERIN               MICHELE                  163900 FRA
  50004200  VAUZELLE             BERNARD                  159790 FRA
  50004100  CHAPUIS              ROBERT                   169900 FRA
  50004000  MONTASSIER           JEAN                     175550 FRA

  SALARY STATISTICS:
         MAXIMUM:     175550 FRA
         MINIMUM:     159790 FRA
         AVERAGE:     167922 FRA
```

Example 3

```
* EXAMPLE 'DISEX3': DISPLAY (USING P* NOTATION)
**********************************************
DEFINE DATA LOCAL
 1 EMP-VIEW VIEW OF EMPLOYEES
   2 NAME
   2 SALARY (1)
   2 BIRTH
   2 CITY
END-DEFINE
*
LIMIT 2
READ EMP-VIEW BY CITY FROM 'N'
 DISPLAY NOTITLE NAME CITY
         VERT AS 'BIRTH/SALARY' BIRTH (EM=YYYY-MM-DD) SALARY (1)
 SKIP 1
 AT BREAK  OF CITY
   DISPLAY P*SALARY (1) AVER(SALARY (1))
   SKIP 1
END-BREAK
END-READ
END
```

```
        NAME                 CITY           BIRTH
                                            SALARY
-------------------- -------------------- ----------

WILCOX               NASHVILLE            1970-01-01
                                               38000

MORRISON             NASHVILLE            1949-07-10
                                               36000


                                               37000
```

 Copyright © Software AG 2002

Example 4

```
/* EXAMPLE 'DISEX4:' DISPLAY (USING 'TEXT', 'C(N)' NOTATION, AND
/*                            ATTRIBUTE NOTATION)
/*****************************************************************
LIMIT 4
READ EMPLOYEES BY DEPT FROM 'T'
 IF LEAVE-DUE GT 40
  DISPLAY NOTITLE 'EMPLOYEE' NAME       /* OVERRIDE STANDARD HEADER
          'LEAVE ACCUMULATED' LEAVE-DUE /* OVERRIDE STANDARD HEADER
          '*'(10)(I)                    /* DISPLAY 10 '*' INTENSIFIED
 ELSE
  DISPLAY NAME LEAVE-DUE
/*****************************************************************
END
```

```
    EMPLOYEE       LEAVE ACCUMULATED
-------------------- -----------------

LAVENDA                33
BOYER                  33
CORREARD               45                **********
BOUVIER                19
```

Example 5

```
/* EXAMPLE 'DISEX5': DISPLAY (HORIZONTAL DISPLAY)
/*************************************************************
LIMIT 4
READ EMPLOYEES BY NAME
 DISPLAY NOTITLE NAME JOB-TITLE SALARY (1:2) 'CURR-CODE (1:2)
 SKIP 1
/*************************************************************
END
```

```
      NAME                  CURRENT            ANNUAL   CURRENCY
                             POSITION            SALARY     CODE
-------------------- ------------------------- ---------- --------

ABELLAN              MAQUINISTA                   1450000 PTA
                                                  1392000 PTA

ACHIESON             DATA BASE ADMINISTRATOR        10500 UKL
                                                    11300 UKL

ADAM                 CHEF DE SERVICE               159980 FRA
                                                        0

ADKINSON             SALES PERSON                   36000 USD
                                                    33100 USD
```

 Copyright © Software AG 2002

Example 6

```
/* EXAMPLE 'DISEX6': DISPLAY (VERTICAL AND HORIZONTAL DISPLAY)
/*************************************************************
LIMIT 1
READ EMPLOYEES BY NAME
 DISPLAY NOTITLE VERT AS CAPTIONED
         NAME CITY 'POSITION' JOB-TITLE
         HORIZ 'SALARY' SALARY (1:2) 'CURRENCY' CURR-CODE (1:2)
/*************************************************************
 SKIP 1
END
```

```
        NAME              SALARY   CURRENCY
          CITY
        POSITION
------------------------- ---------- --------

ABELLAN                      1450000 PTA
MADRID                       1392000 PTA
MAQUINISTA
```

Example 7

```
/* EXAMPLE 'DISEX7': DISPLAY (USING STATEMENT/ELEMENT PARAMETERS)
/*****************************************************************
LIMIT 3
READ EMPLOYEES BY NAME
 DISPLAY NOTITLE (AL=16 GC=+ NL=8 SF=3 UC==)
         PERSONNEL-ID NAME TELEPHONE (LC=< TC=>)
/*****************************************************************
END
```

```
PERSONNEL            NAME         +++++++++++++++TELEPHONE++++++++++
      ID
                                          AREA              TELEPHONE
                                          CODE
================   ================   ==================   =============

60008339           ABELLAN            <1     >             <4356726   >
30000231           ACHIESON           <0332  >             <523341    >
50005800           ADAM               <1033  >             <44864858  >
```

 Copyright © Software AG 2002

DIVIDE

```
DIVIDE [ROUNDED] operand1 INTO operand2 [GIVING operand3]
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S	A		N		N	P	I	F								yes	no
Operand2	C	S	A		NM*		N	P	I	F								yes	no
Operand3		S	A			A	N	P	I	F	B							yes	yes

* "N" if GIVING clause is used, "M" if GIVING clause is not used.

```
DIVIDE operand1 INTO operand2 [GIVING operand3] REMAINDER operand4
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S	A		N		N	P	I									yes	no
Operand2	C	S	A		N		N	P	I									yes	no
Operand3		S	A			A	N	P	I	F	B							yes	yes
Operand4		S	A			A	N	P	I	F	B							yes	yes

Related Statement: COMPUTE

Function

The DIVIDE statement is used to divide two operands.

Result Field

The result field may be a database field or a user-defined variable.

The ROUNDED clause causes the result to be rounded.

If the keyword GIVING is used, *operand2* will not be modified and the result will be stored in *operand3*; if the GIVING clause is not used, the result will be stored in *operand2*. If *operand2* is a constant or a non-modifiable Natural system variable, the GIVING clause is required.

If a database field is used as the result field, the division only results in an update to the internal value of the field as used within the program. The value for the field in the database remains unchanged.

The number of decimal positions for the result of the division is evaluated from the result field (that is, *operand2* if no GIVING clause is used, or *operand3* if the GIVING clause is used).

For the precision of the result, see also Precision of Results for Arithmetic Operations in the Natural Reference documentation.

Division by Zero

If an attempt is made to use a divisor (*operand1*) which is "0", either an error message or a result equal to "0" will be returned; this depends on the setting of the session parameter ZD (which is described in the Natural Reference documentation).

REMAINDER Option

If the keyword REMAINDER is specified, the remainder of the division will be placed into the specified field (*operand4*).

If GIVING and REMAINDER are used, none of the four operands may be an array range.

Internally, the remainder is computed as follows:
1. The quotient of the division of *operand1* into *operand2* is computed.
2. The quotient is multiplied by *operand1*.
3. The product of this multiplication is subtracted from *operand2*.
4. The result of this subtraction is assigned to *operand4*.
For each of these steps, the rules described under Precision of Results for Arithmetic Operations in the Natural Reference documentation apply.

 Copyright © Software AG 2002

Example

```
/* EXAMPLE 'DIVEX1': DIVIDE
/************************************************************
DEFINE DATA LOCAL
1 #A (N7) INIT <20>
1 #B (N7)
1 #C (N3.2)
1 #D (N1)
1 #E (N1) INIT <3>
1 #F (N1)
END-DEFINE
/***********************************************************
DIVIDE 5 INTO #A
WRITE NOTITLE 'DIVIDE 5 INTO #A' 20X '=' #A
/***********************************************************
RESET INITIAL #A
DIVIDE 5 INTO #A GIVING #B
WRITE 'DIVIDE 5 INTO #A GIVING #B' 10X '=' #B
/***********************************************************
DIVIDE 3 INTO 3.10 GIVING #C
WRITE 'DIVIDE 3 INTO 3.10 GIVING #C' 8X '=' #C
/***********************************************************
DIVIDE 3 INTO 3.1 GIVING #D
WRITE 'DIVIDE 3 INTO 3.1 GIVING #D' 9X '=' #D
/***********************************************************
DIVIDE 2 INTO #E REMAINDER #F
WRITE 'DIVIDE 2 INTO #E REMAINDER #F' 7X '=' #E '=' #F
/***********************************************************
END
```

```
DIVIDE 5 INTO #A                    #A:        4
DIVIDE 5 INTO #A GIVING #B          #B:        4
DIVIDE 3 INTO 3.10 GIVING #C        #C:        1.03
DIVIDE 3 INTO 3.1 GIVING #D         #D:        1
DIVIDE 2 INTO #E REMAINDER #F       #E:        1 #F:  1
```

DO/DOEND

Note:
The DO and DOEND statements are only valid in reporting mode.

```
DO statement...DOEND
```

Function

The statements DO and DOEND are used in reporting mode to specify a group of statements to be executed based on a logical condition as specified in any of the following statements:

- AT BREAK
- AT END OF DATA
- AT END OF PAGE
- AT START OF DATA
- AT TOP OF PAGE
- BEFORE BREAK PROCESSING
- FIND ... IF NO RECORDS FOUND
- IF
- IF SELECTION
- ON ERROR
- READ WORK FILE ... AT END OF FILE

Restrictions

WRITE TITLE, WRITE TRAILER, and AT condition statements are not permitted **within** a DO/DOEND statement group.

A loop-initiating statement may be used within a DO/DOEND statement group provided that the loop is closed prior to the DOEND statement.

Example

See the program DOEEX1 in the library SYSEXRM.

 Copyright © Software AG 2002

DOWNLOAD

This statement is only available with Natural Connection. It is described in the Natural Connection documentation.

EJECT

- Syntax 1
- Syntax 2

Syntax 1

```
EJECT { ON } [(rep)]
      { OFF }
```

Function

EJECT ON/OFF With Report Specification - Online and Batch Modes

"EJECT OFF *(rep)*" causes *no* page advance (except as specified with Syntax 2 of the EJECT statement) for the specified report to be executed.

"EJECT ON *(rep)*" causes page advances for the specified report to be executed.

EJECT ON/OFF Without Report Specification - Batch Mode only

EJECT ON/OFF - without (*rep*) notation - may be used in batch mode to control page ejection between the output listings created during the execution of a program.

EJECT ON (default) causes Natural to generate a page eject between the source program listing, the output report and the message "EXECUTION COMPLETED".

EJECT OFF causes Natural to suppress page breaks between the above output. EJECT OFF remains in effect until revoked with a subsequent EJECT ON statement.

Report Specification - *rep*

The notation *(rep)* may be used to specify the identification of the report for which the EJECT statement is applicable. A value in the range 0 - 31 or a logical name which has been assigned using the DEFINE PRINTER statement may be specified. If *(rep)* is not specified, the EJECT statement will be applicable to the first report (report 0).

Syntax 2

```
EJECT [(rep)] [ [ IF ] LESS [THAN] operand1 [LINES] [LEFT] ]
              [ [WHEN]                                     ]
```

 Copyright © Software AG 2002

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S					N	P	I								yes	no

Function

This form of the EJECT statement may be used to cause a page advance without a title or heading line being generated on the next page and without TOP/END PAGE processing.

Report Specification - *rep*

The notation *(rep)* may be used to specify the identification of the report for which the EJECT statement is applicable. A value in the range 0 - 31 or a logical name which has been assigned using the DEFINE PRINTER statement may be specified. If *(rep)* is not specified, the EJECT statement will be applicable to the first report (report 0).

IF LESS THAN *operand1* LINES LEFT

A page advance will be performed only when the current line for the page is greater than the page size minus *operand1*. The value for *operand1* may be specified as a numeric constant or as a variable.

Processing

The execution of an EJECT statement does not cause any statements used with an AT TOP OF PAGE, AT END OF PAGE, WRITE TITLE or WRITE TRAILER statement to be executed. It does not affect system functions evaluated by DISPLAY GIVE SYSTEM FUNCTIONS.

EJECT causes a new physical page only. It causes the Natural system variable *LINE-COUNT to be set to "1" but has no effect on the setting of the Natural system variable *PAGE-NUMBER.

Example

```
/* EXAMPLE 'EJTEX1:' EJECT
/************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
END-DEFINE
/************************************************************
FORMAT PS=15
LIMIT 9
READ EMPLOY-VIEW BY CITY
/************************************************************
 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) /
   EJECT
 END-START
 EJECT WHEN LESS THAN 3 LINES LEFT
/************************************************************
 WRITE '*' (64)
 DISPLAY NOTITLE NOHDR CITY NAME JOB-TITLE 5X *LINE-COUNT
 WRITE '*' (64)
END-READ
END
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                          %%
%%   REPORT OF EMPLOYEES    %%
%%     SORTED BY CITY       %%
%%                          %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

 Copyright © Software AG 2002

```
*****************************************************************
  AIKEN                SENKO                PROGRAMMER                    2
  ****************************************************************
  ****************************************************************
  AIX EN OTHE.         GODEFROY             COMPTABLE                     5
  ****************************************************************
  ****************************************************************
  AJACCIO              CANALE               CONSULTANT                    8
  ****************************************************************
  ****************************************************************
  ALBERTSLUND          PLOUG                KONTORASSISTENT              11
  ****************************************************************
  ****************************************************************
  ALBUQUERQUE          HAMMOND              SECRETARY                    14
  ****************************************************************
```

```
*****************************************************************
  ALBUQUERQUE          ROLLING              MANAGER                       2
  ****************************************************************
  ****************************************************************
  ALBUQUERQUE          FREEMAN              MANAGER                       5
  ****************************************************************
  ****************************************************************
  ALBUQUERQUE          LINCOLN              ANALYST                       8
  ****************************************************************
  ****************************************************************
  ALFRETON             GOLDBERG             JUNIOR                       11
  ****************************************************************
```

END

```
{END}
{ . }
```

Function

The END statement is used to mark the physical end of a Natural program.

In reporting mode, any processing loop which is currently active (that is, which has not been closed with a LOOP statement) is closed by the END statement.

period - .

If a period (.) is used instead of END, it must be preceded by at least one blank if other statements are contained in the same line.

Considerations for Program Execution

When an END statement is executed in a main program (that is, a program executing on level 1), final end-page processing is performed as well as final break processing for user-initiated breaks (PERFORM BREAK PROCESSING) which have not been associated with a processing loop by specifying a reference *(r)* notation.

When an END statement is executed in a subprogram, or in a program invoked with FETCH RETURN, control will be returned to the invoking program without any final processing.

Examples

See any program in this section.

 Copyright © Software AG 2002

END TRANSACTION

```
END [OF] TRANSACTION [operand1 ...]
```

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
Operand1	C	S			N	A	N	P	I	F	B	D	T						yes	no

Related Statements: GET TRANSACTION DATA | BACKOUT TRANSACTION | STORE | UPDATE | DELETE

Function

The END TRANSACTION statement is used to indicate the end of a logical transaction. A logical transaction is the smallest logical unit of work (as defined by the user) which must be performed in its entirety to ensure that the information contained in the database is logically consistent.

Successful execution of an END TRANSACTION statement ensures that all updates performed during the transaction have been or will be physically applied to the database regardless of subsequent user, Natural, database or operating system interruption. Updates performed within a transaction for which the END TRANSACTION statement has not been successfully completed will be backed out automatically.

The END TRANSACTION statement also results in the release of all records placed in hold status during the transaction.

The END TRANSACTION statement can be executed based upon a logical condition.

For further information, see the section Database Access in the Natural Programming Guide.

Databases Affected

An END TRANSACTION statement *without* transaction data (that is, without *operand1*) will only be executed if a database transaction under control of Natural has taken place. Depending on the setting of the Natural profile parameter ET (see your Natural Operations documentation) the statement will be executed only for the database affected by the transaction (ET=OFF), or for all databases that have been referenced since the last execution of a BACKOUT TRANSACTION or END TRANSACTION statement (ET=ON).

An END TRANSACTION statement *with* transaction data (that is, with specifying *operand1*) will always be executed and the transaction data be stored in a database as described in the following section. It depends on the setting of the ET parameter (see above) for which other databases the END TRANSACTION statement will be executed.

Storage of Transaction Data - operand1

For a transaction applied to an Adabas database, or to a DL/I database in a batch-oriented BMP region (in IMS environments only), you may also use this statement to store transaction-related information. These transaction data must not exceed 2000 bytes. They may be read with a GET TRANSACTION DATA statement.

The transaction data are written to the database specified with the profile parameter ETDB (see your Natural Operations documentation).
If the ETDB parameter is not specified, the transaction data are written to the database specified with the profile parameter UDB - except on mainframe computers: here, they are written to the database where the Natural Security system file (FSEC) is located (if FSEC is not specified, it is considered to be identical to the Natural system file, FNAT; if Natural Security is not installed, the transaction data are written to the database where FNAT is located).

Considerations for DL/I Databases

Because PSB scheduling is terminated by a "syncpoint" request, Natural saves the PSB position before executing the END TRANSACTION statement. Before the next command execution, Natural re-schedules the PSB and tries to set the PCB position as it was before the END TRANSACTION statement. The PCB position might be shifted forward if any pointed segment had been deleted in the time period between the END TRANSACTION and the following command.

Considerations for SQL Databases

As most SQL databases close all cursors when a logical unit of work ends, an END TRANSACTION statement must not be placed within a database modification loop; instead, it has to be placed after such a loop.

Considerations for VSAM Databases

For information on the transaction logic that applies when accessing VSAM, see the Natural for VSAM documentation.

Restriction

This statement cannot be used with ENTIRE SYSTEM SERVER.

 Copyright © Software AG 2002

Example 1

```
/* EXAMPLE 'ETREX1S:' END TRANSACTION (STRUCTURED MODE)
/*******************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 COUNTRY
END-DEFINE
/*******************************************************
FIND EMPLOY-VIEW WITH CITY = 'BOSTON'
 ASSIGN COUNTRY = 'USA'
 UPDATE
 END TRANSACTION
/*******************************************************
 AT END OF DATA
  WRITE NOTITLE *NUMBER 'RECORDS UPDATED'
 END-ENDDATA
/*******************************************************
END-FIND
END
```

```
7 RECORDS UPDATED
```

Equivalent reporting-mode example: See the program ETREX1R in the library SYSEXRM.

Example 2

```
/* EXAMPLE 'ETREX2:' END TRANSACTION (WITH ET DATA)
/************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
  2 CITY
1 #PERS-NR (A8) INIT <' '>
END-DEFINE
/************************************************************
REPEAT
 INPUT 'ENTER PERSONNEL NUMBER TO BE UPDATED:' #PERS-NR
  IF #PERS-NR = ' '
   ESCAPE BOTTOM
  END-IF
/************************************************************
FIND EMPLOY-VIEW PERSONNEL-ID = #PERS-NR
 INPUT (AD=M)    NAME / FIRST-NAME / CITY
 UPDATE
 END TRANSACTION #PERS-NR
END-FIND
/************************************************************
END-REPEAT
END
```

```
ENTER PERSONNEL NUMBER TO BE UPDATED: 20027800
```

```
NAME LAWLER
  FIRST-NAME SUNNY
  CITY MILWAUKEE
```

 Copyright © Software AG 2002

ESCAPE

Structured Mode Syntax

```
                  TOP
ESCAPE {BOTTOM [(r)] [IMMEDIATE]}
                  ROUTINE [IMMEDIATE]
```

Reporting Mode Syntax

```
                  TOP
ESCAPE [BOTTOM [(r)] [IMMEDIATE]]
                  ROUTINE [IMMEDIATE]
```

Function

The ESCAPE statement is used to interrupt the linear flow of execution of a processing loop or a routine.

With the keywords TOP, BOTTOM and ROUTINE you indicate where processing is to continue when the ESCAPE statement is encountered.

An ESCAPE TOP/BOTTOM statement, when encountered for processing, will internally refer to the innermost active processing loop. The ESCAPE statement need not be physically placed within the processing loop.

If an ESCAPE TOP/BOTTOM statement is placed in a routine (subroutine, subprogram, or program invoked with FETCH RETURN), the routine(s) entered during execution of the processing loop will be terminated automatically.

ESCAPE TOP

TOP indicates that processing is to continue at the top of the processing loop. The next repetition of the processing loop is begun.

ESCAPE BOTTOM

BOTTOM indicates that processing is to continue with the first statement following the processing loop. The loop is terminated and loop-end processing (final BREAK and END DATA) is executed for all loops being terminated.

If BOTTOM is followed by a label or reference number, processing will continue with the first statement following the processing loop identified by the label or reference number.

If you specify the keyword IMMEDIATE, no loop-end processing will be performed.

In reporting mode, ESCAPE BOTTOM is the default.

ESCAPE ROUTINE

This option indicates that the current Natural routine, which may have been invoked via a PERFORM, CALLNAT, FETCH RETURN, or as a main program, is to relinquish control.

In the case of a subroutine, processing will continue with the first statement after the statement used to invoke the subroutine. In the case of a main program, Natural command mode will be entered.

All loops currently active within the routine will be terminated and loop-end processing performed as well as final processing for user-initiated (PERFORM BREAK) processing. If the program containing the ESCAPE ROUTINE is executed as a main program (level 1), final end-page processing is performed.

If you specify the keyword IMMEDIATE, no loop-end processing will be performed.

Additional Considerations

More than one ESCAPE statement may be contained within the same processing loop.

The execution of an ESCAPE statement may be based on a logical condition.

If an ESCAPE statement is encountered during processing of an AT END OF DATA, AT BREAK or AT END OF PAGE block, the execution of the special condition block will be terminated and ESCAPE processing will continue as required.

If an ESCAPE statement is encountered during processing of an if-no-records-found condition, no loop-end processing will be performed (equivalent to ESCAPE IMMEDIATE).

 Copyright © Software AG 2002

Example

```
/* EXAMPLE 'ESCEX1S': ESCAPE (STRUCTURED MODE)
/*********************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 FIRST-NAME
  2 NAME
  2 AREA-CODE
  2 PHONE
1 #CITY (A20) INIT <' '>
1 #CNTL (A1) INIT <' '>
END-DEFINE
/*********************************************************************
RPT. REPEAT
      INPUT 'ENTER VALUE FOR CITY:  ' #CITY
          / '(OR ''.'' TO TERMINATE)'
      IF #CITY = '.'
       STOP
      END-IF
/*********************************************************************
FND.  FIND EMPLOY-VIEW WITH CITY = #CITY
/*********************************************************************
       IF NO RECORDS FOUND
        WRITE 'NO RECORDS FOUND'
        ESCAPE BOTTOM (FND.)
       END-NOREC
       AT START OF DATA
        INPUT (AD=O) 'RECORDS FOUND:' *NUMBER //
                    'ENTER ''D'' TO DISPLAY RECORDS' #CNTL (AD=M)
        IF #CNTL NE 'D'
        ESCAPE BOTTOM (FND.)
        END-IF
       END-START
/*********************************************************************
       DISPLAY NOTITLE NAME FIRST-NAME PHONE
      END-FIND
     END-REPEAT
/*********************************************************************
END
```

```
ENTER VALUE FOR CITY:   PARIS
(OR '.' TO TERMINATE)
```

```
RECORDS FOUND:        24

ENTER 'D' TO DISPLAY RECORDS D
```

```
       NAME              FIRST-NAME            TELEPHONE

                                              AREA     TELEPHONE
                                              CODE
-------------------- -------------------- ------ ---------------

MAIZIERE             ELISABETH            1033   46758304
MARX                 JEAN-MARIE           1033   40738871
REIGNARD             JACQUELINE           1033   48472153
RENAUD               MICHEL               1033   46055008
REMOUE               GERMAINE             1033   36929371
LAVENDA              SALOMON              1033   40155905
BROUSSE              GUY                  1033   37502323
GIORDA               LOUIS                1033   37497316
SIECA                FRANCOIS             1033   40487413
CENSIER              BERNARD              1033   38070268
DUC                  JEAN-PAUL            1033   38065261
CAHN                 RAYMOND              1033   43723961
MAZUY                ROBERT               1033   44286899
VALLY                ALAIN                1033   47326249
BRETON               JEAN-MARIE           1033   48467146
GIGLEUX              JACQUES              1033   40477399
XOLIN                CHRISTIAN            1033   46060015
```

Equivalent reporting-mode example: See the program ESCEX1R in the library SYSEXRM.

 Copyright © Software AG 2002

EXAMINE

```
EXAMINE [FULL [VALUE [OF] ] ] { operand1                          }
                              { SUBSTRING (operand1,operand2,operand3) }
        [FOR] [FULL [VALUE [OF] ] ] [PATTERN] operand4
        [DELIMITERS-option]
        { DELETE-REPLACE-clause                    }
        { [DELETE-REPLACE-clause] GIVING-clause    }
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C*	S	A			A											yes	no
Operand2	C	S					N	P	I								yes	no
Operand3	C	S					N	P	I								yes	no
Operand4	C	S				A											yes	no

**Operand1* can only be a constant if the GIVING clause is used, but not if the DELETE/REPLACE clause is used.

Related Statements: COMPRESS | SEPARATE

Function

The EXAMINE statement is used to scan the contents of an alphanumeric field, or a range of fields within an array, for a character string; to replace and/or count the number of occurrences of that character string; and to give information about the result of the EXAMINE operation.

operand1

Operand1 is the field whose content is to be examined.

If *operand1* is a DYNAMIC variable, a REPLACE operation may cause its length to be increased or decreased; a DELETE operation may cause its length to be set to "0". The current length of a DYNAMIC variable can be ascertained by using the system variable *LENGTH. For general information on DYNAMIC variables, see your Natural User’s Guide.

operand4

Operand4 is the value to be used for the examine operation.

FULL

If FULL is specified for an operand, the entire value, including trailing blanks, will be processed. If FULL is not specified, trailing blanks in the operand will be ignored.

SUBSTRING

Normally, the content of a field is examined from the beginning of the field to the end or to the last non-blank character.

With the SUBSTRING option, you examine only a certain part of the field. After the field name (*operand1*) in the SUBSTRING clause, you specify first the starting position (*operand2*) and then the length (*operand3*) of the field portion to be examined.

For example, to examine the 5th to 12th position inclusive of a field #A, you would specify:

```
EXAMINE SUBSTRING(#A,5,8).
```

Note:
If you omit operand2, the starting position is assumed to be "1". If you omit operand3, the length is assumed to be from the starting position to the end of the field.

PATTERN

If you wish to examine the field for a value which contains variables, that is symbols for positions not to be examined, you use the PATTERN option. *Operand4* may then include the following symbols for positions to be ignored:

- A period (.), question mark (?) or underscore (_) indicates a single position that is not to be examined.
- An asterisk (*) or a percent sign (%) indicates any number of positions not to be examined.

Example: With PATTERN ’NAT*AL’ you could examine the field for 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 NATIONAL as well as NATAL).

DELIMITERS-option

```
{ ABSOLUTE                    }
{ WITH [DELIMITERS]           }
{ WITH [DELIMITERS] operand5  }
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand5	C	S				A											yes	no

 Copyright © Software AG 2002

The default option is ABSOLUTE. This results in an absolute scan of the field for the specified value regardless of what other characters may surround the value.

WITH DELIMITERS is used to scan for a value which is delimited by blanks or by any characters that are neither letters nor numeric characters.

WITH DELIMITERS *operand5* is used to scan for a value which is delimited by the character(s) specified in *operand5*.

DELETE-REPLACE-clause

```
[AND] { DELETE [FIRST]                                          }
      { REPLACE [FIRST] [WITH] [FULL [VALUE [OF] ] ] operand6 }
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand6	C	S				A											yes	no

The DELETE option is used to delete each value from *operand1*.

The REPLACE option is used to replace each value in *operand1* by the value specified in *operand6*.

If you specify the keyword FIRST, only the first identical value will be deleted/replaced.

If the REPLACE operation results in more characters being generated than will fit into *operand1*, you will receive an error message.

If *operand1* is a DYNAMIC variable, a REPLACE operation may cause its length to be increased or decreased; a DELETE operation may cause its length to be set to "0". The current length of a DYNAMIC variable can be ascertained by using the system variable *LENGTH. For general information on DYNAMIC variables, see your Natural User’s Guide.

GIVING-clause

```
[GIVING] [ NUMBER   ] [IN] operand7
         [ POSITION ]
         [ LENGTH   ]
         [ INDEX    ]
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand7		S					N	P	I								yes	yes

GIVING NUMBER is used to obtain the number of occurrences of the value sought. If the REPLACE FIRST or DELETE FIRST option is also used, the number will not exceed 1.

GIVING POSITION is used to obtain the byte position within *operand1* (or the substring of *operand1*) where the first value identical to *operand4* was found.

GIVING LENGTH is used to obtain the length of *operand1* (or the substring of *operand1*) after all delete/replace operations have been performed.

GIVING INDEX

```
[GIVING] INDEX [IN] operand7...3
```

GIVING INDEX is used to obtain the occurrence number (index) of the *operand1* occurrence in which the first value identical to *operand4* was found.

GIVING INDEX is applicable only if *operand1* is an array. *Operand7* must be specified as many times as there are dimensions contained in *operand1* (maximum three times).

Operand7 will contain "0" if the value sought is found in none of the occurrences.

Note:
If the index range of operand1 includes the occurrence 0 (e.g. 0:5), a value of "0" in operand7 is ambiguous. In this case, an additional GIVING NUMBER clause should be used to ascertain whether the value sought was actually found or not.

 Copyright © Software AG 2002

Example 1

```
/* EXAMPLE 'EXMEX1': EXAMINE
/************************************************************
DEFINE DATA LOCAL
1 #TEXT (A40)
1 #A (A1)
1 #NMB1 (N2)
1 #NMB2 (N2)
1 #NMB3 (N2)
1 #NMBEX2  (N2)
1 #NMBEX3  (N2)
1 #NMBEX4  (N2)
1 #POSEX5  (N2)
1 #LGHEX6  (N2)
END-DEFINE
/************************************************************
WRITE 'EXAMPLE 1 (GIVING NUMBER, WITH DELIMITER)'
MOVE 'ABC   A B C   .A.  .B.  .C.    -A-  -B- ' TO #TEXT
ASSIGN #A = 'A'
EXAMINE #TEXT FOR #A GIVING NUMBER #NMB1
EXAMINE #TEXT FOR #A WITH DELIMITER GIVING NUMBER #NMB2
EXAMINE #TEXT FOR #A WITH DELIMITER '.' GIVING NUMBER #NMB3
WRITE NOTITLE '=' #NMB1 '=' #NMB2 '=' #NMB3
/************************************************************
WRITE / 'EXAMPLE 2 (WITH DELIMITER, REPLACE, GIVING NUMBER)'
WRITE '=' #TEXT
EXAMINE #TEXT FOR #A WITH DELIMITER '-' REPLACE WITH '*'
                    GIVING NUMBER #NMBEX2
WRITE '=' #TEXT '=' #NMBEX2
/************************************************************
WRITE / 'EXAMPLE 3 (REPLACE, GIVING NUMBER)'
WRITE '=' #TEXT
EXAMINE #TEXT ' ' REPLACE WITH '+' GIVING NUMBER #NMBEX3
WRITE '=' #TEXT '=' #NMBEX3
/************************************************************
WRITE / 'EXAMPLE 4 (FULL, REPLACE, GIVING NUMBER)'
WRITE '=' #TEXT
EXAMINE FULL #TEXT ' ' REPLACE WITH '+' GIVING NUMBER #NMBEX4
WRITE '=' #TEXT '=' #NMBEX4
/************************************************************
WRITE / 'EXAMPLE 5 (DELETE, GIVING POSITION)'
WRITE '=' #TEXT
EXAMINE #TEXT '+' DELETE GIVING POSITION #POSEX5
WRITE '=' #TEXT '=' #POSEX5
/************************************************************
WRITE / 'EXAMPLE 6 (DELETE, GIVING LENGTH)'
WRITE '=' #TEXT
EXAMINE #TEXT FOR 'A' DELETE GIVING LENGTH #LGHEX6
WRITE '=' #TEXT '=' #LGHEX6
END
```

```
EXAMPLE 1 (GIVING NUMBER, WITH DELIMITER)
  #NMB1:   4 #NMB2:   3 #NMB3:   1

  EXAMPLE 2 (WITH DELIMITER, REPLACE, GIVING NUMBER)
  #TEXT: ABC   A B C   .A.  .B.  .C.    -A-  -B-
  #TEXT: ABC   A B C   .A.  .B.  .C.    -*-  -B-  #NMBEX2:   1

  EXAMPLE 3 (REPLACE, GIVING NUMBER)
  #TEXT: ABC   A B C   .A.  .B.  .C.    -*-  -B-
  #TEXT: ABC+++A+B+C+++.A.++.B.++.C.++++-*-++-B-  #NMBEX3:  18

  EXAMPLE 4 (FULL, REPLACE, GIVING NUMBER)
  #TEXT: ABC+++A+B+C+++.A.++.B.++.C.++++-*-++-B-
  #TEXT: ABC+++A+B+C+++.A.++.B.++.C.++++-*-++-B-+ #NMBEX4:   1

  EXAMPLE 5 (DELETE, GIVING POSITION)
  #TEXT: ABC+++A+B+C+++.A.++.B.++.C.++++-*-++-B-+
  #TEXT: ABCABC.A..B..C.-*--B-                    #POSEX5:   4

  EXAMPLE 6 (DELETE, GIVING LENGTH)
  #TEXT: ABCABC.A..B..C.-*--B-
  #TEXT: BCBC...B..C.-*--B-                       #LGHEX6:  18
```

 Copyright © Software AG 2002

Example 2

```
/* EXAMPLE 'EXMEX2': EXAMINE SUBSTRING, PATTERN, TRANSLATE
/************************************************************
DEFINE DATA LOCAL
1 #TEXT (A50)
1 #A (A7)
1 #NMB (N2)
1 #START (N2)
1 #TAB(A2/1:10)
END-DEFINE
/************************************************************
MOVE 'ABC   A B C   .A.  .B.  .C.    -A-  -B-  -C- ' TO #TEXT
/************************************************************
ASSIGN #A = 'A B C'
ASSIGN #START = 6
EXAMINE SUBSTRING(#TEXT,#START,9) FOR #A GIVING NUMBER #NMB
WRITE NOTITLE '=' #NMB
/************************************************************
EXAMINE #TEXT FOR PATTERN '*B' GIVING NUMBER #NMB
WRITE NOTITLE '=' #NMB
/************************************************************
MOVE 'AX' TO #TAB(1)
MOVE 'BY' TO #TAB(2)
MOVE 'CZ' TO #TAB(3)
EXAMINE #TEXT TRANSLATE USING #TAB(*)
WRITE NOTITLE '=' #TEXT
EXAMINE #TEXT TRANSLATE USING INVERTED #TAB(*)
WRITE NOTITLE '=' #TEXT
/************************************************************
END
```

```
#NMB:   1
  #NMB:   4
  #TEXT: XYZ   X Y Z   .X.  .Y.  .Z.    -X-  -Y-  -Z-
  #TEXT: ABC   A B C   .A.  .B.  .C.    -A-  -B-  -C-
```

EXAMINE TRANSLATE

```
EXAMINE { operand1                                  } [AND]
        { SUBSTRING (operand1,operand2,operand3)    }
        TRANSLATE { INTO { UPPER } [CASE]           }
                  { LOWER }                         }
                  { USING [INVERTED] operand4       }
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1		S	A			A											yes	no
Operand2	C	S					N	P	I								yes	no
Operand3	C	S					N	P	I								yes	no
Operand4		S	A			A					B						yes	no

Function

The EXAMINE TRANSLATE statement is used to translate the characters contained in a field into upper-case or lower-case, or into other characters.

operand1

Operand1 is the field whose content is to be translated.

SUBSTRING

Normally, the entire content of a field is translated.

With the SUBSTRING option, you translate only a certain part of the field. After the field name (*operand1*) in the SUBSTRING clause, you specify first the starting position (*operand2*) and then the length (*operand3*) of the field portion to be examined.

For example, to translate the 5th to 12th position inclusive of a field #A, you would specify:

```
EXAMINE SUBSTRING(#A,5,8) AND TRANSLATE ...
```

Note:
If you omit operand2, the starting position is assumed to be "1". If you omit operand3, the length is assumed to be from the starting position to the end of the field.

 Copyright © Software AG 2002

INTO UPPER/LOWER CASE

If you specify INTO UPPER CASE, the content of *operand1* will be translated into upper case.

If you specify INTO LOWER CASE, the content of *operand1* will be translated into lower case.

Translation Table

Operand4 is the translation table to be used for character translation.

The table must be of format/length A2 or B2.

Note:
If for a character to be translated more than one translation is defined in the translation table, the last of these translations applies.

INVERTED

If you specify the keyword INVERTED, the translation table (*operand4*) will be used inverted; that is, the translation direction will be reversed.

Example

```
/* EXAMPLE 'EXMEX2': EXAMINE SUBSTRING, PATTERN, TRANSLATE
/************************************************************
DEFINE DATA LOCAL
1 #TEXT (A50)
1 #A (A7)
1 #NMB (N2)
1 #START (N2)
1 #TAB(A2/1:10)
END-DEFINE
/************************************************************
MOVE 'ABC   A B C   .A.  .B.  .C.    -A-  -B-  -C- ' TO #TEXT
/************************************************************
ASSIGN #A = 'A B C'
ASSIGN #START = 6
EXAMINE SUBSTRING(#TEXT,#START,9) FOR #A GIVING NUMBER #NMB
WRITE NOTITLE '=' #NMB
/************************************************************
EXAMINE #TEXT FOR PATTERN '*B' GIVING NUMBER #NMB
WRITE NOTITLE '=' #NMB
/************************************************************
MOVE 'AX' TO #TAB(1)
MOVE 'BY' TO #TAB(2)
MOVE 'CZ' TO #TAB(3)
EXAMINE #TEXT TRANSLATE USING #TAB(*)
WRITE NOTITLE '=' #TEXT
EXAMINE #TEXT TRANSLATE USING INVERTED #TAB(*)
WRITE NOTITLE '=' #TEXT
/************************************************************
END
```

```
#NMB:   1
  #NMB:   4
  #TEXT: XYZ   X Y Z   .X.  .Y.  .Z.    -X-  -Y-  -Z-
  #TEXT: ABC   A B C   .A.  .B.  .C.    -A-  -B-  -C-
```

 Copyright © Software AG 2002

EXPAND

Note:
This statement is not available on mainframe computers.

EXPAND [SIZE OF] DYNAMIC [VARIABLE] *operand1* **TO** *operand2*

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1		S				A	B										no	no
Operand2	C	S							I								no	no

Related statements: REDUCE

Function

The EXPAND DYNAMIC VARIABLE statement expands the allocated size of a dynamic variable (operand1) to the value specified with operand2. If *operand2* is less than the currently allocated size of *operand1*, the statement will be ignored for this dynamic variable. The currently used size (*LENGTH) of the dynamic variable is not modified.

operand1

Operand1 is the dynamic variable for which the size is to be expanded.

operand2

Operand2 is used to specify the new size. The value specified must be a non-negative numeric value.

Copyright © Software AG 2002

FETCH

```
FETCH [REPEAT] operand1 [operand2 [(parameter)] ]...
      [RETURN]
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S				A												yes	no
Operand2	C	S	A	G		A	N	P	I	F	B	D	T	L		G		yes	yes

Related Statements: CALLNAT | PERFORM

Function

The FETCH statement is used to execute a Natural object program written as a main program. The program to be loaded must have been previously stored in the Natural system file with a CATALOG or STOW command. Execution of the FETCH statement does not overwrite any source program in the Natural source work area.

REPEAT

REPEAT causes Natural to suppress the prompt for user input for each INPUT statement issued during the execution of the FETCHed program. It may be used to send information about the execution of the program to the terminal without the user having to reply with ENTER.

RETURN

Without the specification of RETURN, the execution of the program issuing the FETCH statement will be terminated immediately and the FETCHed program will be activated as a *main program* (level 1).

If a program is invoked with FETCH RETURN, the execution of the invoking program will be suspended - not terminated - and the FETCHed program will be activated as a *subordinate program* on a higher level. Control is returned to the invoking program when an END or ESCAPE ROUTINE statement is encountered in the FETCHed program. Processing is continued with the statement following the FETCH RETURN statement.

With FETCH RETURN, you invoke and execute an object of type program as a routine.

Program Name - operand1

The name of the program module (maximum 8 characters) can be specified as an alphanumeric constant or the content of an alphanumeric variable of length 1 to 8.

Natural will attempt to locate the program in the library currently active at the time the FETCH is issued. If the program is not found, Natural will attempt to locate the program in the steplibs. If the program is still not found, an error message will be issued.

 Copyright © Software AG 2002

The program name may contain an ampersand (&); at execution time, this character will be replaced by the current value of the system variable *LANGUAGE. This makes it possible, for example, to invoke different programs for the processing of input, depending on the language in which input is provided.

Parameters - operand2

The FETCH statement may also be used to pass parameter fields to the invoked program. A parameter field may be defined with any format. The parameters are converted to a format suitable for a corresponding INPUT field. All parameters are placed on the top of the Natural stack.

The parameter fields can be read by the FETCHed program using an INPUT statement. The first INPUT statement will result in the insertion of all parameter field values into the fields specified in the INPUT statement. The INPUT statement must have the sign specification (SG=ON) for parameter fields defined with numeric format, because each parameter field defined with numeric format in the FETCH statement will receive a sign position if its value is negative.

If more parameters are passed than are read by the next INPUT statement, the extra parameters are ignored. The number of parameters may be obtained with the Natural system variable *DATA.

Note:
If operand2 is a time variable (format T), only the time component of the variable content is passed, but not the date component.

parameter

If *operand2* is a date variable, you can specify the session parameter DF as *parameter* for this variable. The session parameter DF is described in the Natural Reference documentation.

Additional Considerations

In addition to the parameters passed explicitly with FETCH, the FETCHed program also has access to the established global data area.

The FETCH statement may cause the internal execution of an END TRANSACTION statement based on the setting of the Natural profile parameter OPRB as set by the Natural administrator. If a logical transaction is to span multiple Natural programs, the Natural administrator should be consulted to ensure that the OPRB parameter is set correctly.

Copyright © Software AG 2002

Example

Invoking Program:

```
/* EXAMPLE 'FETEX1': FETCH
/*********************************************************
DEFINE DATA LOCAL
1 #PNUM (A8)
1 #FNC  (A1)
END-DEFINE
/*********************************************************
INPUT 10X 'SELECTION MENU FOR EMPLOYEES SYSTEM' /
      10X '-' (54) //
      10X 'ADD  '     '(A)' /
      10X 'PURGE'     '(P)' /
      10X 'UPDATE'    '(U)' /
      10X 'TERMINATE' '(.)' //
      10X 'PERSONNEL NUMBER:' #PNUM ///
      10X 'PLEASE ENTER FUNCTION: ' #FNC
/*********************************************************
DECIDE ON EVERY VALUE OF #FNC
  VALUE 'A'
    FETCH 'ADD-RT' #PNUM
  VALUE 'P'
    FETCH 'PUR-RT' #PNUM
  VALUE 'U'
    FETCH 'UPD-RT' #PNUM
  VALUE '.'
    STOP
  NONE
    REINPUT 'PLEASE ENTER A VALID FUNCTION' MARK *#FNC
END-DECIDE
/*********************************************************
END
```

 Copyright © Software AG 2002

Invoked Program:

```
/* EXAMPLE 'PUR-RT' (PROGRAM FETCHED IN EXAMPLE 'FETEX1')
/*******************************************************
DEFINE DATA LOCAL
1 #PERS-NR (A8)
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
END-DEFINE
/*******************************************************
INPUT #PERS-NR
/*******************************************************
FIND NUMBER EMPLOY-VIEW WITH PERSONNEL-ID = #PERS-NR
 IF *NUMBER = 0
  WRITE NOTITLE 'NO RECORD FOUND'
  STOP
 END-IF
/*******************************************************
 FIND EMPLOY-VIEW WITH PERSONNEL-ID = #PERS-NR
  DELETE
  END TRANSACTION
  WRITE NOTITLE 'RECORD DELETED'
 END-FIND
/*******************************************************
END
```

```
            SELECTION MENU FOR EMPLOYEES SYSTEM
            -------------------------------------------------------

            ADD   (A)
            PURGE (P)
            UPDATE (U)
            TERMINATE (.)

            PERSONNEL NUMBER: 1150304


            PLEASE ENTER FUNCTION:  P
```

```
RECORD DELETED
```

FIND

```
        [   ALL      ]
        [ (operand1) ]
FIND    [   FIRST    ]  [RECORDS] [IN] [FILE] view-name
        [   NUMBER   ]
        [   UNIQUE   ]
        [PASSWORD = operand2]
        [CIPHER = operand3]
        [WITH] [ [LIMIT] (operand4)] basic-search-criterion
        [COUPLED-clause]...4/42
        [STARTING WITH ISN = operand5]
        [SORTED-BY-clause]
        [RETAIN-clause]
        [WHERE-clause]
        [IF-NO-RECORDS-FOUND-clause]
        statement...
END-FIND    (structured mode only)
[LOOP]      (reporting mode only)
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S					N	P	I								yes	no
Operand2	C	S				A											yes	no
Operand3	C	S					N										yes	no
Operand4	C	S					N	P	I		B						yes	no
Operand5	C	S					N	P	I		B						yes	no

Related Statements: READ | HISTOGRAM

Function

The FIND statement is used to select a set of records from the database based on a search criterion consisting of fields defined as descriptors (keys).

This statement causes a processing loop to be initiated and then executed for each record selected. Each field in each record may be referenced within the processing loop. It is not necessary to issue a READ statement following the FIND in order to reference the fields within each record selected.

 Copyright © Software AG 2002

Considerations for DL/I Databases

When accessing a field starting after the last byte of the given segment occurrence, the storage copy of this field is filled according to its format (numeric, blank, etc.). The term segment occurrences should be substituted for the term records as used in this description of the FIND statement.

Considerations for SQL Databases

FIND FIRST as well as the PASSWORD, CIPHER, COUPLED and RETAIN clauses are not permitted.

FIND UNIQUE is not permitted. (Exception: On mainframe computers, FIND UNIQUE can be used for primary keys; however, this is only permitted for compatibility reasons and should not be used.)

The SORTED BY clause corresponds with the SQL clause ORDER BY.

The basic search criterion for an SQL-database table may be specified in the same manner as for an Adabas file. The term record used in this context corresponds with the SQL term `row`.

Considerations for VSAM Databases

The FIND statement is only valid for key-sequenced (KSDS) and entry-sequenced (ESDS) VSAM datasets. For ESDS, an alternate index for the base cluster must be defined.

Entire System Server Restrictions

FIND NUMBER and FIND UNIQUE as well as the PASSWORD, CIPHER, COUPLED and RETAIN clauses are not permitted.

Processing Limit - ALL/operand1

The number of records to be processed from the selected set may be limited by specifying *operand1* either as a numeric constant or as the name of a numeric variable enclosed in parentheses. ALL may be optionally specified and emphasizes that all selected records are to be processed.

If you specify a limit with *operand1*, this limit applies to the FIND loop being initiated. Records rejected for processing by the WHERE clause are not counted against this limit.

```
FIND (5) IN EMPLOYEES WITH ...

MOVE 10 TO #CNT(N2)
FIND (#CNT) EMPLOYEES WITH ...
```

For this statement, the specified limit has priority over a limit set with a LIMIT statement.

If a smaller limit is set with the LT parameter, the LT limit applies.

Notes:
If you wish to process a 4-digit number of records, specify it with a leading zero: (0nnnn); because Natural interprets every 4-digit number enclosed in parentheses as a line-number reference to a statement.
Operand1 has no influence on the size of an ISN set that is to be retained by a RETAIN clause.
Operand1 is evaluated when the FIND loop is entered. If the value of operand1 is modified within the FIND loop, this does not affect the number of records processed.

FIND FIRST, FIND NUMBER, FIND UNIQUE

These options are used to select the first record of a selected set (FIND FIRST), to determine the number of records in a selected set (FIND NUMBER), or to ensure that only one record satisfies a selection criterion (FIND UNIQUE).

These options are described in detail at the end of the FIND statement description.

view-name

The name of a view as defined either within a DEFINE DATA block or in a separate global or local data. In reporting mode, *view-name* may also be the name of a DDM.

PASSWORD Clause

The PASSWORD clause applies only for Adabas or VSAM databases. This clause is not permitted with Entire System Server.

The PASSWORD clause is used to provide a password (operand2) when retrieving data from an Adabas file which is password protected. If you require access to a password-protected file, contact the person responsible for database security concerning password usage/assignment.

If the password is specified as a constant, the PASSWORD clause should always be coded at the very beginning of a source-code line; this ensures that the password is not visible/displayable in the source code of the program. In TP mode, you may enter the PASSWORD clause invisible by entering the terminal command "%*" before you type in the PASSWORD clause.

If the PASSWORD clause is omitted, the password specified with the PASSW statement applies.

The password value must not be changed during the execution of a processing loop.

Example of PASSWORD Clause:

```
/* EXAMPLE 'FNDPWD': FIND (USING PASSWORD CLAUSE)
/************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 PERSONNEL-ID
1 #PASS (A8)
END-DEFINE
/************************************************************
INPUT 'ENTER PASSWORD FOR EMPLOYEE FILE:' #PASS (AD=N)
LIMIT 2
/************************************************************
FIND EMPLOY-VIEW
     PASSWORD = #PASS
     WITH NAME = 'SMITH'
 DISPLAY NOTITLE NAME PERSONNEL-ID
END-FIND
/************************************************************
END
```

 Copyright © Software AG 2002

```
ENTER PASSWORD FOR EMPLOYEE FILE:
```

CIPHER Clause

The CIPHER clause only applies to Adabas databases. This clause is not permitted with Entire System Server.

The CIPHER clause is used to provide a cipher key (operand3) when retrieving data from Adabas files which are enciphered. If you require access to an enciphered file, contact the person responsible for database security concerning cipher key usage/assignment.

The cipher key may be specified as a numeric constant (8 digits) or the content of a user-defined variable with format/length N8.

If the cipher key is specified as a constant, the CIPHER clause should always be coded at the very beginning of a source-code line; this ensures that the cipher key is not visible/displayable in the source code of the program. In TP mode, you may enter the CIPHER clause invisible by entering the Natural terminal command "%*" before you type in the CIPHER clause.

The value of the cipher key must not be changed during the processing of a loop initiated by a FIND statement.

Example of CIPHER Clause:

```
/* EXAMPLE 'FNDCIP': FIND (USING CIPHER CLAUSE)
/*****************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 PERSONNEL-ID
1 #PASS (A8)
1 #CIPHER (N8)
END-DEFINE
/*****************************************************************
LIMIT 2
INPUT 'ENTER PASSWORD FOR EMPLOYEE FILE:   ' #PASS (AD=N)
    / 'ENTER CIPHER KEY FOR EMPLOYEE FILE: ' #CIPHER (AD=N)
/*****************************************************************
FIND EMPLOY-VIEW
     PASSWORD = #PASS
     CIPHER   = #CIPHER
     WITH NAME = 'SMITH'
 DISPLAY NOTITLE NAME PERSONNEL-ID
END-FIND
/*****************************************************************
END
```

```
ENTER PASSWORD FOR EMPLOYEE FILE:
ENTER CIPHER KEY FOR EMPLOYEE FILE:
```

WITH Clause

The WITH clause is required. It is used to specify the *basic-search-criterion* consisting of key fields (descriptors) defined in the database.

For Adabas files, you may use Adabas descriptors, subdescriptors, superdescriptors, hyperdescriptors, and phonetic descriptors within a WITH clause. On mainframe computers, a non-descriptor (that is, a field marked in the DDM with "N") can also be specified.

For DL/I files, you may only use key fields marked with "D" in the DDM.

For VSAM files, you may use VSAM key fields only.

The number of records to be selected as a result of a WITH clause may be limited by specifying the keyword LIMIT together with a numeric constant or a user-defined variable, enclosed within parentheses, which contains the limit value (*operand4*). If the number of records selected exceeds the limit, the program will be terminated with an error message.

Note:
If the limit is to be a 4-digit number, specify it with a leading zero: (0nnnn); because Natural interprets every 4-digit number enclosed in parentheses as a line-number reference to a statement.

Search Criterion for Adabas Files - basic-search-criterion

```
1   descriptor [(i)] { = | EQ | EQUAL [TO] } value
                 { [OR { = | EQ | EQUAL [TO] } value]... | THRU value [BUT NOT value [THRU value]] }

2   descriptor [(i)] { = | EQ | EQUAL [TO] | ¬ = | NE | NOT = | NOT EQ | NOT EQUAL | NOT EQUAL [TO] | < | LT | LESS THAN | <= | LE | LESS EQUAL | > | GT | GREATER THAN | >= | GE | GREATER EQUAL } value

3   set-name
```

 Copyright © Software AG 2002

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Descriptor		S	A			A	N	P	I	F	B	D	T	L				no	no
Value	C	S				A	N	P	I	F	B	D	T	L				yes	no
Set-name	C	S				A												no	no

descriptor

Adabas descriptor, subdescriptor, superdescriptor, hyperdescriptor, or phonetic descriptor. A field marked as non-descriptor in the DDM can also be specified.

i

A descriptor contained within a periodic group may be specified with or without an index. If no index is specified, the record will be selected if the value specified is located in any occurrence. If an index is specified, the record is selected only if the value is located in the occurrence specified by the index. The index specified must be a constant. An index range must not be used.

No index must be specified for a descriptor which is a multiple-value field. The record will be selected if the value is located in the record regardless of the position of the value.

value

Search value. The formats of the descriptor and the search value must be compatible.

set-name

Identifies a set of records previously selected with a FIND statement in which the RETAIN clause was specified. The set referenced in a FIND must have been created from the same physical Adabas file. *set-name* may be specified as a text constant (maximum 32 characters) or as the content of an alphanumeric variable. *set-name* cannot be used with Entire System Server.

Examples of Basic Search Criterion in WITH Clause:

```
FIND STAFF WITH NAME = 'SMITH'
FIND STAFF WITH CITY NE 'BOSTON'
FIND STAFF WITH BIRTH = 610803
FIND STAFF WITH BIRTH = 610803 THRU 610811
FIND STAFF WITH NAME = 'O HARA' OR = 'JONES' OR = 'JACKSON'
FIND STAFF WITH PERSONNEL-ID = 100082 THRU 100100
                                 BUT NOT 100087 THRU 100095
```

Examples of Basic Search Criterion with Multiple-Value Field:

When the descriptor used in the basic search criterion is a multiple-value field, basically four different kinds of results can be obtained (the field MU-FIELD in the following examples is assumed to be a multiple-value field):

1. FIND XYZ-VIEW WITH **MU-FIELD = 'A'**
   This statement returns records in which *at least one* occurrence of MU-FIELD has the value "A".
2. FIND XYZ-VIEW WITH **MU-FIELD NOT EQUAL 'A'**
   This statement returns records in which *at least one* occurrence of MU-FIELD does *not* have the value "A".
3. FIND XYZ-VIEW WITH **NOT MU-FIELD NOT EQUAL 'A'**
   This statement returns records in which *every* occurrence of MU-FIELD has the value "A".
4. FIND XYZ-VIEW WITH **NOT MU-FIELD = 'A'**
   This statement returns records in which *none* of the occurrences of MU-FIELD has the value "A".

Search Criterion with Null Indicator - basic-search-criterion

```
null-indicator { =        } value
               { EQ       }
               { EQUAL [TO] }
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Null-indicator		S							I									no	no
Value	C	S					N	P	I	F	B							yes	no

Possible *value* is "-1" (= the corresponding field contains no value) or "0" (= the corresponding field does contain a value).

Connecting Search Criteria (for Adabas Files)

Basic-search-criteria can be combined using the Boolean operators AND, OR, and NOT. Parentheses may also be used to control the order of evaluation. The order of evaluation is as follows:

1. **( )** Parentheses
2. **NOT** Negation (only for a *basic-search-criterion* of form [2]).
3. **AND** AND connection
4. **OR** OR connection

Basic-search-criteria may be connected by logical operators to form a complex *search-expression*. The syntax for such a complex *search-expression* is as follows:

```
[NOT] { basic-search-criterion } [ { OR  } search-expression ]...
      { (search-expression)    }   { AND }
```

 Copyright © Software AG 2002

Examples of Complex Search Expression in WITH Clause:

```
FIND STAFF WITH BIRTH LT 19770101 AND DEPT = 'DEPT06'
```

```
FIND STAFF WITH JOB-TITLE = 'CLERK TYPIST'
                    AND (BIRTH GT 19560101 OR LANG = 'SPANISH')
```

```
FIND STAFF WITH JOB-TITLE = 'CLERK TYPIST'
                    AND NOT (BIRTH GT 19560101 OR LANG = 'SPANISH')
```

```
FIND STAFF WITH DEPT = 'ABC' THRU 'DEF'
                    AND CITY = 'WASHINGTON' OR = 'LOS ANGELES'
                    AND BIRTH GT 19360101
```

```
FIND CARS WITH MAKE = 'VOLKSWAGEN'
                   AND COLOR = 'RED' OR = 'BLUE' OR = 'BLACK'
```

Descriptor - Key - Usage

Adabas users may use database fields which are defined as descriptors to construct basic search criteria.

Subdescriptors, Superdescriptors, Hyperdescriptors and Phonetic Descriptors

With Adabas, subdescriptors, superdescriptors, hyperdescriptors and phonetic descriptors may be used to construct search criteria.

- A subdescriptor is a descriptor formed from a portion of a field.
- A superdescriptor is a descriptor whose value is formed from one or more fields or portions of fields.
- A hyperdescriptor is a descriptor which is formed using a user-defined algorithm.
- A phonetic descriptor is a descriptor which allows the user to perform a phonetic search on a field (for example, a person's name). A phonetic search results in the return of all values which sound similar to the search value.

Which fields may be used as descriptors, subdescriptors, superdescriptors, hyperdescriptors and phonetic descriptors with which file is defined in the corresponding DDM.

Values for Subdescriptors, Superdescriptors, Phonetic Descriptors

Values used with these types of descriptors must be compatible with the internal format of the descriptor. The internal format of a subdescriptor is the same as the format of the field from which the subdescriptor is derived. The internal format of a superdescriptor is binary if all of the fields from which it is derived are defined with numeric format; otherwise, the format is alphanumeric. Phonetic descriptors always have alphanumeric format.

Values for subdescriptors and superdescriptors may be specified in the following ways:

- Numeric or hexadecimal constants may be specified. A hexadecimal constant must be used for a value for a superdescriptor which has binary format (see above).
- Values in user-defined variable fields may be specified using the REDEFINE statement to select the portions that form the subdescriptor or superdescriptor value.

Using Descriptors Contained within a Database Array

A descriptor which is contained within a database array may also be used in the construction of basic search criterion. For Adabas databases, such a descriptor may be a multiple-value field or a field contained within a periodic group.

A descriptor contained within a periodic group may be specified with or without an index. If no index is specified, the record will be selected if the value specified is located in any occurrence. If an index is specified, the record is selected only if the value is located in the occurrence specified by the index. The index specified must be a constant. An index range must not be used.

No index must be specified for a descriptor which is a multiple-value field. The record will be selected if the value is located in the record regardless of the position of the value.

Examples using Database Arrays:

The following examples assume that the field SALARY is a descriptor contained within a periodic group, and the field LANG is a multiple-value field.

```
1. FIND EMPLOYEES WITH SALARY LT 20000
```

(results in a search of all occurrences of SALARY)

```
2. FIND EMPLOYEES WITH SALARY (1) LT 20000
```

(results in a search of the first occurrence only)

```
3. FIND EMPLOYEES WITH SALARY (1:4) LT 20000    /* invalid
```

(a range specification must not be specified for a field within a periodic group used as a search criterion)

```
4. FIND EMPLOYEES WITH LANG = 'FRENCH'
```

(results in a search of all values of LANG)

```
5. FIND EMPLOYEES WITH LANG (1) = 'FRENCH'        /* invalid
```

(an index must not be specified for a multiple-value field used as a search criterion)

 Copyright © Software AG 2002

Search Criterion for VSAM Files - basic-search-criterion

```
                  =
                 EQ
            EQUAL [TO]
                 ¬ =
                 NE
               NOT =
             NOT EQ
          NOT EQUAL
     NOT EQUAL [TO]
                  <
1   descriptor   LT          value
            LESS THAN
                 <=
                 LE
          LESS EQUAL
                  >
                 GT
       GREATER THAN
                 >=
                 GE
   GREATER EQUAL

                  =
                 EQ
2   descriptor   EQUAL [TO]  value  THRU  value
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Descriptor		S	A			A	N	P			B							no	no
Value	C	S				A	N	P			B							yes	no

descriptor

The descriptor must be defined in a VSAM file as a VSAM key field and is marked in the DDM with "P" for primary key or "A" for alternate key.

value

The search value.

The formats of the *descriptor* and the search *value* must be compatible.

Search Criterion for DL/I Files - basic-search-criterion

```
                  =
                 EQ
            EQUAL [TO]
                ¬ =
                 NE
              NOT =
            NOT EQ
          NOT EQUAL
     NOT EQUAL [TO]
                 <
                LT
          LESS THAN
                <=
                LE
         LESS EQUAL
                 >
                GT
       GREATER THAN
                >=
                GE
    GREATER EQUAL
1   descriptor  {                 } value

                  =
                 EQ
            EQUAL [TO]
2   descriptor  {                 } value THRU value [BUT NOT value]
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Descriptor		S	A			A	N	P			B							no	no
Value	C	S				A	N	P			B							yes	no

descriptor

The descriptor must be a field defined in DL/I and is marked in the DDM with "D".

value

The search value.

For HDAM databases, only the following *basic-search-criterion* is possible:

```
                  =
                 EQ
            EQUAL [TO]
descriptor  {                 } value
```

 Copyright © Software AG 2002

Connecting Search Criteria - for DL/I Files

```
[NOT] { basic-search-criterion }   [ { OR } search-expression ]...
      { (search-expression)    }   [ { AND }                  ]
```

basic-search-criteria that refer to different segment types must not be connected with the "OR" logical operator.

Examples:

```
FIND COURSE WITH COURSEN > 1
FIND COURSE WITH COURSEN > 1 AND COURSEN < 100
FIND OFFERING WITH (COURSEN-COURSE > 1 OR TITLE-COURSE = 'Natural')
                   AND LOCATION = 'DARMSTADT'
```

Invalid example:

```
FIND OFFERING WITH COURSEN-COURSE > 1 OR LOCATION = 'DARMSTADT'
```

COUPLED-clause

This clause only applies to Adabas databases. This clause is not permitted with Entire System Server.

```
{AND}
{OR } COUPLED [TO] [FILE] view-name

                [VIA descriptor1 {   =   } descriptor2]
                                 {  EQ   }
                                 {EQUAL [TO]}

                [WITH] basic-search-criteria
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Descriptor1		S	A			A	N	P			B							no	no
Descriptor2		S	A			A	N	P			B							no	no

Note:
Without the VIA clause, the COUPLED clause may be specified up to 4 times; with the VIA clause, it may be specified up to 42 times.

The COUPLED clause is used to specify a search which involves the use of the Adabas coupling facility. This facility permits database descriptors from different files to be specified in the search criterion of a single FIND statement.

The same Adabas file must not be used in two different FIND COUPLED clauses within the same FIND statement.

A *set-name* (see RETAIN-clause) must not be specified in the *basic-search-criteria*.

Database fields in a file specified within the COUPLED clause are not available for subsequent reference in the program unless another FIND or READ statement is issued separately against the coupled file.

Note:
If the COUPLED clause is used, the main WITH clause may be omitted. If the main WITH clause is omitted, the keywords AND/OR of the COUPLED clause must not be specified.

Physical Coupling without VIA clause

The files used in a COUPLED clause without VIA must be physically coupled using the appropriate Adabas utility (as described in the Adabas documentation).

Example using Physically Coupled Files:

```
/* EXAMPLE 'FNDCPL': FIND (USING COUPLED FILES)
/*************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
1 VEHIC-VIEW VIEW OF VEHICLES
  2 MAKE
END-DEFINE
/*************************************************************
FIND EMPLOY-VIEW WITH CITY = 'FRANKFURT'
                 AND COUPLED TO VEHIC-VIEW WITH MAKE = 'VW'
 DISPLAY NOTITLE NAME
END-FIND
/*************************************************************
END
```

The reference to NAME in the DISPLAY statement of the above example is valid since this field is contained in the EMPLOYEES file, whereas a reference to MAKE would be invalid since MAKE is contained in the VEHICLES file, which was specified in the COUPLED clause.

In this example, records will be found only if EMPLOYEES and VEHICLES have been physically coupled.

Logical Coupling - VIA clause

The option "VIA *descriptor1 = descriptor2*" allows you to logically couple multiple Adabas files in a search query. *Descriptor1* is a field from the first view, and *descriptor2* is a field from the second view. The two files need not be physically coupled in Adabas. This COUPLED option uses the soft-coupling feature of Adabas Version 5, as described in the Adabas documentation.

Example using VIA Clause:

 Copyright © Software AG 2002

```
/* EXAMPLE 'FNSEX1': FIND (USING SOFT COUPLING)
/************************************************************
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
END-DEFINE
/************************************************************
FIND EMPLOY-VIEW WITH NAME = 'ADKINSON'
  AND COUPLED TO VEHIC-VIEW
    VIA PERSONNEL-ID = PERSONNEL-ID
    WITH MAKE = 'VOLVO'
  DISPLAY PERSONNEL-ID NAME FIRST-NAME
END-FIND
/************************************************************
END
```

```
Page      1                                                91-06-18  14:30:38

  PERSONNEL         NAME              FIRST-NAME
     ID
  --------- -------------------- --------------------

  20011000  ADKINSON             BOB
```

STARTING WITH ISN=operand5

This clause applies only to Adabas and VSAM databases; for VSAM, it is only valid for ESDS.

You can use this clause to specify as *operand5* an Adabas ISN (internal sequence number) or VSAM RBA (relative byte address) respectively, which is to be used as a start value for the selection of records.

This clause may be used for repositioning within a FIND loop whose processing has been interrupted, to easily determine the next record with which processing is to continue. This is particularly useful if the next record cannot be identified uniquely by any of its descriptor values. It can also be useful in a distributed client/server application where the reading of the records is performed by a server program while further processing of the records is performed by a client program, and the records are not processed all in one go, but in batches.

Note:
The start value actually used will not be the value of operand5, but the next higher value.

Example:

See the program FNDSISN in the library SYSEXRM.

SORTED BY-clause

This clause only applies to Adabas and SQL databases.
This clause is not permitted with Entire System Server.

```
SORTED [BY] descriptor...3 [DESCENDING]
```

The SORTED BY clause is used to cause Adabas to sort the selected records based on the sequence of one to three descriptors. The descriptors used for controlling the sort sequence may be different from those used for selection.

By default, the records are sorted in *ascending* sequence of values; if you want them to be in descending sequence, specify the keyword DESCENDING. The sort is performed using the Adabas inverted lists and does not result in any records being read.

Note:
The use of this clause may result in significant overhead if any descriptor used to control the sort sequence contains a large number of values. This is because the entire value list may have to be scanned until all selected records have been located in the list. When a large number of records is to be sorted, you should use the SORT statement.

Adabas sort limits (see the ADARUN LS parameter in the Adabas documentation) are in effect when the SORTED BY clause is used.

A descriptor which is contained in a periodic group must not be specified in the SORTED BY clause. A multiple-value field (without an index) may be specified.

Except on OpenVMS and mainframe computers, non-descriptors may also be specified in the SORTED BY clause.

If the SORTED BY clause is used, the RETAIN clause must not be used.

Example of SORTED BY Clause:

```
/* EXAMPLE 'FNDSOR': FIND (SORTED BY CLAUSE)
/*****************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
  2 PERSONNEL-ID
END-DEFINE
/*****************************************************
LIMIT 10
FIND EMPLOY-VIEW WITH CITY = 'FRANKFURT'
                 SORTED BY NAME PERSONNEL-ID
 DISPLAY NOTITLE NAME (IS=ON) FIRST-NAME PERSONNEL-ID
END-FIND
/*****************************************************
END
```

 Copyright © Software AG 2002

```
       NAME                FIRST-NAME      PERSONNEL
                                               ID
-------------------- -------------------- ---------

BAECKER              JOHANNES               11500345
BECKER               HERMANN                11100311
BERGMANN             HANS                   11100301
BLAU                 SARAH                  11100305
BLOEMER              JOHANNES               11200312
DIEDRICHS            HUBERT                 11600301
DOLLINGER            MARGA                  11500322
FALTER               CLAUDIA                11300311
                     HEIDE                  11400311
FREI                 REINHILD               11500301
```

RETAIN-clause

This clause only applies to Adabas databases. This clause is not permitted with Entire System Server.

RETAIN AS *operand6*

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand6	C	S				A											yes	no

By using the RETAIN clause, the result of an extensive search in large files can be retained for further processing. The selection is retained as an "ISN-set" in the Adabas work file. The set may be used in subsequent FIND statements as a basic search criterion for further refinement of the set or for further processing of the records. The set created is file-specific and may only be used in another FIND statement that processes the same file. The set may be referenced by any Natural program.

Set Name - operand6

The set name is used to identify the record set. It may be specified as an alphanumeric constant or as the content of an alphanumeric user-defined variable. Duplicate set names are not checked; consequently, if a duplicate set name is specified, the new set replaces the old set.

Releasing Sets

There is no specific limit for the number of sets that can be retained or the number of ISNs in a set. It is recommended that the minimum number of ISN sets needed at one time be defined. Sets that are no longer needed should be released using the RELEASE SETS statement.

If they are not released with a RELEASE statement, retained sets exist until the end of the Natural session, or until a logon to another library, when they are released automatically. A set created by one program may be referenced by another program for processing or further refinement using additional search criteria.

Updates by Other Users

The records identified by the ISNs in a retained set are not locked against access and/or update by other users. Before you process records from the set, it is therefore useful to check whether the original search criteria which were used to create the set are still valid: This check is done with another FIND statement, using the set name in the WITH clause as basic search criterion and specifying in a WHERE clause the original search criterion (that is, the basic search criteria as specified in the WITH clause of the FIND statement which was used to create the set).

Restriction

If the RETAIN clause is used, the SORTED BY clause must not be used.

Example of a RETAIN Clause:

```
 * EXAMPLE '': FIND (RETAIN CLAUSE) AND RELEASE
****************************************************
DEFINE DATA LOCAL
 1 EMPLOY-VIEW VIEW OF EMPLOYEES
   2 CITY
   2 BIRTH
   2 NAME
 1 #BIRTH (D)
END-DEFINE
*
MOVE EDITED '19400101' TO #BIRTH (EM=YYYYMMDD)
*
FIND NUMBER EMPLOY-VIEW WITH BIRTH GT #BIRTH
                        RETAIN AS 'AGESET1'
IF *NUMBER = 0
  STOP
END-IF
*
FIND EMPLOY-VIEW WITH 'AGESET1' AND CITY = 'NEW YORK'
  DISPLAY NOTITLE NAME CITY BIRTH (EM=YYYY-MM-DD)
END-FIND
*
RELEASE SET 'AGESET1'
END
```

```
         NAME                 CITY            DATE
                                               OF
                                             BIRTH
-------------------- -------------------- ----------

RUBIN                NEW YORK             1945-10-27
WALLACE              NEW YORK             1945-08-04
```

WHERE-clause

WHERE *logical-condition*

 Copyright © Software AG 2002

The WHERE clause is used to 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 (including AT BREAK evaluation).

The syntax for a *logical-condition* is described in the section Logical Condition Criteria in the Natural Reference documentation.

If a processing limit is specified in a FIND statement containing a WHERE clause, records which are rejected as a result of the WHERE clause are *not* counted against the limit. These records are, however, counted against any global limit specified in a Natural session parameter, the GLOBALS command, or LIMIT statement.

Example of WHERE Clause:

```
/* EXAMPLE 'FNDWHE': FIND (WHERE CLAUSE)
/*******************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 CITY
END-DEFINE
/********************************************************
FIND EMPLOY-VIEW WITH CITY = 'PARIS'
                 WHERE JOB-TITLE = 'INGENIEUR COMMERCIAL'
 DISPLAY NOTITLE CITY JOB-TITLE PERSONNEL-ID NAME
END-FIND
/********************************************************
END
```

```
      CITY                  CURRENT          PERSONNEL         NAME
                             POSITION             ID
-------------------- ------------------------- --------- --------------------

PARIS                INGENIEUR COMMERCIAL      50007300  CAHN
PARIS                INGENIEUR COMMERCIAL      50006500  MAZUY
PARIS                INGENIEUR COMMERCIAL      50004400  VALLY
PARIS                INGENIEUR COMMERCIAL      50002800  BRETON
PARIS                INGENIEUR COMMERCIAL      50001000  GIGLEUX
```

IF NO RECORDS FOUND-clause

Structured Mode Syntax

```
IF NO [RECORDS] [FOUND]
      { ENTER     }
      { statement...}
END-NOREC
```

Reporting Mode Syntax

```
IF NO [RECORDS] [FOUND]
      { ENTER                 }
      {   statement           }
      { DO statement...DOEND  }
```

The IF NO RECORDS FOUND clause may be used to cause a processing loop initiated with a FIND statement to be entered in the event that no records meet the selection criteria specified in the WITH and WHERE clauses.

If no records meet the specified WITH and WHERE criteria, the IF NO RECORDS FOUND clause causes the FIND processing loop to be executed once with an "empty" record. If this is not desired, specify the statement ESCAPE BOTTOM within the IF NO RECORDS FOUND clause.

If one or more statements are specified with the IF NO RECORDS FOUND clause, the statements will be executed immediately before the processing loop is entered. If no statements are to be executed before entering the loop, the keyword ENTER must be used.

Database Values

Unless other value assignments are made in the statements accompanying an IF NO RECORDS FOUND clause, Natural will reset to empty all database fields which reference the file specified in the current loop.

Evaluation of System Functions

Natural system functions are evaluated once for the empty record that is created for processing as a result of the IF NO RECORDS FOUND clause.

Restriction

This clause cannot be used with FIND FIRST, FIND NUMBER and FIND UNIQUE.

Example of IF NO RECORDS FOUND Clause:

 Copyright © Software AG 2002

```
/* EXAMPLE 'FNDIFN': FIND (IF NO RECORDS FOUND CLAUSE)
/*****************************************************************
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
```

```
      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 ***
```

System Variables with the FIND Statement

The Natural system variables *ISN, *NUMBER, and *COUNTER are automatically created for each FIND statement issued. A reference number must be supplied if the system variable was referenced outside the current processing loop or through a FIND UNIQUE, FIND FIRST, or FIND NUMBER statement. The format/length of each of these system variables is P10; this format/length cannot be changed.

*ISN

For Adabas databases, *ISN contains the Adabas internal sequence number (ISN) of the record currently being processed. *ISN is not available for the FIND NUMBER statement.

For VSAM databases, *ISN contains the relative byte address (RBA) of the record currently being processed (ESDS files only).

For DL/I and SQL databases, and with Entire System Server, *ISN is not available.

 Copyright © Software AG 2002

*NUMBER

Contains the number of records which satisfied the basic search criterion specified in the WITH clause.

For DL/I databases, *NUMBER contains "0" if no segment occurrences satisfy the search criterion, and a value of "9999" if at least one segment occurrence satisfies the search criterion.

For VSAM databases, *NUMBER only contains a meaningful value if the EQUAL TO operator is used in the search criterion. With any other operator, *NUMBER will be "0" if no records have been found; any other value indicates that records have been found, but the value will have no relation to the number of records actually found.

For SQL databases, see the section System Variables in the Natural Reference documentation.

With Entire System Server, *NUMBER is not available.

*COUNTER

Contains the number of times the processing loop has been entered.

Example using System Variables:

```
/* EXAMPLE 'FNDVAR': FIND (SYSTEM VARIABLES *ISN, *NUMBER, *COUNTER)
/**********************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 CITY
END-DEFINE
/**********************************************************************
LIMIT 3
FIND EMPLOY-VIEW WITH CITY = 'MADRID'
 DISPLAY NOTITLE PERSONNEL-ID NAME
                 *ISN *NUMBER *COUNTER
END-FIND
/**********************************************************************
END
```

```
PERSONNEL         NAME           ISN       NMBR      CNT
     ID
  --------- -------------------- --------- --------- ---------

  60000114  DE JUAN                    401        41         1
  60000136  DE LA MADRID               402        41         2
  60000209  PINERO                     406        41         3
```

Multiple FIND Statements

Multiple FIND statements may be issued to create nested loops whereby an inner loop is entered for each record selected in the outer loop.

Example of Multiple FIND Statements:

In the following example, first all people named SMITH are selected from the EMPLOYEES file. Then the PERSONNEL-ID from the EMPLOYEES file is used as the search key for an access to the VEHICLES file. The resulting report shows the NAME and FIRST-NAME (obtained from the EMPLOYEES file) of all people named SMITH as well as the MAKE of each car (obtained from the VEHICLES file) owned by these people:

```
/* EXAMPLE 'FNDMUL': FIND (USING MULTIPLE FILES)
/*********************************************************************
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. FIND EMPLOY-VIEW WITH NAME = 'SMITH'
VEH.  FIND VEHIC-VIEW WITH PERSONNEL-ID = EMP.PERSONNEL-ID
        IF NO RECORDS FOUND
         MOVE '*** NO CAR ***' TO MAKE
        END-NOREC
        DISPLAY NOTITLE
            EMP.NAME (IS=ON)
            EMP.FIRST-NAME (IS=ON) VEH.MAKE
      END-FIND
    END-FIND
/*********************************************************************
END
```

```
      NAME               FIRST-NAME              MAKE
-------------------- -------------------- --------------------

SMITH                GERHARD              ROVER
                     SEYMOUR              *** NO CAR ***
                     MATILDA              FORD
                     ANN                  *** NO CAR ***
                     TONI                 TOYOTA
                     MARTIN               *** NO CAR ***
                     THOMAS               FORD
                     SUNNY                *** NO CAR ***
                     JUNE                 JAGUAR
                     MARK                 FORD
                     LOUISE               CHRYSLER
                     MAXWELL              MERCEDES-BENZ
                                          MERCEDES-BENZ
                     ELSA                 CHRYSLER
                     CHARLY               CHRYSLER
                     LEE                  *** NO CAR ***
```

 Copyright © Software AG 2002

FIND FIRST

The FIND FIRST statement may be used to select and process the first record which meets the WITH and WHERE criteria.

For Adabas databases, the record processed will be the record with the lowest Adabas ISN from the set of qualifying records.

This statement does *not* initiate a processing loop.

Restrictions

FIND FIRST can only be used in reporting mode.

FIND FIRST is not available for DL/I and SQL databases.

The IF NO RECORDS FOUND clause must not be used in a FIND FIRST statement.

System Variables with FIND FIRST

The following Natural system variables are available with the FIND FIRST statement:

*ISN

Contains the Adabas ISN of the selected record. *ISN will be "0" if no record is found after the evaluation of the WITH and WHERE criteria.

*ISN is not available for VSAM databases or with Entire System Server.

*NUMBER

Contains the number of records found after the evaluation of the WITH criterion and before evaluation of any WHERE criterion. *NUMBER will be "0" if no record meets the WITH criterion.

*NUMBER is not available with Entire System Server.

*COUNTER

Contains "1" if a record was found; contains "0" if no record was found.

Example of FIND FIRST

See the program FNDFIR in the library SYSEXRM.

FIND NUMBER

The FIND NUMBER statement is used to determine the number of records which satisfy the WITH/WHERE criteria specified. It does *not* result in the initiation of a processing loop and *no data fields from the database are made available*.

Note:
Use of the WHERE clause may result in significant overhead.

Restrictions

The SORTED BY and IF NO RECORDS FOUND clauses must not be used with the FIND NUMBER statement.

The WHERE clause cannot be used in structured mode.

FIND NUMBER is not available for DL/I databases.

FIND NUMBER is not available with Entire System Server.

System Variables with FIND NUMBER

The following Natural system variables are available with the FIND NUMBER statement:

*NUMBER

Contains the number of records found after the evaluation of the WITH criterion.

*COUNTER

Contains the number of records found after the evaluation of the WHERE criterion.

*COUNTER is only available if the FIND NUMBER statement contains a WHERE clause.

 Copyright © Software AG 2002

Example of FIND NUMBER:

```
 * EXAMPLE '': FIND NUMBER
******************************
DEFINE DATA LOCAL
 1 EMPLOY-VIEW VIEW OF EMPLOYEES
   2 CITY
   2 BIRTH
 1 #BIRTH (D)
END-DEFINE
*
MOVE EDITED '19500101' TO #BIRTH (EM=YYYYMMDD)
*
FIND NUMBER EMPLOY-VIEW WITH CITY = 'MADRID'
                        WHERE BIRTH LT #BIRTH
*
WRITE NOTITLE 'TOTAL RECORDS SELECTED:       ' *NUMBER
            / 'TOTAL BORN BEFORE 1 JAN 1950: ' *COUNTER
*
END
```

```
TOTAL RECORDS SELECTED:                41
TOTAL BORN BEFORE 1 JAN 1950:          16
```

FIND UNIQUE

The FIND UNIQUE statement may be used to ensure that only one record is selected for processing. It does `not` result in the initiation of a processing loop. If a WHERE clause is specified, an automatic internal processing loop is created to evaluate the WHERE clause.

If no records or more than one record satisfy the criteria, an error message will be issued. This condition can be tested with the ON ERROR statement.

Restrictions

FIND UNIQUE can only be used in reporting mode.

FIND UNIQUE is not available for DL/I databases or with Entire System Server.

For SQL databases, FIND UNIQUE cannot be used. (Exception: On mainframe computers, FIND UNIQUE can be used for primary keys; however, this is only permitted for compatibility reasons and should not be used.)

The SORTED BY and IF NO RECORDS FOUND clauses must not be used with the FIND UNIQUE statement.

Example of FIND UNIQUE

See the program FNDUNQ in the library SYSEXRM.

FOR

```
FOR operand1 [ [:] = ]
             [ EQ    ] operand2
             [ FROM  ]
             [ TO    ]
             [ THRU  ] operand3  [ [STEP] operand4 ]

             statement...

END-FOR      (structured mode only)
[LOOP]       (reporting mode only)
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S					N	P	I	F								yes	yes
Operand2	C	S			N		N	P	I	F								yes	no
Operand3	C	S			N		N	P	I	F								yes	no
Operand4	C	S			N		N	P	I	F								yes	no

Related Statement: REPEAT

Function

The FOR statement is used to initiate a processing loop and to control the number of times the loop is processed.

Control Variable - *operand1* and Initial Setting - *operand2*

Operand1 is used to control the number of times the processing loop is to be executed. It may be a database field or a user-defined variable. The value specified after the keyword FROM (*operand2*) is assigned to the control variable field before the processing loop is entered for the first time. This value is incremented (or decremented if the STEP value is negative) using the value specified after the STEP keyword (*operand4*) each additional time the loop is processed.

The control variable value may be referenced during the execution of the processing loop and will contain the current value of the control variable.

TO Value - *operand3*

The processing loop is terminated when operand1 is greater than (or less than if the initial value of the STEP value was negative) the value specified for operand3.

 Copyright © Software AG 2002

STEP Value - *operand4*

The STEP value may be positive or negative. If a STEP value is not specified, an increment of "+1" is used.

The compare operation will be adjusted to "less than" or "greater than" depending on the sign of the STEP value when the loop is entered for the first time.

Operand4 must not be "0".

Consistency Check

Before the FOR loop is entered, the values of the operands are checked to ensure that they are consistent (that is, the value of *operand3* can be reached or exceeded by repeatedly adding *operand4* to *operand2*). If the values are not consistent, the FOR loop is not entered (however, no error message is output).

Example

```
/* EXAMPLE 'FOREX1S': FOR (STRUCTURED MODE)
/****************************************
DEFINE DATA LOCAL
1 #INDEX (I1)
1 #ROOT  (N2.7)
END-DEFINE
/****************************************
FOR #INDEX 1 TO 5
 COMPUTE #ROOT = SQRT (#INDEX)
 WRITE NOTITLE '=' #INDEX 3X '=' #ROOT
END-FOR
/****************************************
SKIP 1
FOR #INDEX 1 TO 5 STEP 2
 COMPUTE #ROOT = SQRT (#INDEX)
 WRITE '=' #INDEX 3X '=' #ROOT
END-FOR
/****************************************
END
```

```
#INDEX:    1   #ROOT:   1.0000000
  #INDEX:    2   #ROOT:   1.4142135
  #INDEX:    3   #ROOT:   1.7320508
  #INDEX:    4   #ROOT:   2.0000000
  #INDEX:    5   #ROOT:   2.2360679

  #INDEX:    1   #ROOT:   1.0000000
  #INDEX:    3   #ROOT:   1.7320508
  #INDEX:    5   #ROOT:   2.2360679
```

Equivalent reporting-mode example: See the program FOREX1R in the library SYSEXRM.

 Copyright © Software AG 2002

FORMAT

FORMAT [(*rep*)] *parameter*...

Function

The FORMAT statement is used to specify input and output parameter settings. Settings specified with a FORMAT statement override (at compilation time) default settings in effect for the session that have been set by a GLOBALS command, SET GLOBALS statement, or by the Natural administrator. These settings may in turn be overridden by parameters specified in a DISPLAY, INPUT, PRINT, WRITE, WRITE TITLE, or WRITE TRAILER statement.

The settings remain in effect until the end of a program or until another FORMAT statement is encountered.

A FORMAT statement does not generate any executable code in the Natural program. It is not executed in dependence of the logical flow of a program. It is evaluated during program compilation in order to set parameters for compiling DISPLAY, WRITE, and INPUT statements. The settings defined with a FORMAT statement are applicable to all DISPLAY, WRITE, and INPUT statements which follow. Multiple FORMAT statements are permitted within a program, but only one per report.

Report Specification - *rep*

The notation (*rep*) may be used to specify the identification of the report for which the FORMAT statement is applicable. A value in the range 0 - 31 or a logical name which has been assigned using the DEFINE PRINTER statement may be specified. If (*rep*) is not specified, the FORMAT statement will be applicable to the first report (report 0).

Parameters

The parameters can be specified in any order and must be separated by one or more spaces. A single entry must not be split between two statement lines.

Field sensitive parameter settings applied here will only be regarded for variable fields used in a INPUT, WRITE, DISPLAY or PRINT statement of the selected report. They do not apply for text-constants used in any of the mentioned statements.

```
DEFINE DATA LOCAL
1 VARI (A4)     INIT <'1234'>        /*     Output
END-DEFINE                           /*    Produced
FORMAT PM=I                          /*    ---------
WRITE 'Text'           VARI          /*    Text 4321
WRITE 'Text' (PM=I)    VARI          /*    txeT 4321
END
```

See the section Session Parameters in the Natural Reference documentation for a description of the parameters which may be used.

 Copyright © Software AG 2002

Example

```
/* EXAMPLE 'FMTEX1:' FORMAT
/************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 POST-CODE
  2 COUNTRY
END-DEFINE
/************************************************************
FORMAT AL=7    /* ALPHANUMERIC FIELD OUTPUT LENGTH
       FC=+    /* FILLER CHARACTER FOR FIELD HEADER
       GC=*    /* FILLER CHARACTER FOR GROUP HEADER
       HC=L    /* HEADER LEFT JUSTIFIED
       IC=<<   /* INSERT CHARACTERS
       IS=ON   /* IDENTICAL SUPPRESS ON
       TC=>>   /* TRAILING CHARACTERS
       UC==    /* UNDERLINE CHARACTER
       ZP=OFF  /* ZERO PRINT OFF
/************************************************************
LIMIT 5
READ EMPLOY-VIEW BY NAME
 DISPLAY NOTITLE NAME 3X CITY 3X POST-CODE 3X COUNTRY
END-READ
/************************************************************
END
```

```
NAME+++++++   CITY+++++++   POSTAL+++++   COUNTRY++++
                           ADDRESS
===========   ===========   ===========   ===========

<<ABELLAN>>   <<MADRID >>   <<28014  >>   <<E  >>
<<ACHIESO>>   <<DERBY  >>   <<DE3 4TR>>   <<UK >>
<<ADAM   >>   <<JOIGNY >>   <<89300  >>   <<F  >>
<<ADKINSO>>   <<BEDFORD>>   <<1730   >>   <<USA>>
              <<FRAMING>>   <<1701   >>
```

GET

```
GET [IN] [FILE] view-name
    [PASSWORD = operand1]
    [CIPHER = operand 2]
    [RECORD] { operand3 }
             { *ISN[(r)] } operand4 ...
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S				A												yes	no
Operand2	C	S					N											no	no
Operand3	C	S			N		N	P	I		B							yes	no
Operand4		S	A			A	N	P	I	F	B	D	T	L				yes	yes

Function

The GET statement is used to read a record with a given Adabas ISN (Internal Sequence Number).

The GET statement does not cause a processing loop to be initiated.

Restrictions

The GET statement cannot be used for DL/I and SQL databases.

The GET statement cannot be used with Entire System Server.

view-name

The name of a view as defined either within a DEFINE DATA statement or in a separate global or local data area. In reporting mode, view-name may also be the name of a DDM.

PASSWORD and CIPHER

These clauses are applicable only to Adabas and VSAM databases.

The PASSWORD clause is used to provide a password when retrieving data from an Adabas file which is password protected.

 Copyright © Software AG 2002

The CIPHER clause is used to provide a cipher key when retrieving data from an Adabas file which is enciphered.

See the statements FIND and PASSW for further information.

*ISN / operand3

The ISN must be provided either in the form of a numeric constant or user-defined variable (*operand3*), or via the Natural system variable *ISN.

Note for VSAM databases:
For VSAM ESDS, the RBA must be contained in a user-defined variable (numeric format) or must be specified as an integer constant. The same rules apply to VSAM RRDS with the exception that the RRN must be provided instead of the RBA.

Reference to Database Fields - operand4

Subsequent references to database fields that have been read with a GET statement must contain the label or line number of the GET statement.

Operand4 is not valid in structured mode.

Example

```
/* EXAMPLE 'GETEX1': GET
/*********************************************************************
DEFINE DATA LOCAL
1 PERSONS VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
1 SALARY-INFO VIEW OF EMPLOYEES
  2 NAME
  2 CURR-CODE (1:1)
  2 SALARY    (1:1)
1 #ISN-ARRAY  (B4/1:10)
1 #LINE-NR    ( N2)
END-DEFINE
/*********************************************************************
FORMAT PS=16
LIMIT 10
READ PERSONS BY NAME
   MOVE *COUNTER TO #LINE-NR
   MOVE *ISN TO #ISN-ARRAY (#LINE-NR)
   DISPLAY #LINE-NR PERSONNEL-ID NAME FIRST-NAME
/*********************************************************************
 AT END OF PAGE
   INPUT / 'PLEASE SELECT LINE-NR FOR SALARY INFORMATION:' #LINE-NR
   IF #LINE-NR = 1 THRU 10
      GET SALARY-INFO #ISN-ARRAY (#LINE-NR)
      WRITE / SALARY-INFO.NAME
              SALARY-INFO.SALARY    (1)
              SALARY-INFO.CURR-CODE (1)
   END-IF
 END-ENDPAGE
/*********************************************************************
END-READ
END
```

```
PAGE      1                                                97-09-29  14:53:1

  #LINE-NR PERSONNEL         NAME              FIRST-NAME
              ID
  -------- --------- -------------------- --------------------

    1      60008339  ABELLAN              KEPA
    2      30000231  ACHIESON             ROBERT
    3      50005800  ADAM                 EDWIN
    4      20005700  ADKINSON             TIMMIE
    5      20008600  ADKINSON             MARTHA
    6      20008800  ADKINSON             JEFF
    7      20009800  ADKINSON             PHYLLIS
    8      20011000  ADKINSON             BOB
    9      20012700  ADKINSON             HAZEL
   10      20013800  ADKINSON             DAVID

  PLEASE SELECT LINE-NR FOR SALARY INFORMATION: 1

  ABELLAN                1450000 PTA
```

 Copyright © Software AG 2002

GET SAME

Note:
This statement is only valid for Natural users who are using Adabas or VSAM. This statement cannot be used with Entire System Server.

Structured Mode Syntax

```
GET SAME [(r)]
```

Reporting Mode Syntax

```
GET SAME [(r)] [operand1 ...]
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1		S	A			A	N	P			B						no	yes

Function

The GET SAME statement is used to re-read the record currently being processed. It is most frequently used to obtain database array values (periodic groups or multiple-value fields) if the number and range of existing or desired occurrences was not known when the record was initially read.

Statement Reference - *r*

The notation "(*r*)" is used to specify the statement which contains the FIND or READ statement used to initially read the record. If "(*r*)" is not specified, the GET SAME statement will be related to the innermost active processing loop. "(*r*)" may be specified as a reference statement number or as a statement label.

operand1

As operand1, you specify the field(s) to be made available as a result of the GET SAME statement.

Note:
Operand1 cannot be specified if the field is defined in a DEFINE DATA statement.

Restrictions

An UPDATE or DELETE statement must not reference a GET SAME statement. These statements should instead make reference to the FIND, READ or GET statement used to read the record initially.

For VSAM databases, GET SAME can only be applied to ESDS and RRDS. For ESDS, the RBA must be contained in a user-defined variable (numeric format) or be specified as an integer constant. The same applies to RRDS, except that the RRN must be provided instead of the RBA.

Example

```
/* EXAMPLE 'GSAEX1S': GET SAME
/*******************************************************
DEFINE DATA LOCAL
1 I          (P3)
1 #NAME      (A30)
1 POST-ADDRESS VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 ADDRESS-LINE   (I:I)
  2 C*ADDRESS-LINE
  2 POST-CODE
  2 CITY
END-DEFINE
/*******************************************************
FORMAT PS=20
MOVE 1 TO I
READ POST-ADDRESS BY NAME
  COMPRESS NAME FIRST-NAME INTO #NAME WITH DELIMITER ','
  WRITE // 12T #NAME
  WRITE /  12T ADDRESS-LINE (I.1)
  IF C*ADDRESS-LINE > 1
    FOR I = 2 TO C*ADDRESS-LINE
      GET SAME                      /* READ NEXT OCCURRENCE
      WRITE 12T ADDRESS-LINE (I.1)
    END-FOR
  END-IF
  WRITE / POST-CODE CITY
  SKIP 3
END-READ
END
```

 Copyright © Software AG 2002

```
PAGE      1                                                  97-09-01  11:35:33



             ABELLAN,KEPA

             CASTELAN 23-C

  28014      MADRID




             ACHIESON,ROBERT

             144 ALLESTREE LANE
             DERBY
             DERBYSHIRE

  DE3 4TR    DERBY
```

GET TRANSACTION DATA

Note:
This statement is only valid for transactions applied to Adabas databases, or to DL/I databases in a batch-oriented BMP region (in IMS environments only).

```
GET TRANSACTION [DATA] operand1...
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S				A	N	P	I	F	B	D	T					yes	yes

Related Statements: END TRANSACTION

Function

The GET TRANSACTION DATA statement is used to read the data saved with a previous END TRANSACTION statement.

GET TRANSACTION DATA does not create a processing loop.

Note for DL/I:
The GET TRANSACTION DATA statement retrieves checkpoint data saved by an END TRANSACTION statement.

System Variable *ETID

The content of the Natural system variable *ETID identifies the transaction data to be retrieved from the database.

Field Specification operand1

The sequence, lengths and formats of the fields used in the GET TRANSACTION DATA statement must be identical to the sequence, lengths and formats of the fields specified with the corresponding END TRANSACTION statement.

Note for DL/I:
The first operand1 must be an 8-byte checkpoint ID.

No Transaction Data Stored

If the GET TRANSACTION DATA statement is issued and no transaction data are found, all fields specified in the GET TRANSACTION DATA statement will be filled with blanks regardless of format definition. Make sure that arithmetic operations are not performed on "empty" transaction data, because this would result in an abnormal termination of the program.

 Copyright © Software AG 2002

Example

```
/* EXAMPLE 'GTREX1S': GET TRANSACTION DATA (STRUCTURED MODE)
/*****************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
 2 PERSONNEL-ID
 2 NAME
 2 FIRST-NAME
 2 MIDDLE-I
 2 CITY
1 #PERS-NR (A8) INIT <' '>
END-DEFINE
/*****************************************************************
GET TRANSACTION DATA #PERS-NR
IF #PERS-NR NE ' '
 WRITE 'LAST TRANSACTION PROCESSED FROM PREVIOUS SESSION' #PERS-NR
END-IF
/*****************************************************************
REPEAT
/*****************************************************************
 INPUT  10X 'ENTER PERSONNEL NUMBER TO BE UPDATED:' #PERS-NR
 IF #PERS-NR = ' '
  STOP
 END-IF
/*****************************************************************
 FIND EMPLOY-VIEW WITH PERSONNEL-ID = #PERS-NR
  IF NO RECORDS FOUND
   REINPUT 'NO RECORD FOUND'
  END-NOREC
  INPUT (AD=M) PERSONNEL-ID (AD=O)
             / NAME
             / FIRST-NAME
             / CITY
  UPDATE
  END TRANSACTION #PERS-NR
 END-FIND
/*****************************************************************
END-REPEAT
END
```

HISTOGRAM

```
HISTOGRAM [ ALL ] [VALUE] [IN] [FILE] view-name
          [ (operand1) ]
    [PASSWORD = operand2]
    [ [IN] { ASCENDING  } [SEQUENCE] ]
           { DESCENDING }
           { VARIABLE operand3 }
    [VALUE] [FOR] [FIELD] operand4
    [STARTING/ENDING-clause]
    [WHERE logical-condition]
    statement...
END-HISTOGRAM      (structured mode only)
[LOOP]             (reporting mode only)
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S					N	P	I									yes	no
Operand2	C	S				A												yes	no
Operand3		S				A												yes	no
Operand4		S				A	N	P	I	F	B	D	T	L				no	no

Related Statements: FIND | READ

Function

The HISTOGRAM statement is used to read the values of a database field which is defined as a descriptor, subdescriptor, or a superdescriptor. The values are read directly from the Adabas inverted lists (or VSAM index).

The HISTOGRAM statement causes a processing loop to be initiated but does not provide access to any database fields other than the field specified in the HISTOGRAM statement.

Note for SQL databases:
HISTOGRAM returns the number of rows which have the same value in a specific column.

Restrictions

This statement cannot be used with DL/I databases or Entire System Server.

 Copyright © Software AG 2002

When applied to a VSAM database, the HISTOGRAM statement is only valid for KSDS and ESDS.

Processing Loop Limit - operand1/ALL

You can limit the number of descriptor values to be processed with the HISTOGRAM statement by specifying *operand 1* - either as a numeric constant (0 to 99999999) or as a user-defined variable (containing an integer value). ALL may optionally be specified to emphasize that all descriptor values are to be processed.

For this statement, the specified limit has priority over a limit set with a LIMIT statement.

If a smaller limit is set with the LT parameter, the LT limit applies.

Notes:
If you wish to process a 4-digit number of descriptor values, specify it with a leading zero: (0nnnn); because Natural interprets every 4-digit number enclosed in parentheses as a line-number reference to a statement.
Operand1 is evaluated when the HISTOGRAM loop is entered. If the value of operand1 is modified within the HISTOGRAM loop, this does not affect the number of values read.

view-name

As *view-name*, you specify the name of a view, which is defined either within a DEFINE DATA statement or in a separate global or local data area.

The view must not contain any other fields apart from the field used in the HISTOGRAM statement (*operand4*).

If the field in the view is a periodic-group field or multiple-value field that is defined with an index range, only the first occurrence of that range is filled by the HISTOGRAM statement; all other occurrences are not affected by the execution of the HISTOGRAM statement.

In reporting mode, *view-name* may also be the name of a DDM.

PASSWORD Clause

The PASSWORD clause is used to provide a password (operand2) when retrieving data from an Adabas file which is password-protected. See the statements FIND and PASSW for further information.

SEQUENCE Clause

This clause can only be used for Adabas, VSAM and SQL databases.

With this clause, you can determine whether the values are to be read in ascending sequence or in descending sequence.

- The default sequence is ascending (which may, but need not, be explicity specified by using the keyword ASCENDING).
- If the values are to be read in descending sequence, you specify the keyword DESCENDING.
- If it is to be determined at runtime whether the values are to be read in ascending or descending sequence, you specify the keyword VARIABLE followed by a variable (*operand3*). The value of *operand3* at the start of the HISTOGRAM processing loop then determines the sequence. *Operand3* has to be of format/length A1 and can contain the value "A" (for "ascending") or "D" (for "descending").

Note for Adabas databases:
Descending sequence requires the following Adabas versions (or above): Version 3.1 on UNIX and Windows, Version 3.2 on OpenVMS, and Version 6.2 on mainframe computers.

Note for SQL databases:
On mainframe computers, the VARIABLE option cannot be used for SQL databases.

Examples of SEQUENCE Clause:

See the programs HSTDSCND and HSTVSEQ in the library SYSEXRM.

Descriptor - operand4

As *operand4*, a descriptor, subdescriptor, superdescriptor or hyperdescriptor may be specified.

A descriptor contained within a periodic group may be specified with or without an index. If no index is specified, the descriptor will be selected if the value specified is located in any occurrence. If an index is specified, the descriptor will be selected only if the value is located in the occurrence specified by the index. The index specified must be a constant. An index range must not be used.

For a descriptor which is a multiple-value field an index must not be specified; the descriptor will be selected if the value is located in the record regardless of the position of the value.

 Copyright © Software AG 2002

STARTING-ENDING-clause

```
[ [STARTING] [WITH ] [VALUES] operand5 ] [ [ THRU ] operand6 ]
             [FROM ]                     [ENDING AT]
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand5	C	S				A	N	P	I	F	B	D	T	L				yes	no
Operand6	C	S				A	N	P	I	F	B	D	T	L				yes	no

Starting and ending values may be specified using the keywords STARTING and ENDING (or THRU) followed by a constant or a user-defined variable representing the value with which processing is to begin/end.

If a starting value is specified and the value is not present, the next higher value is used as the starting value. If no higher value is present, the HISTOGRAM loop will not be entered.

If an ending value is specified, values will be read up to and including the ending value.

Hexadecimal constants may be specified as a starting or ending value for descriptors of format A or B.

WHERE Clause

The WHERE clause may be used to specify an additional selection criterion (*logical-condition*) which is evaluated *after* a value has been read and *before* any processing is performed on the value (including the AT BREAK evaluation).

The descriptor specified in the WHERE clause must be the same descriptor referenced in the HISTOGRAM statement. No other fields from the selected file are available for processing with a HISTOGRAM statement. See the section Logical Condition Criteria in the Natural Reference documentation for the syntax and explanation of logical criteria.

System Variables

The following Natural system variables are available with the HISTOGRAM statement:

- ***NUMBER** - Contains the number of database records that contain the last value read. (For SQL databases, see the description of the system variable *NUMBER in the Natural Reference documentation.)
- ***ISN** - Contains the number of the occurrence in which the descriptor value last read is contained. *ISN will contain "0" if the descriptor is not contained within a periodic group. *ISN is not available for SQL and VSAM databases.
- ***COUNTER** - Contains a count of the total number of values which have been read (after evaluation of the WHERE clause).

*NUMBER and *ISN are only set after the evaluation of the WHERE clause. They must not be used in the logical condition of the WHERE clause.

 Copyright © Software AG 2002

Example

```
/* EXAMPLE 'HSTEX1S': HISTOGRAM (STRUCTURED MODE)
/*****************************************************
DEFINE DATA LOCAL
 1 EMPLOY-VIEW VIEW OF EMPLOYEES
   2 CITY
END-DEFINE
/*****************************************************
LIMIT 8
HISTOGRAM EMPLOY-VIEW CITY STARTING FROM 'M'
 DISPLAY NOTITLE CITY
         'NUMBER OF/PERSONS' *NUMBER *COUNTER
END-HISTOGRAM
/*****************************************************
END
```

```
      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
```

Equivalent reporting-mode example: See the program HSTEX1R in the library SYSEXRM.

IF

Structured Mode Syntax

```
IF logical-condition
   [THEN] statement...
   [ELSE statement...]
END-IF
```

Reporting Mode Syntax

```
IF logical-condition
   [THEN] { statement
          { DO statement...DOEND }
   [ELSE { statement
         { DO statement...DOEND } ]
```

Related Statements: DECIDE FOR | DECIDE ON

Function

The IF statement is used to control execution of a statement or group of statements based on a logical condition.

Note:
If no action is to be performed if the condition is met, you specify the statement IGNORE in the THEN clause.

logical-condition

The logical condition which is used to determine whether the statement or statements specified with the IF statement are to be executed.

Examples:

```
IF #A = #B
IF LEAVE-TAKEN GT 30
IF #SALARY(1) * 1.15 GT 5000
IF SALARY (4) = 5000 THRU 6000
IF DEPT = 'A10' OR = 'A20' OR = 'A30'
```

 Copyright © Software AG 2002

For further information, see the section Logical Condition Criteria in the Natural Reference documentation.

THEN

In the THEN clause, you specify the *statement(s)* to be executed if the logical condition is true.

ELSE

In the ELSE clause, you specify the *statement(s)* to be executed if the logical condition is `not` true.

Example

```
 * EXAMPLE 'IFEX1S': IF (STRUCTURED MODE)
****************************************
DEFINE DATA LOCAL
 1 EMPLOY-VIEW VIEW OF EMPLOYEES
   2 PERSONNEL-ID
   2 NAME
   2 FIRST-NAME
   2 SALARY (1)
   2 BIRTH
 1 VEHIC-VIEW VIEW OF VEHICLES
   2 PERSONNEL-ID
   2 MAKE
 1 #BIRTH (D)
END-DEFINE
*
MOVE EDITED '19450101' TO #BIRTH (EM=YYYYMMDD)
SUSPEND IDENTICAL SUPPRESS
LIMIT 20
*
FND. FIND EMPLOY-VIEW WITH CITY = 'FRANKFURT'
                      SORTED BY NAME BIRTH
  IF SALARY (1) LT 40000
    WRITE NOTITLE '*****' NAME 30X 'SALARY LT 40000'
  ELSE
    IF BIRTH GT #BIRTH
      FIND VEHIC-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (FND.)
        DISPLAY (IS=ON) NAME BIRTH (EM=YYYY-MM-DD)
                        SALARY (1) MAKE (AL=8)
      END-FIND
    END-IF
  END-IF
END-FIND
END
```

 Copyright © Software AG 2002

```
        NAME            DATE      ANNUAL     MAKE
                         OF       SALARY
                       BIRTH
-------------------- ---------- ---------- --------

BAECKER              1956-01-05      74400 BMW
***** BECKER                                          SALARY LT 40000
BLOEMER              1979-11-07      45200 FIAT
FALTER               1954-05-23      70800 FORD
***** FALTER                                          SALARY LT 40000
***** GROTHE                                          SALARY LT 40000
***** HEILBROCK                                       SALARY LT 40000
***** HESCHMANN                                       SALARY LT 40000
HUCH                 1952-09-12      67200 MERCEDES
***** KICKSTEIN                                       SALARY LT 40000
***** KLEENE                                          SALARY LT 40000
***** KRAMER                                          SALARY LT 40000
```

Equivalent reporting-mode example: See the program IFEX1R in the library SYSEXRM.

IF SELECTION

Structured Mode Syntax

```
IF SELECTION [NOT UNIQUE [IN [FIELDS] ] ] operand1 ...
   [THEN] statement...
   [ELSE  statement...]
END-IF
```

Reporting Mode Syntax

```
IF SELECTION [NOT UNIQUE [IN [FIELDS] ] ] operand1 ...
   [THEN] { statement
          { DO statement...DOEND }
   [ELSE { statement
         { DO statement...DOEND } ]
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1		S	A			A							L	C			yes	no

Function

The IF SELECTION statement is used to verify that in a sequence of alphanumeric fields one and only one contains a value.

The statements specified in the THEN clause will be executed if one of the following conditions is true:

- None of the specified fields (*operand1*) contains a value.
- More than one of the specified fields (*operand1*) contains a value.

This statement is generally used to verify that a terminal user has entered only one function in response to a map displayed via an INPUT statement.

Note:
If **no** action is to be performed if one of the conditions is met, you specify the statement IGNORE in the THEN clause.

 Copyright © Software AG 2002

Selection Field - operand1

As *operand1* you specify the fields which are to be checked.

If you specify a control variable (format C), it is considered to contain a value if its status has been changed to "MODIFIED".

Example

```
/* EXAMPLE 'IFSEL': IF SELECTION
/************************************************
DEFINE DATA LOCAL
 1 #A (A1)
 1 #B (A1)
END-DEFINE
/************************************************
INPUT 'SELECT FUNCTION:' //
  10X 'READ EMPLOYEES FILE:' #A
  10X 'READ VEHICLES FILE: ' #B
/************************************************
IF SELECTION NOT UNIQUE #A #B
 REINPUT 'PLEASE ENTER ONE FUNCTION ONLY:'
END-IF
/************************************************
IF #A NE ' '
 FETCH 'READEMPL'
END-IF
IF #B NE ' '
 FETCH 'READVEHC'
END-IF
/************************************************
END
```

```
SELECT FUNCTION:

             READ EMPLOYEES FILE: x          READ VEHICLES FILE: x
```

```
PLEASE ENTER ONE FUNCTION ONLY:
SELECT FUNCTION:

             READ EMPLOYEES FILE: x          READ VEHICLES FILE: x
```

IGNORE

```
IGNORE
```

Function

The IGNORE statement is an "empty" statement which itself does not perform any function.

During the development phase of an application, you can insert IGNORE temporarily within statement blocks in which one or more statements are required, but which you intend to code later (for example, within AT BREAK or AT START/END OF DATA). This allows you to continue programming in another part of the application without the as yet incomplete statement block leading to an error.

The IGNORE statement can also be used in condition statements like IF or DECIDE FOR, if no function is to be performed in the case of a condition being met.

Example

```
...
...
AT TOP OF PAGE
   IGNORE         /* top-of-page processing still to be coded
END-TOPPAGE
...
...
```

 Copyright © Software AG 2002

INCLUDE

```
INCLUDE copycode-name [operand1...99]
```

Operand	Possible Structure	Possible Formats	Referencing Permitted	Dynamic Definition
Operand1	C	A	no	no

Function

The INCLUDE statement is used to include source lines from an external object of type copycode into another object at compilation.

The INCLUDE statement is evaluated at *compilation* time. The source lines of the copycode will not be physically included in the source of the program that contains the INCLUDE statement, but they will be included during the program compilation and thus in the resulting object module.

A source code line which contains an INCLUDE statement must not contain any other statement.

copycode-name

As *copycode-name* you specify the name of the copycode whose source is to be included.
The *copycode-name* may contain an ampersand (&); at compile time, this character will be replaced by the current value of the Natural system variable *LANGUAGE. This feature allows the use of multi-lingual *copycode-name*s.

The object you specify must be of the type copycode. The copycode must be contained either in the same library as the program which contains the INCLUDE statement or in the respective steplib (the default steplib is SYSTEM).

When the source of a copycode is modified, all programs using that copycode must be compiled again to reflect the changed source in their object codes.

The source code of the copycode must consist of syntactically complete statements.

operand1

You can dynamically insert values in the copycode which is included. These values are specified with *operand1*.

In the copycode, the values are referenced with the notation **&***n***&**""; that is, you mark the position where a value is to be inserted with "&*n*&". "n" is the sequential number of each value passed with the INCLUDE statement. For example, "&3&" would refer to the third value specified with the statement.

For every "&*n*&" notation in the copycode you must specify a value in the INCLUDE statement. For example, if the copycode contains "&5&", *operand1* must be specified at least five times.

Values that are specified in the INCLUDE statement but not referenced in the copycode will be ignored.

Example 1

```
/* EXAMPLE 'INCEX1:' INCLUDE
/******************************
/* ...
/* ...
/* ...
/******************************
INCLUDE MEM1
/******************************
/* ...
/* ...
/* ...
END
```

Example 2

Copycode to be included:

```
/* EXAMPLE 'COPEX1': COPYCODE USING PARAMETERS
READ (&4&) &1&  BY &2&  = &3&
  DISPLAY &2&
  IF &2&  = &5&  DO
    WRITE 5X 'LAST RECORD FOUND'
    STOP
  DOEND
 LOOP
```

Program containing INCLUDE statement:

```
/* EXAMPLE 'COPEX2': PROGRAM USING COPYCODE WITH PARAMETERS
*
INCLUDE COPEX1 'EMPLOYEES' 'NAME' '''ALDEN''' '20' '''ALLEN'''
END
```

 Copyright © Software AG 2002

```
Page      1                                                  02-01-30  14:01:26

       LAST-NAME
  --------------------

  ALDEN
  ALEXANDER
  ALEXANDER
  ALEXANDER
  ALEXANDER
  ALEXANDER
  ALEXANDER
  ALEXANDER
  ALEXANDER
  ALEXANDER
  ALLDERIDGE
  ALLDERIDGE
  ALLDERIDGE
  ALLEN
       LAST RECORD FOUND
```

Example 3

Copycode 1 to be included:

```
   /* EXAMPLE 'Object ASSIGN':
       &1& := &2&
```

Copycode 2 to be included:

```
   /* EXAMPLE 'Object ASET':
      INCLUDE ASSIGN &1& := &2&
```

Program containing INCLUDE statement:

```
   /* EXAMPLE: PROGRAM USING COPYCODE WITH PARAMETERS
   *
     reset #a(i4)
    move 123 to #a
    write '=' #a
     include ASSIGN '#a' '5'
    write '=' #a
     move 123 to #a
    write '=' #a
     include ASET '''#a'''  '''5'''
    write '=' #a
    end
```

```
Page   1                    02-01-30  14:01:26

     #A:          123
     #A:            5
     #A:          123
     #A:            5
```

 Copyright © Software AG 2002

INPUT

- Syntax 1 - Dynamic Screen Layout Specification
- Syntax 2 - Using Predefined Map Layout

Related Statement: REINPUT

Function

The INPUT statement is used in interactive mode to create a formatted screen or map for data entry.
It may also be used in conjunction with the Natural stack (see the STACK statement); and on mainframe computers, it may also be used to provide user data for programs being executed in batch mode.

Input Modes

The INPUT statement may be used in screen, forms, or keyword/delimiter mode. Screen mode is generally used with video terminals/screens. Forms mode may be used with TTY terminals. Delimiter mode is used with TTY terminals, and also in batch mode (on mainframe computers). The default mode is screen mode.

You can change the input mode with the session parameter IM or the terminal commands %F and %D.

Screen Mode

In screen mode, execution of the INPUT statement results in the display of a screen according to the fields and positioning notation specified. The message line of the screen is used by Natural for error messages. The position of the message line (top or bottom of screen) may be controlled by the terminal command %M. The terminal user may position to specific fields using the various tabulation keys.

As Natural allows for screen window processing, the layout of the logical screen map may be larger (theoretically 250 characters per line and 250 lines, but limited by the internal screen buffer) than the physical screen size.

The windowing terminal command %W may be used to modify logical and physical window position and size (see the terminal command %W for details of window handling).

For input fields (AD=A or AD=M) that are not fully displayed on the physical screen, the following rules apply:

- Input fields whose beginning is not inside the window are always made protected.
- Input fields which begin inside and end outside the window are only made protected if the values they contain cannot be displayed completely in the window. Please note that in this case it is decisive whether the value length, not the field length, exceeds the window size. Filler characters (as specified with the profile parameter FC or session parameter AD) do not count as part of the value.
- Before an input field thus protected can be accessed and processed, the window size must be adjusted so as to fully display the field or value respectively (see the terminal command %W).

Non-Screen Modes

The INPUT statement may be used for an operation on line-oriented devices or for the processing of batch input from sequential files.

The same map layouts as defined for screen mode operation can also be processed in non-screen mode.

Forms mode and keyword/delimiter mode are also available to process the input either by simulating the screen layout in line mode or by just processing the data without any map layout.

See also:
Using the INPUT Statement in Non-Screen Modes | Using the INPUT Statement in Batch Mode on Mainframe Computers | Processing Data from the Natural Stack

Entering Data in Response to an INPUT Statement

Data for an alphanumeric field must be entered left-justified. Any character, including a blank, is meaningful. The data are assigned one character per byte to the internal field. Data entered for an alphanumeric field are not validated.

Lower and upper case translation are controlled by the terminal commands %L and %U as well as the attributes AD=T and AD=W.

Data for a numeric field may be placed anywhere in the input field. Leading and/or trailing blanks, leading zeros, a leading sign and one decimal point are permitted. Natural adjusts the value according to the internal definition of the field. If SG=OFF is specified, Natural does not assume or allocate a position for a sign position. Data for a field defined with format P must be entered in decimal form. Natural will convert decimal to packed wherever necessary. A field containing all blanks is interpreted as a zero value. Data for a numeric field are validated by Natural to ensure that the value consists only of leading and/or trailing blanks, an optional leading sign, an optional decimal point, and numeric characters. If no decimal point is entered, it is assumed to be to the right of the value entered.

Data for a binary field must be entered for all positions (two characters per byte). Only valid hexadecimal characters (0 - 9, A - F) may be used. A blank (H’20’ in ASCII or H’40’ in EBCDIC respectively) is valid and is converted to binary zeros. Data for a binary field are validated by Natural for hexadecimal characters.

Data for format L fields may be entered as blank ("false") or non-blank ("true").

Data for format F, D, and T are entered according to the rules stated for F, D, and T constants.

Error Correction

If the value entered in an input field does not correspond to the format or edit mask of the field, Natural displays an error message (without terminating the program execution) and positions the cursor in the field in error. The user may then enter a valid value, whereupon processing continues.

Split-Screen Feature

In general, each INPUT statement generates a new page (or terminal screen) of output. Any INPUT statement which is specified within an AT END OF PAGE statement will not produce a new screen. This feature allows for the creation of a split screen where the upper portion of the screen may be used to display multiple lines and the lower portion can be used to create an input map for communication. The PS parameter (page size) should be used, either in a SET GLOBALS or FORMAT statement, to set the logical page size to ensure that the input map is built on the same physical screen.

The first INPUT line will be placed after the last displayed line. If the NO ERASE option is used, the first INPUT line will be placed at the top of the page.

 Copyright © Software AG 2002

Syntax 1 - Dynamic Screen Layout Specification

```
INPUT [WINDOW = 'window-name'] [NO ERASE]
      [(statement-parameters)]
      [WITH-TEXT-option]
      [MARK-option]
      [ALARM-option]
      [nX]   [ 'text' [(attributes)]      ]   [ *IN    ]
      [nT]   [ 'c'(n) [(attributes)]      ]   [ *OUT   ]  {operand1 [(parameter)]}...
      [x/y]  [        '-'                 ]   [ *OUTIN ]
             [        '='                 ]
             [        /...                ]
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S	A	G	N	A	N	P	I	F	B	D	T	L		G		yes	yes

This form of the INPUT statement is used to create a layout of an INPUT screen, or to create an INPUT data layout which is to be read (on mainframe computers) in batch mode from a sequential input file.

INPUT WINDOW='window-name'

With the option WINDOW='*window-name*', you indicate that the INPUT statement is to be executed for the specified window. The specified window must be defined in a DEFINE WINDOW statement. The specified window is only active for the duration of that INPUT statement, and is automatically deactivated when the INPUT statement has been executed.

See also the statements DEFINE WINDOW and SET WINDOW.

NO ERASE

NO ERASE causes a screen map of an INPUT statement to be overlaid onto an existing screen without erasing the screen contents.

Screen as used here refers to a logical screen rather than a physical screen.

All unprotected fields that existed on the screen are converted to protected (display only) fields. The old data remain on the screen until the new layout is displayed. If a field from the new screen content partially overlays an existing field, the one character before the new field and the next character in the existing field will be replaced by a blank.

statement-parameters

One or more parameters, enclosed within parentheses, may be specified immediately after the INPUT statement or an element being displayed.

Each parameter specified in this manner will override any previous parameter specified in a GLOBALS command, SET GLOBALS or FORMAT statement. If more than one parameter is specified, one or more blanks must be present between each entry. An entry may not be split between two statement lines.

The parameter settings appiled here will only be regarded for variable fields, but they have no effect on text-constants. If you would like to set field attributes for a text-constant, they have to be set explicitly for this element.

Example:

```
DEFINE DATA LOCAL
1 VARI (A4)     INIT <'1234'>                  /*    Displays
END-DEFINE                                     /*       as
FORMAT AD=M                                    /*    ---------
INPUT           'Text'           VARI          /*    Text 1234
INPUT (PM=I)    'Text'           VARI          /*    Text 4321
INPUT           'Text' (PM=I)    VARI (PM=I)   /*    txeT 4321
INPUT           'Text' (PM=I)    VARI          /*    txeT 1234
END
```

Examples of using parameters at the statement and element level are provided on the next page. For information on which parameters you can use and a description of each parameter, see the section Session Parameters in the Natural Reference documentation.

WITH TEXT-option

[WITH] TEXT [*operand1 | operand2] [(attributes)] [,operand3]...7

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S					N	P	I		B							yes	no
Operand2	C	S				A												yes	no
Operand3	C	S				A	N	P	I	F	B	D	T	L				yes	no

WITH TEXT is used to provide text which is to be displayed in the message line. This is usually a message indicating what action should be taken to process the screen or to correct an error.

Message Text from Natural Message File - *operand1

Operand1 represents the number of the message text as stored in the Natural message file. The value provided is used to access the Natural message file in order to retrieve a message text.

Messages are stored in the Natural message file by library. A maximum of 9999 messages may be stored per library.

 Copyright © Software AG 2002

For more information on messages, see Generating Message Files in the Natural SYSERR Utility documentation.

Message Text - operand2 and Attributes - attributes

Operand2 represents the message to be placed in the message line.

Attributes may be used to assign display and color attributes for operand1/2. The following *attributes* may be used:

```
[ AD = { B } ]   [ CD = { BL } ]
       { C }             { GR }
       { D }             { NE }
       { I }             { PI }
       { N }             { RE }
       { U }             { TU }
       { V }             { YE }
         1                 2
```

1. Display attributes (see the session parameter AD in the Natural Reference documentation).
2. Color attributes (see the session parameter CD in the Natural Reference documentation).

Dynamic Replacement of Message Text - operand3

Operand3 represents a numeric or text constant or the name of a variable.

The values provided are used to replace parts of the message text.

The notation ":n:" is used within the message text as a reference to *operand3* contents, where "n" represents the *operand3* occurrence (1 - 7).

Example:

```
...
  MOVE 'MESSAGE-1' TO #FIELD
  ...
  INPUT WITH TEXT 'THE ERROR IS :1: ',#FIELD ...
  ...
```

This would cause the following message to be output:

THE ERROR IS MESSAGE-1

Note:
Multiple specifications of operand3 must be separated from each other by a comma. If the comma is used as a decimal character (as defined with the session parameter DC) and numeric constants are specified as operand3, put blanks before and after the comma so that it cannot be misinterpreted as a decimal character.
Alternatively, multiple specifications of operand3 can be separated by the input delimiter character (as defined with the session parameter ID); however, this is not possible in the case of ID=/ (slash), because the slash has a different meaning in the INPUT statement syntax.

Insignificant zeros or blanks will be removed from the field value before it is displayed in a message.

MARK-option

MARK [**POSITION** *operand4* [**IN**]] [**FIELD**] { *operand1* | **fieldname* }

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand4	C	S					N	P	I								yes	no
Operand1	C	S	A				N	P	I								yes	no

Field to be Marked - operand1

Each AD=A or AD=M (that is, non-protected field) specified in an INPUT statement is assigned a field reference number, beginning with 1. The MARK option positions the cursor to the field number specified. The **fieldname* notation may be used to position to a field using the name of the field as a reference.

 Copyright © Software AG 2002

MARK POSITION

With MARK POSITION, you can have the cursor placed at a specific position - as specified with *operand4* - within a field. *Operand4* must not contain decimal digits.

Examples:

```
MARK 3
MARK #A
MARK *#
MARK POSITION 3 IN #A
```

ALARM-option

```
[ [AND] [SOUND] ALARM]
```

This option causes the sound alarm feature of the terminal to be activated when the INPUT statement is executed. The appropriate hardware must be available to be able to use this feature.

Default Prompting Text

Unless the session parameter IP (INPUT prompting) is set to IP=OFF, the field name of the field used in an INPUT statement will be displayed preceding the field value (forms mode) or as a prompting keyword to select the field (keyword/delimiter mode). This default field name may be overridden by specifying either a ’text’ element (which replaces the default name) or ’-’ (which suppresses the display of the default field name) immediately preceding the field name.

Field Positioning, Text Specification, Attribute Assignment

```
[ nX ]   [ ´text´ [(attributes)] ]   [ *IN    ]
[ nT ]   [ ´c´ (n) [(attributes)] ]  [ *OUT   ] {operand1 [(parameter)]}...
[ x/y ]  [ ´-´                    ]  [ *OUTIN ]
         [ ´=´                    ]
         [ /...                   ]
```

Several notations are available for field positioning, attribute assignment, and text creation.

*n*X

Causes *n* spaces to be inserted between fields.

Note: (for Mainframes only)
This notation inserts n spaces between columns. n must not be "0".

*n*T

Causes positioning (tabulation) to print position *n*.

x/y

Places the next element on line *x*, beginning in column *y*. *y* must not be "0". Backward positioning in the same line is not permitted.

'text'

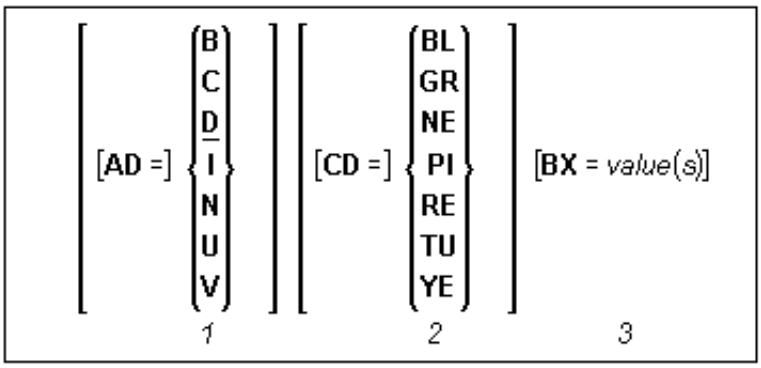
Causes text to be displayed write protected.

'c' (n)

Identical to '*text*', except that the character *c* is displayed *n* times. *n* must be 1 - 132.

attributes

Indicates the attributes to be used for display. The following *attributes* may be used:



1. Display attributes (see the session parameter AD in the Natural Reference documentation).
2. Color attributes (see the session parameter CD in the Natural Reference documentation).
3. Outlining attributes (see the session parameter BX in the Natural Reference documentation).

Minus Sign '-'

When placed before a field, '-' suppresses the generation of a field name as prompting text.

Note:
Any text string before a field will replace the field name as prompting text.

Equal Sign '='

When placed before a field, '=' results in the display of the field heading followed by the field contents.

Slash Sign /

When placed between fields or text elements, "/" causes positioning to the beginning of the next print line.

The contents of fields may be specified for input, output only, and output for modification using the attribute settings AD=A, AD=O, and AD=M respectively. The default is AD=A. All fields specified with AD=A (input only) or AD=M (output for modification) will create unprotected fields on the screen. A value for such a field

 Copyright © Software AG 2002

may be entered by the user.
For TTY devices, output for modification fields will occupy twice the size of the field (one for output, one for input) so that a new value may be entered. An input field (AD=A/M) specified as non-displayable will always start on a new line on a TTY device.

Example:

INPUT #A (AD=A) #B (AD=O) #C (AD=M)

#A is an input field which is unprotected, i.e., a value is to be entered for the field.

#B is a field which is to be displayed write-protected, i.e., no value may be entered for the field.

#C is a field whose current value is to be displayed, and the value may be modified by entering a new value for the field.

*IN, *OUT and *OUTIN

Equivalent to the attributes AD=A, AD=O, AD=M respectively.

Field Specification - operand1

Operand1 represents the field to be used. Database fields or user-defined variables may be specified.

Natural directly maps the content of each field from the data area to the INPUT statement, no move operation is necessary.

When the content of a database field is modified as a result of INPUT processing, only the value as contained in the data area is modified. Appropriate database UPDATE/STORE statements must be used to change the content of the database.

When the name of a group of database fields is referenced in an INPUT statement, all fields belonging to that group will be individually used as input fields.

When reference is made to a range of occurrences within an array, all occurrences are individually processed as input fields, but no prompting text will be created for each individual occurrence, only for the first one.

parameters

One or more parameters, enclosed within parentheses, may be specified immediately after operand1.

Each parameter specified in this manner will override any previous parameter specified in a GLOBALS command, SET GLOBALS or FORMAT statement. If more than one parameter is specified, one or more blanks must be present between each entry. An entry may not be split between two statement lines.

Note:
The edit mask parameter EM will be referenced dynamically in the DDM if an edit mask is defined for a database field. Edit masks may be specified for output and input fields. When an edit mask is defined for an input field, the data for the field must be entered according to the edit mask specification.

Example 1 - Syntax 1

```
/* EXAMPLE 'IPTEX1': INPUT
/*****************************************************
DEFINE DATA LOCAL
1 #PNUM (A8)
1 #FNC  (A1)
END-DEFINE
/*****************************************************
INPUT 10X 'SELECTION MENU FOR EMPLOYEES SYSTEM' /
      10X '-' (54) //
      10X 'ADD  '     '(A)' /
      10X 'PURGE'     '(P)' /
      10X 'UPDATE'    '(U)' /
      10X 'TERMINATE' '(A)' ///
      10X 'PLEASE ENTER FUNCTION: ' #FNC
/*****************************************************
DECIDE ON EVERY VALUE OF #FNC
  VALUE 'A'
    WRITE 'Add function selected'
    /* invoke the object containing the Add function here
  VALUE 'P'
    WRITE 'Purge function selected'
    /* invoke the object containing the Purge function here
  VALUE 'U'
    WRITE 'Update function selected'
    /* invoke the object containing the Update function here
  VALUE '.'
    STOP
  NONE
    REINPUT 'PLEASE ENTER A VALID FUNCTION' MARK *#FNC
END-DECIDE
/*****************************************************
END
```

```
          SELECTION MENU FOR EMPLOYEES SYSTEM
          -------------------------------------------------------

          ADD   (A)
          PURGE (P)
          UPDATE (U)
          TERMINATE (.)


          PLEASE ENTER FUNCTION:
```

Example 2 - Syntax 1

```
/* EXAMPLE 'INPEX1': INPUT WINDOW
*
DEFINE WINDOW WIND1
  SIZE 10 * 40
  BASE 5 / 10
  FRAMED ON POSITION TEXT
*
INPUT WINDOW='WIND1' 'PLEASE ENTER HERE:'
          / #STRING(A15)
*
END
```

 Copyright © Software AG 2002

```
> r                                     > +  Program     INPEX1   Lib SYSEXRM
  All    ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
    0010 /* EXAMPLE 'INPEX1': INPUT WINDOW
    0020 *
    0030 D +---------------------------------Top+
    0040   ! PLEASE ENTER HERE:                  !
    0050   ! #STRING                             !
    0060   !                                     !
    0070 * !                                     !
    0080 I !                                     !
    0090   !                                     !
    0100 * !                                     !
    0110 E !                                     !
    0120   +------------------------------Bottom+
    0130
    0140
    0150
    0160
    0170
    0180
    0190
    0200
         ....+....1....+....2....+....3....+....4....+....5....+... S 11   L 1
```

Example 3 - Syntax 1

```
/* EXAMPLE 'INPEX2': INPUT WINDOW
*
ASSIGN #START (A30) = 'EXAM_'
*
INPUT (AD=M) MARK POSITION 5 IN *#START
     / 'PLEASE COMPLETE START VALUE FOR SEARCH'
     / 5X #START
END
```

```
PLEASE COMPLETE START VALUE FOR SEARCH
       #START EXAM[]
```

Syntax 2 - Using Predefined Map Layout

```
INPUT [WINDOW = 'window -name'] [WITH-TEXT-option]
      [MARK-option]
      [ALARM-option]
      [USING] MAP map-name [NO ERASE]
      [ operand1 ...
      [ NO PARAMETER ]
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Map-name	C	S				A												yes	no
Operand1	C	S	A			A	N	P	I	F	B	D	T	L	C			yes	yes

This form of the INPUT statement is used to perform input processing using a map layout that has been created using the Natural map editor.

Map layouts can be used in two ways:

- the program does not provide a parameter list;
- the program does provide a parameter list (*operand1*).

INPUT USING MAP Without Parameter List

In this case the *map-name* must be specified as an alphanumeric constant (up to 8 characters).

The map used in this manner must have been created prior to the compilation of the program which references the map.

The names of the fields to be processed are taken dynamically from the map source definition at compilation time. The field names used in both program and map must be identical.

All fields to be referenced in the INPUT statement must be accessible at that point.

In structured mode, fields must have been previously defined (database fields must be properly referenced to processing loops or views). In reporting mode, user-defined variables may be newly defined in the map.

When the map layout is changed, the programs using the map need not be recataloged. However, when array structures or names, formats/lengths of fields are changed, or fields are added/deleted in the map, the programs using the map must be recataloged.

The map source must be available at program compilation; otherwise the INPUT USING MAP statement cannot be compiled. If you wish to compile the program even if the map is not yet available, specify NO PARAMETER: the INPUT USING MAP can then be compiled even if the map is not yet available.

 Copyright © Software AG 2002

INPUT Fields Defined in the Program

By specifying the names of the fields to be processed within the program (*operand1*), it is possible to have the names of the fields in the program differ from the names of the fields in the map.

The sequence of fields in the program must match the map sequence. Please note that the map editor sorts the fields as specified in the map in alphabetical order by field name. For more information, see the map editor description in your Natural User's Guide.

The program editor line command ".I(*mapname*)" can be used to obtain a complete INPUT USING MAP statement with a parameter list derived from the fields defined in the specified map.

When the layout of the map is changed, the program using the map need not be recataloged. However, when field names, field formats/lengths, or array structures in the map are changed or fields are added or deleted in the map, the program must be recataloged.

A check is made at execution time to ensure that the format and length of the fields as specified in the program match the fields as specified in the map. If both layouts do not agree, an error message is produced.

INPUT WINDOW='window-name'

This option is described under **Syntax 1** of the INPUT statement.

WITH TEXT/MARK/ALARM-options

These options are described under **Syntax 1** of the INPUT statement.

USING MAP

USING MAP invokes a map definition which has been previously stored in a Natural system file using the map editor.

The *map-name* may be a 1- to 8-character alphanumeric constant or user-defined variable. If a variable is used, it must have been previously defined. The map name may contain an ampersand (&); at execution time, this character will be replaced by the current value of the Natural system variable *LANGUAGE. This feature allows the use of multi-lingual maps.

The execution of the INPUT statement causes the corresponding map to replace the current contents of the screen, unless the NO ERASE option is specified, in which case the map will overlay the current contents of the screen.

NO ERASE

This option is described under **Syntax 1** of the INPUT statement.

Field Specification - operand1

A list of database fields and/or user-defined variables, all of which must have been previously defined. The fields must agree in number, sequence, format and length with the fields in the referenced map; otherwise, an error occurs.

When the content of a database field is modified as a result of INPUT processing, only the value as contained in the data area is modified. Appropriate database UPDATE/STORE statements must be used to change the content of the database.

Using the INPUT Statement in Non-Screen Modes

Forms Mode

The terminal command %F causes forms mode to be in effect.

In forms mode, Natural will display all output text of the map layout on the terminal field by field according to the positioning parameters. This permits the user to enter data on a field by field basis. When all data are entered, the hardcopy output is produced exactly as it would have appeared on the screen.

In forms mode, entering %R permits the operator to retype the entire form in case of an error. The input is processed as in the first execution of the INPUT statement.

Keyword/Delimiter Mode

The terminal command %D causes keyword/delimiter mode to be in effect.

Data can be entered using keywords or positional input values.

Using keyword input, the terminal operator may enter data for the individual fields using the prompting text that, in forms mode, would have been displayed before the value as a keyword to identify the field. The keyword must be followed by the input assign character (IA parameter), followed immediately by the data. Any spaces following the assign character are taken as data up to the delimiter character (ID parameter). A delimiter character is not required after the last data element. Keyword data for the different fields may be entered in any order separated by the delimiter character. If the operator types in a keyword which is not defined in the INPUT statement, an error message will be returned. Data need not be entered for all input fields. Fields for which no data are entered are set to blank for alphanumeric fields and zero for numeric and hexadecimal fields.

Using positional value input, the terminal operator enters only data for all input fields separated by the currently defined input delimiter character (ID parameter). The sequence of fields for input must correspond to the sequence of the fields in the INPUT statement.

The user may switch from positional to keyword input by entering a number of values in positional input separated by the delimiter character and then switching to keyword mode for selected fields by specifying keywords in front of the values.

After a keyword has been used to position to a field, any non-keyword input following the keyword will be processed as positional input to be assigned to fields following the previously selected field in the INPUT statement.

Note:
A keyword and the corresponding input field must be on the same logical line. If their aggregate length exceeds the line size, adjust the line size (LS parameter) accordingly so that keyword and field fit onto one line.

Data entered in keyword/delimiter mode are validated as for screen mode. An error message will be returned if an attempt is made to enter more characters than defined for a field.

If the INPUT statement is to be processed in keyword/delimiter mode on a buffered (3270-type) terminal or a workstation, all data to be assigned to one INPUT statement must be entered on one screen. ENTER is only to be used when all data to the INPUT statement have been entered.

Processing Data from the Natural Stack

Data elements that have been placed in the Natural stack via a FETCH, RUN or STACK statement will be processed by the next INPUT statement encountered for execution.

 Copyright © Software AG 2002

The INPUT statement will process the data in keyword/delimiter mode as described above.

If data elements are not available to fill all input fields, fields will be filled with blank/zero depending on the field format. If more data elements are specified than input fields exist, the remaining data are ignored.

When a field is filled with data from the stack, the field attributes do not apply to the data.

The Natural system variable *DATA may be referenced to determine the number of data elements currently available in the Natural stack.

Using the INPUT Statement in Batch Mode on Mainframe Computers

Forms Mode

In batch forms mode, the INPUT map is displayed. A data record is read for each line containing one or more AD=A and/or AD=M fields, and the data contained in the record are assigned to the appropriate field (or fields).

Input data fields are assumed to be contiguous. Unless the delimiter character is used, input data must be entered in the exact length according to the internal definition of the field. For numeric fields, space must be allowed for a sign (if SG=ON) and decimal point when appropriate.

Data may optionally be entered using the delimiter character to separate the values of the individual fields. In this case, data need not be entered in the exact number of positions according to the internal definition but are processed from left to right beginning in position 1. The rules for data entry are the same as described under Entering Data in Response to an INPUT Statement. In addition, the assign character may be used to specify that the contents of an *OUTIN field are not to be reset.

Keyword/Delimiter Mode

Keyword/delimiter mode, when used in batch mode, functions the same as keyword/delimiter mode in TP mode with the following exceptions:

- The entire input map may be printed under the control of the terminal command %Q.
- *OUTIN fields retain their original values unless explicitly changed.

Use of Terminal Commands in Batch Mode

The following Natural terminal commands may be used when using the INPUT statement in batch mode on a mainframe computer:

Command	Explanation
%*	Record Suppression. When entered in position one and two of a record, %* causes the printing of the next input record to be suppressed. DATA RECORD %* SUPPRESSED DATA RECORD
%	Record Continuation. When % is entered as the last non-blank character of a record, the next input record will be treated as a continuation record. DATA, RECORD, WITH, CONTINUATION, % CONTINUATION RECORD INPUT V1 V2 V3 V4 V5 V6 DISPLAY V1 V2 V3 V4 V5 V6 will produce the following output: DATA RECORD WITH CONTINUATION CONTINUATION RECORD
%/	End-of-file. When entered in the first two positions of a record (without any trailing non-blank characters), %/ causes an end-of-file condition.
%%	Set restart point in input data stream.
%.	Reading of input values for the current INPUT statement will be terminated.
%K*nn*	Simulate PF keys.
%KP*n*	Simulate PA keys.
%Q	This command causes printing of maps used to read input data to be suppressed.

See the Natural Reference documentation for further information on terminal commands.

Additional JCL statements are required when using the INPUT statement for data entry in batch mode. The Natural administrator should be contacted to ensure that these statements have been provided before attempting to execute Natural in batch mode.

 Copyright © Software AG 2002

INTERFACE

```
INTERFACE        interface-name

                 [EXTERNAL]
                 [ID interface-GUID]
                 [property-definition]
                 [method-definition]

END-INTERFACE
```

Function

An interface is a collection of methods and properties that belong together semantically and represent a certain feature of a class.

You can define one or several interfaces for a class. Defining several interfaces allows you to structure/group methods according to what they do, e.g., you put all methods that deal with persistency (load, store, update) in one interface and put other methods in other interfaces.

The INTERFACE statement is used to define an interface. It may only be used in a Natural class module and can be defined as follows:

- within a DEFINE CLASS statement. This form is used when the interface is only to be implemented in one class, or
- in a copycode which is included by the INTERFACE USING clause of the DEFINE CLASS statement. This form is used when the interface is to be implemented in more than one class.

The properties and methods that are associated with the interface are defined by the property and method definitions.

interface-name

This is the name to be assigned to the interface. The interface name can be up to a maximum of 32 characters long and must conform to the Natural naming conventions for user variables (please refer to the Natural Reference documentation for further information). It must be unique per class and different from the class name.

If the interface is planned to be used by clients written in different programming languages, the interface name should be chosen in a way that it does not conflict with the naming conventions that apply in these languages. Bolero for example uses the Java naming convention. So, an interface that is planned to be used in a Bolero client should also respect the Java naming conventions.

Copyright © Software AG 2002

EXTERNAL

Note:
This clause is only available with Natural Version 5.1.1 for UNIX/OpenVMS and Windows or a higher version.

The EXTERNAL clause is used to indicate that this interface is implemented by the class, but which is originally defined in a different class. The clause is only relevant if the class is to be registered with DCOM. Interfaces with the EXTERNAL clause are ignored when the class is registered with DCOM. It is assumed that the interface is registered by the class that originally defines it.

ID Clause

The ID clause is used to assign a globally unique ID to the interface. The interface GUID is the name of a GUID defined in a data area that is included by the LOCAL clause. The interface GUID is a (named) alpha constant. A GUID must be assigned to an interface if the class is to be registered with DCOM.

property-definition

```
PROPERTY        property-name

                [(format-length/array-definition)]
                [ID dispatch-ID]
                [READONLY]
                [IS operand]

END-PROPERTY
```

The property definition is used to define a property of the interface.

Properties are attributes of an object that can be accessed by clients. An object that represents an employee might for example have a ’Name’ property and a ’Department’ property. Retrieving or changing the name or department of the employee by accessing her ’Name’ or ’Department’ property is much simpler for a client than calling one method that returns the value and another method that changes the value.

Each property needs a variable in the object data area of the class to store its value - this is referred to as the object data variable. The property definition is used to make this variable accessible to clients. The property definition defines the name and format of the property and connects it to the object data variable. In the simplest case, the property takes the name and format of the object data variable itself. It is also possible to override the name and format within certain limits.

property-name

This is the name to be assigned to the property. The property name can contain up to a maximum of 32 characters and must conform to the Natural naming conventions for user variables (please refer to the Natural Reference documentation for further information).

 Copyright © Software AG 2002

If the property is planned to be used by clients written in different programming languages, the property name should be chosen in a way that it does not conflict with the naming conventions that apply in these languages. Bolero for example uses the Java naming convention. So, a property that is planned to be used in a Bolero client should also respect the Java naming conventions.

format-length/array-definition

This defines the format of the property as it will be seen by clients.

If *format-length/array-definition* is omitted, the format-length and array-definition will be taken from the object data variable assigned in the IS clause.

If *format-length/array-definition* is specified, it must be data transfer-compatible both to and from the format of the object data variable specified in *operand* in the IS clause. In the case of a READONLY property, the data transfer-compatibility needs to hold only in one direction: with the object data variable as source operand and the property as destination operand. If an array-definition is specified, it must be equal in dimensions, occurrences per dimension, lower bounds and upper bounds to the array definition of the corresponding object data variable. This is expressed by specifying an asterisk for each dimension.

ID Clause

Note:
This clause is only available with Natural Version 5.1.1 for UNIX/OpenVMS and Windows or a higher version.

The ID clause is used to assign a specific numeric identifier to a property. This identifier (the so-called dispatch ID) is only relevant if the class is to be registered with DCOM.

Normally, Natural automatically assigns a dispatch ID to a property. It is only necessary to explicitly define a specific dispatch ID for a property if the property belongs to an interface with the EXTERNAL clause. (This is an interface that shall be implemented in this class, but which is originally defined in a different class.) In this case the dispatch IDs to be used are usually dictated by the original implementation of the interface.

The dispatch ID is a positive, non-zero constant of format I4.

READONLY

If this keyword is specified, the value of the property can only be read and not set. The format of the object data variable specified in *operand* in the IS clause must be data transfer-compatible to the format specified in *format-length/array-definition*. It does not have to be data transfer-compatible in the inverse direction.

If the keyword READONLY is omitted, the property value can be both read and set.

IS Clause

The *operand* in the IS clause assigns an object data variable as the place to store the property value. The assigned object data variable may not be a group. The variable is referenced in normal operand syntax. This means that if the object data variable is an array, it must be referenced with index notation. Only the full index range notation and asterisk notation is allowed.

The IS clause should not be used if the INTERFACE statement will be included from a copycode member and reused in several classes. If you want to reuse the INTERFACE statement, you must assign the object data variable in a PROPERTY statement outside the INTERFACE statement.

If the IS clause is omitted, the property is connected to the object data variable with the same name as the property. If a variable with this name is not defined or if it is a group, a syntax error results.

Copyright © Software AG 2002

Examples

Let the object data area contain the following data definitions:

```
1 Salary(p7.2)
  1 SalaryHistory(p7.2/1:10)
```

Then the following property definitions are allowed:

```
property Salary
  end-property
  property Pay is Salary
  end-property
  property Pay(P7.2) is Salary
  end-property
  property Pay(N7.2) is Salary
  end-property
  property SalaryHistory
  end-property
  property OldPay is SalaryHistory(*)
  end-property
  property OldPay is SalaryHistory(1:10)
  end-property
  property OldPay(P7.2/*) is SalaryHistory(1:10)
  end-property
  property OldPay(N7.2/*) is SalaryHistory(*)
  end-property
```

 Copyright © Software AG 2002

The following property definitions are not allowed:

```
/* Not data transfer-compatible. */
  property Pay(L) is Salary
  end-property
  /* Not data transfer-compatible. */
  property OldPay(L/*) is SalaryHistory(*)
  end-property
  /* Not data transfer-compatible. */
  property OldPay(L/1:10) is SalaryHistory(1:10)
  end-property
  /* Assigns an array to a scalar. */
  property OldPay(P7.2) is SalaryHistory(1:10)
  end-property
  /* Takes only a sub-array. */
  property OldPay(P7.2/3:5) is SalaryHistory(*)
  end-property
  /* Index specification omitted in ODA variable SalaryHistory. */
  property OldPay is SalaryHistory
  end-property
  /* Only asterisk notation allowed in property format specification. */
  property OldPay(P7.2/1:10) is SalaryHistory(*)
  end-property
```

method-definition

```
METHOD          method-name

                [ID dispatch-ID]
                [IS subprogram-name]
                [PARAMETER {USING {local-data-area}
                                  {parameter-data-area}}
                           {      data-definition...  }]

END-METHOD
```

```
METHOD        method - name

              [ ID dispatch-ID          ]
              [ IS subprogram-name      ]
              [                                                  ]
              [ PARAMETER   { USING { local-data-area      } }   ]
              [             {       { parameter-data-area  } }   ]...
              [             {       data-definition...     }   ]

END-METHOD
```

The method definition is used to define a method for the interface.

To make the interface reusable in different classes, include the interface definition from a copycode and define the subprogram after the interface definition with a METHOD statement. Then you can implement the method differently in different classes.

 Copyright © Software AG 2002

method-name

This is the name to be assigned to the method. The method name can contain a maximum of up to 32 characters and must conform to the Natural naming conventions for user variables (please refer to the Natural Reference documentation for further information). It must be unique per interface.

If the method is planned to be used by clients written in different programming languages, the method name should be chosen in a way that it does not conflict with the naming conventions that apply in these languages. Bolero for example uses the Java naming convention. So, a method that is planned to be used in a Bolero client should also respect the Java naming conventions.

ID Clause

Note:
This clause is only available with Natural Version 5.1.1 for UNIX/OpenVMS and Windows or a higher version.

The ID clause is used to assign a specific numeric identifier to a method. This identifier (the so-called dispatch ID) is only relevant if the class is to be registered with DCOM.

Normally, Natural automatically assigns a dispatch ID to a method. It is only necessary to explicitly define a specific dispatch ID for a property if the property belongs to an interface with the EXTERNAL clause. (This is an interface that shall be implemented in this class, but which is originally defined in a different class.) In this case, the dispatch IDs to be used are usually dictated by the original implementation of the interface.

The dispatch ID is a positive, non-zero constant of format I4.

IS Clause

This is the name of the subprogram that implements the method. The name of the subprogram consists of up to 8 characters. The default is method-name (if the IS clause is not specified).

PARAMETER Clause

This specifies the parameters of the method, and has the same syntax as the PARAMETER clause of the DEFINE DATA statement. For further information on the DEFINE DATA statement, see the Natural Statements documentation.

The parameters must match the parameters which are later used in the implementation of the subprogram. This is ensured best by using a parameter data area.

Parameters that are marked ’BY VALUE’ in the parameter data area are input parameters of the method.

Parameters which 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’ are available with Version 4.1.2 and all subsequent releases. Optional parameters need not to be specified when the method is called. They can be left unspecified by using the nX notation in the SEND METHOD statement.

Copyright © Software AG 2002

LIMIT

```
LIMIT n
```

Function

The LIMIT statement is used to limit the number of iterations of a processing loop initiated with a FIND, READ, or HISTOGRAM statement.

The limit remains in effect for all subsequent processing loops in the program until it is overridden with another LIMIT statement. The LIMIT statement does not apply to individual statements in which a limit is explicitly specified (for example, FIND (*n*) ...).

If the limit is reached, processing stops and a message is displayed (see also the session parameter LE in the Natural Reference documentation).

If no LIMIT statement is specified, the default limit defined during Natural installation will be used.

Limit Specification - *n*

The limit **n** must be specified as a numeric constant in the range from 0 to 99999999 (leading zeros are optional). The processing loop is not entered if the limit is set to "0".

Record Counting

To determine whether a processing loop has reached the limit, each record read in the loop is counted against the limit. If the processing loop has reached the limit, the following will apply:

- A record that is rejected because of criteria specified in a FIND or READ statement WHERE clause is `not` counted against the limit.
- A record that is rejected as a result of an ACCEPT/REJECT statement `is` counted against the limit.

 Copyright © Software AG 2002

Example 1

```
/* EXAMPLE 'LMTEX1': LIMIT
/*****************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 CITY
END-DEFINE
/*****************************************************
LIMIT 4
/*****************************************************
READ EMPLOY-VIEW BY NAME STARTING FROM 'BAKER'
  DISPLAY NOTITLE NAME PERSONNEL-ID CITY *COUNTER
END-READ
/*****************************************************
END
```

```
      NAME         PERSONNEL         CITY           CNT
                        ID
-------------------- --------- -------------------- ---------

BAKER                 20016700  OAK BROOK                   1
BAKER                 30008042  DERBY                       2
BALBIN                60000110  BARCELONA                   3
BALL                  30021845  DERBY                       4
```

Example 2

```
/* EXAMPLE 'LMTEX2': LIMIT
/*****************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
END-DEFINE
/*****************************************************
LIMIT 3
/*****************************************************
FIND EMPLOY-VIEW WITH NAME > 'A'
  READ EMPLOY-VIEW BY NAME STARTING FROM 'BAKER'
    DISPLAY NOTITLE 'CNT(0100)' *COUNTER(0100)
                    'CNT(0110)' *COUNTER(0110)
  END-READ
END-FIND
/*****************************************************
END
```

```
CNT(0100) CNT(0110)
  --------- ---------

          1         1
          1         2
          1         3
          2         1
          2         2
          2         3
          3         1
          3         2
          3         3
```

 Copyright © Software AG 2002

LOOP

Note: This statement may be used in reporting mode only; it is not permitted in structured mode.

```
[CLOSE] LOOP [(r)]
```

Function

The LOOP statement is used to close a processing loop. It causes processing of the current pass through the loop to be terminated and control to be returned to the beginning of the processing loop.

When the processing loop for which the LOOP statement is issued is terminated (that is, when all records have been processed or iterations have been performed), execution continues with the statement after the LOOP statement.

Statement Reference Notation - *r*

The LOOP statement may be specified with a statement label or reference number, in which case all inner loops up to and including the loop initiated by the statement referenced will be closed. If no statement reference is specified, the innermost active processing loop will be close.

Database Variable References

A LOOP statement, in addition to closing a processing loop, will eliminate all field references to FIND, FIND FIRST, FIND UNIQUE, READ and GET statements contained within the loop.

A field within a view may be referenced outside the processing loop by using the view name as a qualifier.

Restrictions

A LOOP statement may not be specified based on a conditional statement such as IF or AT BREAK.

Example 1

```
0010 FIND ...
0020   READ ...
0030     READ ...
0040 LOOP (0010)    /* closes all loops
```

Example 2

```
0010 FIND ...
0020   READ ...
0030     READ ...
0040     LOOP       /* closes loop initiated on line 0030
0050   LOOP         /* closes loop initiated on line 0020
0060 LOOP           /* closes loop initiated on line 0010
```

 Copyright © Software AG 2002

METHOD

```
METHOD method–name
            OF [INTERFACE ] interface–name
            IS subprogram–name
END–METHOD
```

Function

The METHOD statement assigns a subprogram as the implementation to a method, outside an interface definition. It is used if the interface definition in question is included from a copycode and is to be implemented in a class-specific way.

The METHOD statement may only be used within the DEFINE CLASS statement and after the interface definition. The interface and method names specified must be defined in the interface definitions.

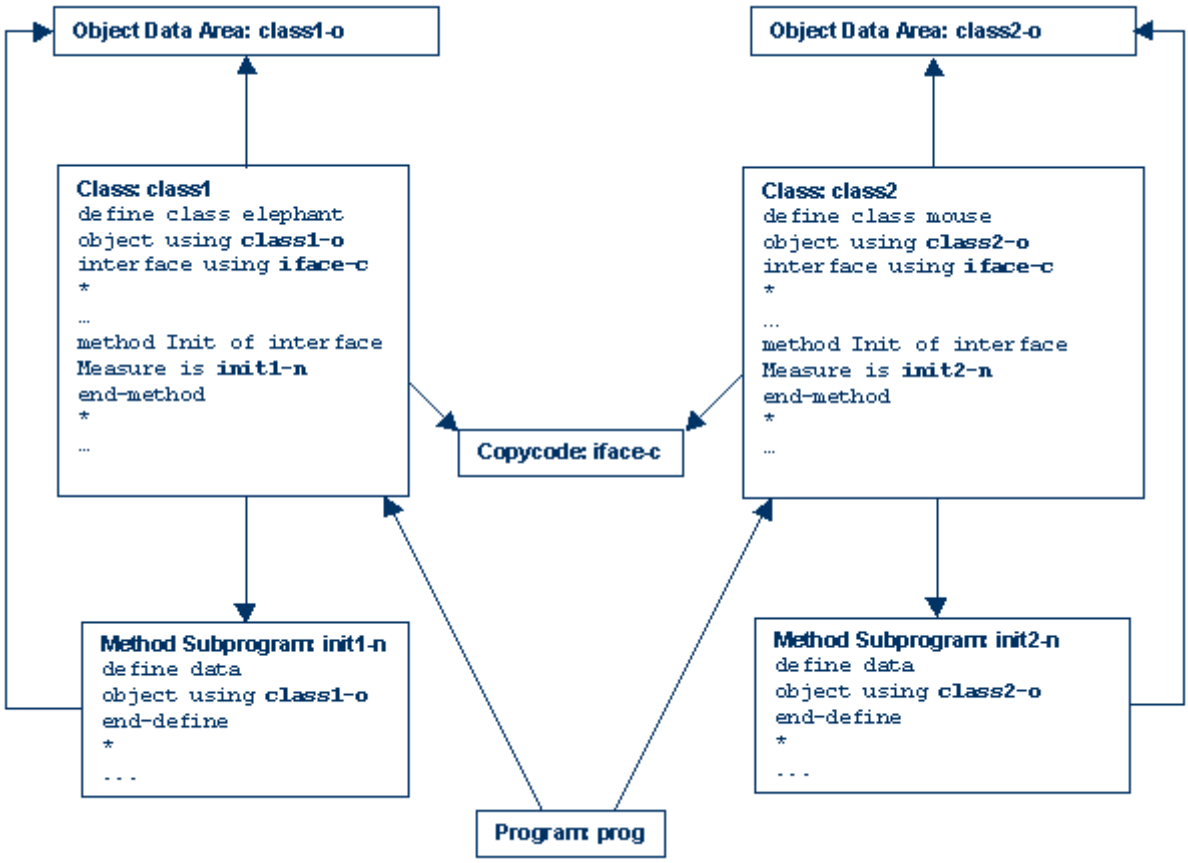
Example

The following example shows how the same interface is implemented differently in two classes and how the PROPERTY statement and the METHOD statement are used to achieve this.

The interface *Measure* is defined in the Copycode *iface-c*. The Classes *Elephant* and *Mouse* implement both the interface *Measure*. Therefore, they both include the Copycode *iface-c*. But the Classes implement the property *Height* using different variables from their respective object data areas, and they implement the method *Init* with different subprograms. They use the PROPERTY statement to assign the selected data area variable to the property and the METHOD statement to assign the selected subprogram to the method.

Now the program *prog* can create objects of both classes and initialize them using the same method *Init*, leaving the specifics of the initialization to the respective Class implementation.

Copyright © Software AG 2002



The following shows the complete contents of the Natural modules used in the example above:

Copycode: iface-c

```
interface Measure
  *
  property Height(p5.2)
  end-property
  *
  property Weight(i4)
  end-property
  *
  method Init
  end-method
  *
  end-interface
```

 Copyright © Software AG 2002

Class: class1

```
define class elephant
  object using class1-o
  interface using iface-c
  *
  property Height of interface Measure is height
  end-property
  *
  property Weight of interface Measure is weight
  end-property
  *
  method Init of interface Measure is init1-n
  end-method
  *
  end-class
  end
```

Object Data Area: class1-o

```
*   *** Top of Data Area ***
    1 HEIGHT                  P 5.2
    1 WEIGHT                  I 2
  *   *** End of Data Area ***
```

Method Subprogram: init1-n

```
define data
  object using class1-o
  end-define
  *
  height := 17.3
  weight := 120
  *
  end
```

Class: class2

```
define class mouse
  object using class2-o
  interface using iface-c
  *
  property Height of interface Measure is size
  end-property
  *
  property Weight of interface Measure is weight
  end-property
  *
  method Init of interface Measure is init2-n
  end-method
  *
  end-class
  end
```

Object Data Area: class2-o

```
*   *** Top of Data Area ***
    1 SIZE                    P 3.2
    1 WEIGHT                  I 1
  *   *** End of Data Area ***
```

Method Subprogram: init2-n

```
define data
  object using class2-o
  end-define
  *
  size := 1.24
  weight := 2
  *  end
```

Program: prog

```
define data local
  1 #o handle of object
  1 #height(p5.2)
  1 #weight(i4)
  end-define
  *
  create object #o of class "Elephant"
  send "Init" to #o
  #height := #o.Height
  #weight := #o.Weight
  write #height #weight
  *
  create object #o of class "Mouse"
  send "Init" to #o
  #height := #o.Height
  #weight := #o.Weight
  write #height #weight
  *
  end
```

 Copyright © Software AG 2002

MOVE

```
MOVE [ROUNDED] operand1 [(parameter)] TO operand2 ...
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S	A		N	A	N	P	I	F	B	D	T	L	C	G	O	yes	no
Operand2		S	A		M	A	N	P	I	F	B	D	T	L	C	G	O	yes	yes

```
MOVE { operand1                                  } [(parameter)] TO
     { SUBSTRING (operand1,operand3,operand4)    }
                                                 { operand2                                  }
                                                 { SUBSTRING (operand2,operand5,operand6)    } ...
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S	A			A	N*											yes	no
Operand2		S	A			A												yes	no
Operand3	C	S					N	P	I									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

* see text.

```
MOVE BY { [NAME]  } operand1 TO operand2
        { POSITION }
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1				G														yes	no
Operand2				G														yes	no

```
MOVE EDITED operand1 TO operand2 (EM = value)
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S	A			A					B							yes	no
Operand2		S	A			A	N	P	I	F	B	D	T	L				yes	yes

```
MOVE EDITED operand1 (EM = value) TO operand2
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S	A		N	A	N	P	I	F	B	D	T	L				yes	no
Operand2		S	A			A					B							yes	yes

```
MOVE { LEFT } [JUSTIFIED] operand1 [(parameter)] TO operand2
     { RIGHT }
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S	A		N	A	N	P	I	F	B	D	T	L				yes	no
Operand2		S	A			A												yes	yes

Related Statement: COMPUTE

Function

The MOVE statement is used to move the value of an operand to one or more operands (field or array).

If *operand2* is a DYNAMIC variable, its length may be modified by the MOVE operation. The current length of a DYNAMIC variable can be ascertained by using the system variable *LENGTH. For general information on the DYNAMIC variable, see your Natural User's Guide.

 Copyright © Software AG 2002

ROUNDED

This option causes *operand2* to be rounded.

ROUNDED is ignored if *operand2* is not numeric.

If *operand2* is of format N or P and *operand2* is specified more than once, ROUNDED is ignored for target operands with seven positions after the decimal point.

parameter

As *parameter*, you can specify the option "PM=I" or the session parameter DF:

PM=I

In order to support languages whose writing direction is from right to left, you can specify "PM=I" so as to transfer the value of *operand1* in inverse (right-to-left) direction to *operand2*.

For example, as a result of the following statements, the content of #B would be "ZYX":

```
MOVE 'XYZ' TO #A
MOVE #A (PM=I) TO #B
```

PM=I can only be specified if *operand2* has alphanumeric format.

Any trailing blanks in *operand1* will be removed, then the value is reversed and moved to *operand2*. If *operand1* is not of alphanumeric format, the value will be converted to alphanumeric format before it is reversed.

See also the use of PM=I in conjunction with MOVE LEFT/RIGHT JUSTIFIED.

DF

If *operand1* is a date variable and *operand2* is an alphanumeric field, you can specify the session parameter DF as parameter for this date variable. The session parameter DF is described in the Natural Reference documentation.

SUBSTRING

Without the SUBSTRING option, the whole content of a field is moved. The SUBSTRING option allows you to move only a certain part of an alphanumeric field. After the field name (*operand1*) in the SUBSTRING clause you specify first the starting position (*operand3*) and then the length (*operand4*) of the field portion to be moved.

For example, to move the 5th to 12th position inclusive of the value in a field #A into a field #B, you would specify:

```
MOVE SUBSTRING(#A,5,8) TO #B
```

If *operand1* is a DYNAMIC variable, the specified field portion to be moved must be within its current length; otherwise, a runtime error will occur.

Copyright © Software AG 2002

Also, you can move a value of an alphanumeric or numeric field into a certain part of the target field. After the field name (*operand2*) in the SUBSTRING clause you specify first the starting position (*operand5*) and then the length (*operand6*) of the field portion into which the value is to be moved.

For example, to move the value of a field #A into the 3rd to 6th position inclusive of a field #B, you would specify:

```
MOVE #A TO SUBSTRING(#B,3,4)
```

If *operand2* is a DYNAMIC variable, the specified starting position (*operand5*) must not be greater than the variable's current length plus 1; a greater starting position will lead to a runtime error, because it would cause an undefined gap within the content of *operand2*.

If you omit *operand3/5*, the starting position is assumed to be "1". If you omit *operand4/6*, the length is assumed to be from the starting position to the end of the field.

If *operand2* is a DYNAMIC variable and the specified starting position (*operand5*) is the variable's current length plus 1, which means that the MOVE operation is used to increase the length of the variable, *operand6* must be specified in order to determine the new length of the variable.

Note:
MOVE with the SUBSTRING option is a byte-by-byte move (that is, the rules described under Rules for Arithmetic Assignment in the Natural Reference documentation do not apply).

MOVE BY NAME

This option is used to move individual fields contained in a data structure to another data structure, independent of their position in the structure. A field is moved only if its name appears in both structures (this includes REDEFINEd fields as well as fields resulting from a redefinition). The individual fields may be of any format. The operands can also be views.

Note:
The sequence of the individual moves is determined by the sequence of the fields in operand1.

MOVE BY NAME with Arrays

If the data structures contain arrays, these will internally be assigned the index "(*)" when moved; this may lead to an error if the arrays do not comply with the rules for assignment operations with arrays (see the section Processing of Arrays in the Natural Reference documentation).

 Copyright © Software AG 2002

Example 1 of MOVE BY NAME with Arrays:

```
DEFINE DATA LOCAL
  1 #GROUP1
    2 #FIELD (A10/1:10)
  1 #GROUP2
    2 #FIELD (A10/1:10)
END-DEFINE
...
MOVE BY NAME #GROUP1 TO #GROUP2
...
```

In this, example, the MOVE statement would internally be resolved as:

```
MOVE #GROUP1.#FIELD (*) TO #GROUP2.#FIELD (*)
```

If part of an indexed group is moved to another part of the same group, this may lead to unexpected results as shown in the example below.

Example 2 of MOVE BY NAME with Arrays:

```
DEFINE DATA LOCAL
  1 #GROUP1 (1:5)
    2 #FIELDA (N1) INIT <1,2,3,4,5>
    2 REDEFINE #FIELDA
      3 #FIELDB (N1)
END-DEFINE
...
MOVE BY NAME #GROUP1 (2:4) TO #GROUP1 (1:3)
...
```

In this, example, the MOVE statement would internally be resolved as:

```
MOVE #FIELDA (2:4) TO #FIELDA (1:3)
MOVE #FIELDB (2:4) TO #FIELDB (1:3)
```

First, the contents of the occurrences 2 to 4 of #FIELDA are moved to the occurrences 1 to 3 of #FIELDA; that is, the occurrences receive the following values:

Occurrence:	1.	2.	3.	4.	5.
Value before:	1	**2**	**3**	**4**	5
Value after:	**2**	**3**	**4**	4	5

Then the contents of the occurrences 2 to 4 of #FIELDB are moved to the occurrences 1 to 3 of #FIELDB; that is, the occurrences receive the following values:

Occurrence:	1.	2.	3.	4.	5.
Value before:	2	**3**	**4**	**4**	5
Value after:	**3**	**4**	**4**	4	5

MOVE BY POSITION

This option allows you to move the contents of fields in a group to another group, regardless of the field names. The values are moved field by field from one group to the other in the order in which the fields are defined (this does not include fields resulting from a redefinition). The individual fields may be of any format. The number of fields in each group must be the same; also, the level structure and array dimensions of the fields must match. Format conversion is done according to the rules for arithmetic assignment described in the Natural Reference documentation. The operands can also be views.

Example of MOVE BY POSITION:

```
DEFINE DATA LOCAL
  1 #GROUP1
    2 #FIELD1A (N5)
    2 #FIELD1B (A3/1:3)
    2 REDEFINE #FIELD1B
      3 #FIELD1BR (A9)
  1 #GROUP2
    2 #FIELD2A (N5)
    2 #FIELD2B (A3/1:3)
    2 REDEFINE #FIELD2B
      3 #FIELD2BR (A9)
END-DEFINE
...
MOVE BY POSITION #GROUP1 TO #GROUP2
...
```

In this example, the content of #FIELD1A is moved to #FIELD2A, and the content of #FIELD1B to #FIELD2B; the fields #FIELD1BR and #FIELD2BR are not affected.

MOVE EDITED

An edit mask may be specified with *operand1* or *operand2*.

If an edit mask is specified for *operand2*, the value of *operand1* will be placed into *operand2* using this edit mask.

If an edit mask is specified for *operand1*, the edit mask will be applied to *operand1* and the result will be moved to *operand2*. The length of the *operand1* value after the edit mask has been applied to it must not exceed the length of *operand2*.

For details on edit masks, see the session parameter EM in the Natural Reference documentation.

MOVE LEFT/RIGHT JUSTIFIED

This option is used to cause the values to be moved to be left- or right-justified in operand2.

With MOVE LEFT JUSTIFIED, any leading blanks in *operand1* are removed before the value is placed left-justified into *operand2*. The remainder of *operand2* will then be filled with blanks. If the value is longer than *operand2*, the value will be truncated on the right-hand side.

With MOVE RIGHT JUSTIFIED, any trailing blanks in *operand1* are truncated before the value is placed right-justified into *operand2*. The remainder of *operand2* will then be filled with blanks. If the value is longer than *operand2*, the value will be truncated on the left-hand side.

 Copyright © Software AG 2002

MOVE LEFT/RIGHT JUSTIFIED cannot be used if *operand2* is a DYNAMIC variable.

MOVE LEFT/RIGHT JUSTIFIED with PM=I

When you use MOVE LEFT/RIGHT JUSTIFIED in conjunction with PM=I, the move is performed in the following steps:

1. If *operand1* is not of alphanumeric format, the value is converted to alphanumeric format.
2. Any trailing blanks in *operand1* are removed.
3. In the case of LEFT JUSTIFIED, any leading blanks in *operand1* are also removed.
4. The value is reversed, and then moved to *operand2*.
5. If applicable, the remainder of *operand2* is filled with blanks, or the value is truncated (see above).

Other Considerations

If a database field is used as the result field, the MOVE operation results in an update only to the internal value of the field as used within the program. The value of the field in the database remains unchanged.

A Natural system function may be used only if the MOVE statement is specified in conjunction with an AT BREAK, AT END OF DATA or AT END OF PAGE statement.

See also the section Rules for Arithmetic Assignment in the Natural Reference documentation.

Note:
If operand1 is a time variable (format T), only the time component of the variable content is transferred, but not the date component (except with MOVE EDITED).

Example 1

```
/* EXAMPLE 'MOVEX1': MOVE
/************************************************************
DEFINE DATA LOCAL
1 #A (N3)
1 #B (A5)
1 #C (A2)
1 #D (A7)
1 #E (N1.0)
1 #F (A5)
1 #G (N3.2)
1 #H (A6)
END-DEFINE
/************************************************************
MOVE 5 TO #A
WRITE NOTITLE 'MOVE 5 TO #A' 30X '=' #A
/************************************************************
MOVE 'ABCDE' TO #B #C #D
WRITE 'MOVE ABCDE TO #B #C #D' 20X '=' #B '=' #C '=' #D
/************************************************************
MOVE -1 TO #E
WRITE 'MOVE -1  TO #E' 28X '=' #E
/************************************************************
MOVE ROUNDED 1.995 TO #E
WRITE 'MOVE ROUNDED 1.995 TO #E' 18X '=' #E
/************************************************************
MOVE RIGHT JUSTIFIED 'ABC' TO #F
WRITE 'MOVE RIGHT JUSTIFIED ''ABC'' TO #F' 10X '=' #F
/************************************************************
MOVE EDITED '003.45' TO #G (EM=999.99)
WRITE 'MOVE EDITED ''003.45'' TO #G (EM=999.99)' 4X '=' #G
/************************************************************
MOVE EDITED 123.45 (EM=999.99) TO #H
WRITE 'MOVE EDITED 123.45 (EM=999.99) TO #H' 6X '=' #H
/************************************************************
END
```

```
MOVE 5 TO #A                              #A:    5
MOVE ABCDE TO #B #C #D                    #B: ABCDE #C: AB #D: ABCDE
MOVE -1  TO #E                            #E: -1
MOVE ROUNDED 1.995 TO #E                  #E:  2
MOVE RIGHT JUSTIFIED 'ABC' TO #F          #F:   ABC
MOVE EDITED '003.45' TO #G (EM=999.99)    #G:    3.45
MOVE EDITED 123.45 (EM=999.99) TO #H      #H: 123.45
```

 Copyright © Software AG 2002

Example 2

```
/* EXAMPLE 'MOVEX2': MOVE BY NAME
/*********************************************************
DEFINE DATA LOCAL
1 #SBLOCK
  2 #FIELDA (A10) INIT <'AAAAAAAAAA'>
  2 #FIELDB (A10) INIT <'BBBBBBBBBB'>
  2 #FIELDC (A10) INIT <'CCCCCCCCCC'>
  2 #FIELDD (A10) INIT <'DDDDDDDDDD'>
1 #TBLOCK
  2 #FIELD1 (A15) INIT <' '>
  2 #FIELDA (A10) INIT <' '>
  2 #FIELD2 (A10) INIT <' '>
  2 #FIELDB (A10) INIT <' '>
  2 #FIELD3 (A20) INIT <' '>
  2 #FIELDC (A10) INIT <' '>
END-DEFINE
/*********************************************************
MOVE BY NAME #SBLOCK TO #TBLOCK
/*********************************************************
WRITE NOTITLE 'CONTENTS OF #TBLOCK AFTER MOVE BY NAME:'
       // '=' #TBLOCK.#FIELD1
        / '=' #TBLOCK.#FIELDA
        / '=' #TBLOCK.#FIELD2
        / '=' #TBLOCK.#FIELDB
        / '=' #TBLOCK.#FIELD3
        / '=' #TBLOCK.#FIELDC
/*********************************************************
END
```

```
CONTENTS OF #TBLOCK AFTER MOVE BY NAME:

  #FIELD1:
  #FIELDA: AAAAAAAAAA
  #FIELD2:
  #FIELDB: BBBBBBBBBB
  #FIELD3:
  #FIELDC: CCCCCCCCCC
```

MOVE ALL

```
MOVE ALL operand1 TO operand2 [UNTIL operand3]
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S				A	N				B						yes	no
Operand2		S	A			A											yes	yes
Operand3	C	S					N	P	I								yes	no

Function

The MOVE ALL statement is used to move repeatedly the value of *operand1* to *operand2* until *operand2* is full.

Source Operand - operand1

The source operand contains the value to be moved.

All digits of a numeric operand including leading zeros are moved.

Target Operand - operand2

The target operand is not reset prior to the execution of the MOVE ALL operation. This is of particular importance when using the UNTIL option since data previously in operand2 is retained if not explicitly overlaid during the MOVE ALL operation.

UNTIL Option - operand3

The UNTIL option is used to limit the MOVE ALL operation to a given number of positions in *operand2*. *Operand3* is used to specify the number of positions. The MOVE ALL operation is terminated when this value is reached.

If *operand3* is greater than the length of *operand2*, the MOVE ALL operation is terminated when *operand2* is full.

The UNTIL option may also be used to assign an initial value to a DYNAMIC variable: if *operand2* is a DYNAMIC variable, its length after the MOVE ALL operation will correspond to the value of *operand3*. The current length of a DYNAMIC variable can be ascertained by using the system variable *LENGTH. For general information on DYNAMIC variables, see your Natural User's Guide.

 Copyright © Software AG 2002

Example

```
/* EXAMPLE 'MOAEX1': MOVE ALL
/*********************************************************
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 4
RD. READ EMPLOY-VIEW BY NAME
     SUSPEND IDENTICAL SUPPRESS
/*********************************************************
FD.  FIND VEH-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
      IF NO RECORDS FOUND
       MOVE ALL '*' TO FIRST-NAME (RD.)
       MOVE ALL '*' TO CITY (RD.)
       MOVE ALL '*' TO MAKE (FD.)
      END-NOREC
/*********************************************************
      DISPLAY NOTITLE (ES=OFF IS=ON ZP=ON AL=15)
              NAME (RD.) FIRST-NAME (RD.) CITY (RD.)
              MAKE (FD.) (IS=OFF)
/*********************************************************
     END-FIND
    END-READ
END
```

```
   NAME         FIRST-NAME         CITY            MAKE
--------------- --------------- --------------- ---------------

ABELLAN         *************** *************** ***************
ACHIESON        ROBERT          DERBY           FORD
ADAM            *************** *************** ***************
ADKINSON        TIMMIE          BEDFORD         GENERAL MOTORS
```

Equivalent reporting-mode example: See the program MOAEX1R in the library SYSEXRM.

MULTIPLY

Syntax 1

```
MULTIPLY [ROUNDED] operand1 BY operand2
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S	A		M		N	P	I	F								yes	no
Operand2	C	S	A		N		N	P	I	F								yes	no

Syntax 2

```
MULTIPLY [ROUNDED] operand1 BY operand2 GIVING operand3
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S	A		M		N	P	I	F								yes	no
Operand2	C	S	A		N		N	P	I	F								yes	no
Operand3		S	A		M	A	N	P	I	F	B							yes	yes

Related Statement: COMPUTE

Function

The MULTIPLY statement is used to multiply two operands.

For multiplications involving arrays, see also the section Arithmetic Operations with Arrays in the Natural Reference documentation.

Result Field

The result of the multiplication may be stored in *operand1* or in *operand3*.

If Syntax 1 is used, the result is stored in *operand1*.

If Syntax 2 (with GIVING clause) is used, *operand1* will not be modified and the result will be stored in *operand3*.

 Copyright © Software AG 2002

If *operand1* is a numeric constant, the GIVING clause is required.

If a database field is used as the result field, the multiplication results in an update only to the internal value of the field as used within the program. The value for the field in the database remains unchanged.

Example

```
/* EXAMPLE 'MULEX1': MULTIPLY
/************************************************************
DEFINE DATA LOCAL
1 #A (N3) INIT <20>
1 #B (N5)
1 #C (N3.1)
1 #D (N2)
1 #ARRAY1 (N5/1:4,1:4) INIT (2,*) <5>
1 #ARRAY2 (N5/1:4,1:4) INIT (4,*) <10>
END-DEFINE
/************************************************************
MULTIPLY #A BY 3
WRITE NOTITLE 'MULTIPLY #A BY 3' 25X '=' #A
/************************************************************
MULTIPLY #A BY 3 GIVING #B
WRITE 'MULTIPLY #A BY 3 GIVING #B' 15X '=' #B
/************************************************************
MULTIPLY ROUNDED 3 BY 3.5 GIVING #C
WRITE 'MULTIPLY ROUNDED 3 BY 3.5 GIVING #C' 6X '=' #C
/************************************************************
MULTIPLY 3 BY -4 GIVING #D
WRITE 'MULTIPLY 3 BY -4 GIVING #D' 15X '=' #D
/************************************************************
MULTIPLY -3 BY -4 GIVING #D
WRITE 'MULTIPLY -3 BY -4 GIVING #D' 14X '=' #D
/************************************************************
MULTIPLY 3 BY 0 GIVING #D
WRITE 'MULTIPLY  3 BY  0 GIVING #D' 14X '=' #D
/************************************************************
WRITE /  '=' #ARRAY1 (2,*) '=' #ARRAY2 (4,*)
MULTIPLY #ARRAY1 (2,*) BY #ARRAY2 (4,*)
WRITE 'MULTIPLY #ARRAY1 (2,*) BY #ARRAY2 (4,*)'
      /  '=' #ARRAY1 (2,*) '=' #ARRAY2 (4,*)
/************************************************************
END
```

```
MULTIPLY #A BY 3                          #A:   60
  MULTIPLY #A BY 3 GIVING #B                #B:    180
  MULTIPLY ROUNDED 3 BY 3.5 GIVING #C       #C:   10.5
  MULTIPLY 3 BY -4 GIVING #D                #D: -12
  MULTIPLY -3 BY -4 GIVING #D               #D:  12
  MULTIPLY  3 BY  0 GIVING #D               #D:   0

  #ARRAY1:      5      5      5      5 #ARRAY2:     10     10     10     10
  MULTIPLY #ARRAY1 (2,*) BY #ARRAY2 (4,*)
  #ARRAY1:     50     50     50     50 #ARRAY2:     10     10     10     10
```

NEWPAGE

```
                                   EVEN [IF] TOP [OF] [PAGE]
NEWPAGE [(rep)] [ [ IF ]                                    ]
                [ [WHEN] LESS [THAN] operand1 [LINES] [LEFT] ]
[ [WITH] TITLE]                    (see WRITE TITLE syntax)
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S					N	P	I									yes	no

Related Statements: AT TOP OF PAGE | AT END OF PAGE

Function

The NEWPAGE statement is used to cause an advance to a new page. NEWPAGE also causes any AT END OF PAGE and WRITE TRAILER statements to be executed. A default title containing the date, time of day, and page number will appear on each new page unless a WRITE TITLE, WRITE NOTITLE, or DISPLAY NOTITLE statement is specified to define specific title processing.

Notes:
The advance to a new page is not performed at the time when the NEWPAGE statement is executed. It is performed only when a subsequent statement
which produces output is executed.
If the NEWPAGE statement is not used, page advance is controlled automatically based on the session parameter PS.

Report Specification - rep

The notation (*rep*) may be used to specify the identification of the report for which the NEWPAGE statement is applicable. A value in the range 0 - 31 or a logical name which has been assigned using the DEFINE PRINTER statement may be specified.

If (*rep*) is not specified, the NEWPAGE statement will be applicable to the first report (report 0).

EVEN IF TOP OF PAGE

This option is used to cause a new page (with corresponding AT TOP OF PAGE and page title processing) to be generated, even if a new page was initiated immediately before the NEWPAGE statement was encountered.

 Copyright © Software AG 2002

WHEN LESS THAN operand1 LINES LEFT

This option is used to cause a new page to be generated when there are less than `operand1` lines left on the current page (current line count compared with value for session parameter PS).

WITH TITLE

The WITH TITLE option may be used to specify a title which is to be written to the new page generated. The title is specified using the same syntax as described for the WRITE TITLE statement, except that the SKIP clause in a NEWPAGE WITH TITLE statement is not allowed.

Example

```
/* EXAMPLE 'NWPEX1S': NEWPAGE
/*********************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 SALARY (1)
  2 CURR-CODE (1)
END-DEFINE
/*********************************************************
LIMIT 15
READ EMPLOY-VIEW BY CITY FROM 'DENVER'
 DISPLAY CITY (IS=ON) NAME SALARY (1) CURR-CODE (1)
  AT BREAK OF CITY
   SKIP 1
/*********************************************************
   NEWPAGE WHEN LESS THAN 10 LINES LEFT
   WRITE '****************************************'
     /   'SUMMARY FOR ' OLD(CITY)
     /   '****************************************'
     /   '****************************************'
     /   'SUM OF SALARIES:' SUM(SALARY(1))
     /   'AVG OF SALARIES:' AVER(SALARY(1))
     /   '****************************************'
    NEWPAGE
/*********************************************************
   END-BREAK
END-READ
END
```

```
PAGE      1                                                97-03-27  15:21:13

          CITY                 NAME           ANNUAL   CURRENCY
                                              SALARY     CODE
  -------------------- -------------------- ---------- --------

  DENVER               TANIMOTO                  33000 USD
                       MEYER                     50000 USD

  ****************************************
  SUMMARY FOR  DENVER
  ****************************************
  ****************************************
  SUM OF SALARIES:      83000
  AVG OF SALARIES:      41500
  ****************************************
```

```
PAGE      2                                                97-03-27  15:21:13

          CITY                 NAME           ANNUAL   CURRENCY
                                              SALARY     CODE
  -------------------- -------------------- ---------- --------

  DERBY                DEAKIN                     7950 UKL
                       GARFIELD                   7200 UKL
                       MUNN                       8200 UKL
                       MUNN                       5200 UKL
                       GREBBY                     8450 UKL
                       WHITT                      7450 UKL
                       PONSONBY                   4450 UKL
                       MAGUIRE                    4000 UKL
                       HEYWOOD                    3800 UKL
                       BRYDEN                     6450 UKL
                       SMITH                     29000 UKL
                       CONQUEST                  34000 UKL
                       ACHIESON                  10500 UKL
```

 Copyright © Software AG 2002

```
PAGE      3                                                97-03-27  15:21:13

          CITY                 NAME           ANNUAL   CURRENCY
                                              SALARY     CODE
  -------------------- -------------------- ---------- --------

  ****************************************
  SUMMARY FOR  DERBY
  ****************************************
  ****************************************
  SUM OF SALARIES:     136650
  AVG OF SALARIES:      10511
  ****************************************
```

Equivalent reporting-mode example: See program NWPEX1R in library SYSEXRM.

NOTITLE...

The following is a list of the NOTITLE... related statements:

- DISPLAY
- PRINT
- WRITE

 Copyright © Software AG 2002

OBTAIN

Note: This statement is valid in reporting mode only.

```
OBTAIN operand1 ...
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S	A	G		A	N	P	I	F	B	D	T	L				yes	yes

Function

The OBTAIN statement is used to make available database fields, or a range of occurrences of a database array in contiguous storage. It may also be used to make available multiple ranges of occurrences.

To be able to identify the different ranges, the beginning index of the range must be used as a qualifier before the index positioning within the range. The index always positions within the range beginning with 1.

operand1

With *operand1* you specify the field(s) to be made available as a result of the OBTAIN statement.

Examples

See the programs OBTEX1 and OBTEX2 in the library SYSEXRM.

ON ERROR

Structured Mode Syntax

```
ON ERROR
    statement...
END-ERROR
```

Reporting Mode Syntax

```
ON ERROR { statement
         { DO statement...DOEND }
```

Function

The ON ERROR statement is used to intercept execution time errors which would otherwise result in a Natural error message, followed by termination of Natural program execution, and a return to command input mode.

When the ON ERROR statement block is entered for execution, the normal flow of program execution has been interrupted and cannot be resumed except for error 3145 (record requested in hold), in which case a RETRY statement will cause processing to be resumed exactly where it was suspended.

This statement is non-procedural (that is, its execution depends on an event, not on where in a program it is located).

ON ERROR Processing within Subroutines

When a subroutine structure is built by using CALLNAT, PERFORM or FETCH RETURN, each module may contain an ON ERROR statement.

When an error occurs, Natural will automatically trace back the subroutine structure and select the first ON ERROR statement encountered in a subroutine for execution. If no ON ERROR statement is found in any module on any level, standard error message processing is performed and program execution is terminated.

Restriction

Only one ON ERROR statement is permitted in a Natural object.

 Copyright © Software AG 2002

System Variables *ERROR-NR and *ERROR-LINE

The Natural system variable *ERROR-NR contains the number of the error detected by Natural.

The Natural system variable *ERROR-LINE contains the line number of the statement which caused the error.

Exiting from an ON ERROR Block

An ON ERROR block may be exited by using a FETCH, STOP, TERMINATE, RETRY or ESCAPE ROUTINE statement. If the block is not exited using one of these statements, standard error message processing is performed and program execution is terminated.

Example

```
/* EXAMPLE 'ONEEX1': ON ERROR
/***************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
1 #NAME (A20)
1 #CITY (A20)
END-DEFINE
/***************************************
REPEAT
 INPUT 'ENTER NAME:' #NAME
  IF #NAME = ' '
   STOP
  END-IF
  FIND EMPLOY-VIEW WITH NAME = #NAME
   INPUT (AD=M) 'ENTER NEW VALUES:' ///
                'NAME:' NAME /
                'CITY:' CITY
   UPDATE
   END TRANSACTION
/***************************************
   ON ERROR
    IF *ERROR-NR = 3009
     WRITE 'LAST TRANSACTION NOT SUCCESSFUL'
         / 'HIT ENTER TO RESTART PROGRAM'
     FETCH 'PROGUPD'
    END-IF
   END-ERROR
/***************************************
  END-FIND
END-REPEAT
END
```

OPEN CONVERSATION

```
OPEN CONVERSATION USING [SUBPROGRAMS] {operand1}...
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S	A			A											yes	no

Related Statements: CLOSE CONVERSATION | DEFINE DATA CONTEXT

Function

The statement OPEN CONVERSATION is used in conjunction with Natural RPC. It allows the client to open a conversation and specify the remote subprograms to be included in the conversation.

When the OPEN CONVERSATION statement is executed, it assigns a unique ID identifying the conversation to the system variable *CONVID.

Subprogram Names - operand1

As *operand1* you specify the names of the remote subprograms to be included in the conversation. The name of a subprogram can be specified either as a constant of 1 to 8 characters, or as an alphanumeric variable of length 1 to 8.

Further Information and Examples

See the Natural RPC description in your Natural RPC documentation or Natural Installation and Operations documentation.

 Copyright © Software AG 2002

OPEN DIALOG

```
OPEN DIALOG operand1 [USING] [PARENT] operand2
                     [ [GIVING] [DIALOG-ID] operand3 ]
                     [ WITH { { operand4 [(AD = { M } )] }
                              { { O }       { A }       }
                              { nX                      }...
                              { PARAMETERS-clause       } } ]
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S				A												yes	no
Operand2	C	S				*										G		no	no
Operand3		S							I									yes	no
Operand4	C	S	A			A	N	P	I	F	B	D	T	L	C	G	O	yes	no

* Handle

Note:
This statement is only available under Windows and Windows NT.

Related Statement: CLOSE DIALOG

Function

This statement is used to open a dialog dynamically.

Dialog Name - operand1

Operand1 is the name of the dialog to be opened.

If the PARAMETERS-clause is used, *operand1* must be a constant.

Handle Name - operand2

Operand2 is the handle name of the parent.

Dialog ID - operand3

Operand3 is a unique identifier returned from the creation of the dialog. It must be defined with format/length I4.

AD=

If operand4 is a variable, you can mark it in one of the following ways:

AD=O	non-modifiable
AD=M	modifiable
AD=A	input only

The default setting for AD= is AD=M.

Operand4 cannot be explicitly specified if operand4 is a constant. AD=O always applies to constants.

AD=M

By default, the passed value of a parameter can be changed in the dialog and the changed value passed back to the invoking object, where it overwrites the original value.

Exception: For a field defined with BY VALUE in the dialog's parameter data area, no value is passed back.

AD=O

If you mark a parameter with AD=O, the passed value can be changed in the dialog, but the changed value cannot be passed back to the invoking object; that is, the field in the invoking object retains its original value.

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=A

If you mark a parameter with AD=A, its value will not be passed to the dialog, but it will receive a value from the dialog. AD=A fields will be reset to empty before the dialog is opened.

For a field defined with BY VALUE in the dialog's parameter data area, the invoking object cannot receive a value. In this case, AD=A only causes the field to be reset to empty before the dialog is invoked.

Passing Parameters to the Dialog

When a dialog is opened, parameters may be passed to this dialog.

As *operand4* you specify the parameters which are passed to the dialog.

With the *PARAMETERS-clause*, parameters may be passed selectively.

 Copyright © Software AG 2002

*n*X

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 dialog.
A parameter that is to be skipped must be defined with the keyword OPTIONAL in the dialog's DEFINE DATA PARAMETER statement. OPTIONAL means that a value can - but need not - be passed from the invoking object to such a parameter.

PARAMETERS-clause

```
PARAMETERS {parameter-name = operand4}... END-PARAMETERS
```

Note:
You can only use the PARAMETERS-clause if operand1 is a constant and the dialog is cataloged.

Parameter-name is the name of the parameter as defined in the parameter data area section of the dialog.

Note:
If the value of a parameter marked with AD=O and passed "by reference" is changed in a dialog, this will lead to a runtime error.

Further Information and Examples

See the section Event-Driven Programming Techniques in the Natural User's Guide for Windows.

OPTIONS

This use of this statement is platform-dependent.

 Copyright © Software AG 2002

PASSW

PASSW = *operand1*

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S				A											yes	no

Function

The PASSW statement is used to specify a password for access to Adabas or VSAM files which have been password-protected.

Password - operand1

The password (*operand1*) may be specified as an alphanumeric constant or the content of an alphanumeric variable. It may consist of up to 8 characters, and must not contain special characters or embedded blanks. If the password is specified as a constant, it must be enclosed in apostrophes.

The password specified with the PASSW statement applies to all database access statements (FIND, GET, HISTOGRAM, READ, STORE) for which no individual password is specified. It remains in effect until another password is specified in the execution of a subsequent PASSW statement or the Natural session is terminated.

A password specified with a specific database access statement applies only to that statement, not to any subsequent statement.

Copyright © Software AG 2002

Natural Security Considerations

In the security profile of a library, you can specify a default Adabas password (as described in the Natural Security documentation); this password applies to all database access statements for which neither an individual password is specified nor a PASSW statement applies. It applies within the library in whose security profile it is specified, and also remains in effect in other libraries you subsequently log on to and in whose security profiles no password is specified.

Restriction

This statement is not valid for DL/I databases.

Password Display Protection - Mainframe only

If the password is specified as a constant, the PASSW statement should always be coded at the very beginning of a source-code line, and there should be no blank between the keyword "PASSW" and the equal sign; this ensures that the password is not visible/displayable in the source code of the program.

In TP mode, you may enter the PASSW statement invisible by entering the terminal command "%*" before you type in the PASSW statement.

In batch mode, a password may be provided by specifying:

```
ADHOC
PASSW='password'
END
ENDHOC
```

The password value will not appear in the printed output.

Example

```
/* EXAMPLE 'PWDEX1:' PASSW
/*********************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
END-DEFINE
/*********************************
PASSW='PASSW1'
/*********************************
LIMIT 5
READ EMPLOY-VIEW
 DISPLAY NOTITLE PERSONNEL-ID NAME
END-READ
/*********************************
END
```

 Copyright © Software AG 2002

```
PERSONNEL         NAME
     ID
  --------- --------------------

  50005600  MORENO
  50005500  BLOND
  50005300  MAIZIERE
  50004900  CAOUDAL
  50004600  VERDIE
```

PERFORM

```
PERFORM subroutine-name [ operand2 [ (AD = { M } ) ] ]...
                                           { O }
                                           { A }
                                   nX
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand2	C	S	A	G		A	N	P	I	F	B	D	T	L	C	G	O	yes	yes

Related Statements: DEFINE SUBROUTINE | DEFINE DATA PARAMETER | CALLNAT | FETCH

Function

The PERFORM statement is used to invoke a Natural *subroutine*.

subroutine-name

For a subroutine name (maximum 32 characters), the same naming conventions apply as for user-defined variables (see the Natural Reference documentation).

The subroutine name is independent of the name of the module in which the subroutine is defined (it may but need not be the same).

The subroutine to be invoked must be defined with a DEFINE SUBROUTINE statement. It may be an inline or external subroutine (see DEFINE SUBROUTINE statement).

Within one object, no more than 50 external subroutines may be referenced.

Data Available in a Subroutine

Inline Subroutines

No explicit parameters can be passed from the invoking object to an inline subroutine.

An inline subroutine has access to the currently established global data area as well as the local data area defined within the same object module.

External Subroutines

An external subroutine has access to the currently established global data area. Moreover parameters can be passed with the PERFORM statement from the invoking object to the external subroutine (*see operand2*); thus, you may reduce the size of the global data area.

 Copyright © Software AG 2002

Parameters - operand2

When an *external* subroutine is invoked with the PERFORM statement, one or more parameters (*operand2*) can be passed with the PERFORM statement from the invoking object to the external subroutine. For an *inline* subroutine, *operand2* cannot be specified.

If parameters are passed, the structure of the parameter list must be defined in a DEFINE DATA statement.

By default, the parameters are passed "by reference", that is, the data are transferred via address parameters, the parameter values themselves are not moved.
However, it is also possible to pass parameters "by value", that is, pass the actual parameter values. To do so, you define these fields in the DEFINE DATA PARAMETER statement of the subroutine with the option BY VALUE or BY VALUE RESULT.

- If parameters are passed "by reference" the following applies: The sequence, format and length of the parameters in the invoking object must match exactly the sequence, format and length of the DEFINE DATA PARAMETER structure of the invoked subroutine. The names of the variables in the invoking object and the subroutine may be different.
- If parameters are passed "by value" the following applies: The sequence of the parameters in the invoking object must match exactly the sequence in the DEFINE DATA PARAMETER structure of the invoked subroutine. Formats and lengths of the variables in the invoking object and the subroutine may be different; however, they have to be data transfer compatible. The names of the variables in the invoking object and the subroutine may be different.
  If parameter values that have been modified in the subroutine are to be passed back to the invoking object, you have to define these fields with BY VALUE RESULT.
  With BY VALUE (without RESULT) it is not possible to pass modified parameter values back to the invoking object (regardless of the AD specification; see also below).

Note:
With BY VALUE, an internal copy of the parameter values is created. The subroutine accesses this copy and can modify it, but this will not affect the original parameter values in the invoking object.
With BY VALUE RESULT, an internal copy is likewise created; however, after termination of the subroutine, the original parameter values are overwritten by the (modified) values of the copy.

For both ways of passing parameters, the following applies:

In the parameter data area of the invoked subroutine, a redefinition of groups is only permitted within a REDEFINE block.

If an array is passed, its number of dimensions and occurrences in the subroutine’s parameter data area must be same as in the PERFORM parameter list.

Note:
If multiple occurrences of an array that is defined as part of an indexed group are passed with the PERFORM statement, the corresponding fields in the subroutine’s parameter data area must not be redefined, as this would lead to the wrong addresses being passed.

AD=

If operand2 is a variable, you can mark it in one of the following ways:

AD=O	non-modifiable
AD=M	modifiable
AD=A	input only

The default setting for AD= is AD=M.

If *operand2* is a constant, AD cannot be explicitly specified. For constants AD=O always applies.

AD=M

By default, the passed value of a parameter can be changed in the subroutine and the changed value passed back to the invoking object, where it overwrites the original value.

Exception: For a field defined with BY VALUE in the subroutine's parameter data area, no value is passed back.

AD=O

If you mark a parameter with AD=O, the passed value can be changed in the subroutine, but the changed value cannot be passed back to the invoking object; that is, the field in the invoking object retains its original value.

Note:
Internally, AD=O is processed in the same way as BY VALUE (see note under Parameters - operand2).

AD=A

If you mark a parameter with AD=A, its value will not be passed **to** the subroutine; it will be reset to empty before the subroutine is invoked, and can be used to receive a value **from** the subroutine.

For a field defined with BY VALUE in the subroutine's parameter data area, the invoking object cannot receive a value. In this case, AD=A only causes the field to be reset to empty before the subroutine is invoked.

*n*X

Note:
This notation is not available on mainframe computers.

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 external subroutine.
A parameter that is to be skipped must be defined with the keyword OPTIONAL in the subroutine's DEFINE DATA PARAMETER statement. OPTIONAL means that a value can - but need not - be passed from the invoking object to such a parameter.

Nested PERFORM Statements

The invoked subroutine may contain a PERFORM statement to invoke another subroutine (the number of nested levels is limited by the size of the required memory).

A subroutine may invoke itself (recursive subroutine). If database operations are contained within an external subroutine that is invoked recursively, Natural will ensure that the database operations are logically separated.

Parameter Transfer with Dynamic Variables

See CALLNAT

 Copyright © Software AG 2002

Example 1

```
/* EXAMPLE 'PEREX1S': PERFORM (STRUCTURED MODE)
/*********************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 ADDRESS-LINE  (A20/2)
  2 PHONE
1 #ARRAY (A75/1:4)
1 REDEFINE #ARRAY
  2 #ALINE  (A25/1:4,1:3)
1 #X (N2) INIT <1>
1 #Y (N2) INIT <1>
END-DEFINE
/*********************************************
LIMIT 5
FIND EMPLOY-VIEW WITH CITY = 'BALTIMORE'
 MOVE NAME            TO #ALINE (#X,#Y)
 MOVE ADDRESS-LINE(1) TO #ALINE (#X+1,#Y)
 MOVE ADDRESS-LINE(2) TO #ALINE (#X+2,#Y)
 MOVE PHONE           TO #ALINE (#X+3,#Y)
 IF  #Y = 3
  RESET INITIAL #Y
  PERFORM PRINT
 ELSE
  ADD 1 TO #Y
 END-IF
 AT END OF DATA
  PERFORM PRINT
 END-ENDDATA
END-FIND
/*********************************************
DEFINE SUBROUTINE PRINT
  WRITE NOTITLE (AD=OI) #ARRAY(*)
  RESET #ARRAY(*)
  SKIP 1
END-SUBROUTINE
/*********************************************
END
```

```
JENSON                  LAWLER                  FORREST
2120 HASSELL            4588 CANDLEBERRY AVE    37 TENNYSON DRIVE
#206                    BALTIMORE               BALTIMORE
(301)998-5038           (301)629-0403           (301)881-3609

ALEXANDER               NEEDHAM
409 SENECA DRIVE        12609 BUILDERS LANE
BALTIMORE               BALTIMORE
(301)345-3690           (301)641-9789
```

Example 2

Program containing PERFORM statement:

```
/* EXAMPLE 'PEREX2' PERFORM EXTERNAL WITH PARAMETER
/*******************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 ADDRESS-LINE  (A20/2)
  2 PHONE
1 #ALINE  (A25/1:4,1:3)
1 #X (N2) INIT <1>
1 #Y (N2) INIT <1>
END-DEFINE
/*******************************************
LIMIT 5
FIND EMPLOY-VIEW WITH CITY = 'BALTIMORE'
 MOVE NAME            TO #ALINE (#X,#Y)
 MOVE ADDRESS-LINE(1) TO #ALINE (#X+1,#Y)
 MOVE ADDRESS-LINE(2) TO #ALINE (#X+2,#Y)
 MOVE PHONE           TO #ALINE (#X+3,#Y)
 IF  #Y = 3
  DO
   RESET INITIAL #Y
   PERFORM PEREX2E #ALINE(*,*)
  DOEND
 ELSE
  ADD 1 TO #Y
 AT END OF DATA
  PERFORM PEREX2E #ALINE(*,*)
LOOP
/*******************************************
END
```

Invoked Subroutine:

```
/* EXAMPLE 'PEREX2E' SUBROUTINE WITH PARAMETER
/*******************************************
DEFINE DATA PARAMETER
1 #ALINE  (A25/1:4,1:3)
END-DEFINE
/*******************************************
DEFINE SUBROUTINE PEREX2E
 WRITE NOTITLE (AD=OI) #ALINE(*,*)
 RESET #ALINE(*,*)
 SKIP 1
RETURN
/*******************************************
END
```

 Copyright © Software AG 2002

```
JENSON                    LAWLER                    FORREST
2120 HASSELL              4588 CANDLEBERRY AVE      37 TENNYSON DRIVE
#206                      BALTIMORE                 BALTIMORE
(301)998-5038             (301)629-0403             (301)881-3609

ALEXANDER                 NEEDHAM
409 SENECA DRIVE          12609 BUILDERS LANE
BALTIMORE                 BALTIMORE
(301)345-3690             (301)641-9789
```

PERFORM BREAK PROCESSING

```
PERFORM BREAK [PROCESSING] [(r)]
   AT BREAK statement...
```

Related Statement: AT BREAK

Function

This statement is used to establish break processing in loops created by FOR, REPEAT, CALL LOOP and CALL FILE statements where no automatic break processing is established, or whenever a user-initiated break processing is desired. Unlike automatic break processing which is executed immediately after the record is read, the PERFORM BREAK statement is executed when it is encountered in the normal flow of the program.

This statement causes a check for a break processing condition (based on the value of a control field) and also results in the evaluation of Natural system functions. This check and system function evaluation are performed each time the statement is encountered for execution. This statement may be executed depending on a condition specified in an IF statement.

Statement Reference Notation - *r*

By default, the final PERFORM BREAK condition is true at the end of execution of the program, subprogram or subroutine.

The (*r*) notation may be used to relate the final processing of a PERFORM BREAK to a specific loop. In this case the PERFORM BREAK is executed in the loop end handling of this loop; after the final automatic BREAK processing and before the AT END OF DATA statements are executed.

AT BREAK *statement...*

See the syntax of the AT BREAK statement.

 Copyright © Software AG 2002

Example

```
/* EXAMPLE 'PBPEX1': PERFORM BREAK PROCESSING
/*******************************************************
DEFINE DATA LOCAL
1 #INDEX (N2)
1 #LINE  (N2) INIT <1>
END-DEFINE
/*******************************************************
FOR #INDEX 1 TO 18
PERFORM BREAK PROCESSING
  AT BREAK OF #INDEX /1/
   WRITE NOTITLE / 'PLEASE COMPLETE LINES 1-9 ABOVE' /
   RESET INITIAL #LINE
  END-BREAK
 WRITE NOTITLE '_' (64) '=' #LINE
 ADD 1 TO #LINE
END-FOR
/*******************************************************
END
```

```
____________________________________________________________ #LINE:   1
____________________________________________________________ #LINE:   2
____________________________________________________________ #LINE:   3
____________________________________________________________ #LINE:   4
____________________________________________________________ #LINE:   5
____________________________________________________________ #LINE:   6
____________________________________________________________ #LINE:   7
____________________________________________________________ #LINE:   8
____________________________________________________________ #LINE:   9

PLEASE COMPLETE LINES 1-9 ABOVE

____________________________________________________________ #LINE:   1
____________________________________________________________ #LINE:   2
____________________________________________________________ #LINE:   3
____________________________________________________________ #LINE:   4
____________________________________________________________ #LINE:   5
____________________________________________________________ #LINE:   6
____________________________________________________________ #LINE:   7
____________________________________________________________ #LINE:   8
____________________________________________________________ #LINE:   9

PLEASE COMPLETE LINES 1-9 ABOVE
```

Equivalent reporting-mode example: See the program PBPEX1R in the library SYSEXRM.

PRINT

```
PRINT [(rep)] [NOTITLE] [NOHDR] [(statement-parameters)]
      { [ nX ]  { 'text' [(attributes)]                  }
      { [ nT ]  { 'c'(n) [(attributes)]                  }
      { [ / ]...{ ['='] operand1 [(parameters)]          }...
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S	A	G	N	A	N	P	I	F	B	D	T	L		G	O	yes	no

Related Statement: WRITE

Function

The PRINT statement is used to produce output in free format.

The PRINT statement differs from the WRITE statement in the following aspects:

- The output for each operand is written according to the value content rather than the length of the operand. Leading zeros for numeric values and trailing blanks for alphanumeric values are suppressed. The session parameter AD defines whether numeric values are printed left or right justified. With AD=L, the trailing blanks of a numeric value are suppressed. With AD=R, the leading blanks of a numeric value are printed.
- If the resulting output exceeds the current line size (LS parameter), the output is continued on the next line as follows:
  An alphanumeric constant or the content of an alphanumeric variable (without edit mask) is split at the rightmost blank or character which is neither a letter nor a numeric character contained on the current line. The first part of the split value is output to the current line, and the second part is written to the next line. For all other operands, the entire value is written to the next line.

Report Specification - rep

The notation (*rep*) may be used to specify the identification of the report for which the PRINT statement is applicable. A value in the range 0 - 31 or a logical name which has been assigned using the DEFINE PRINTER statement may be specified. If (*rep*) is not specified, the PRINT statement will be applicable to the first report (report 0).

NOTITLE

Natural generates a single title line for each page resulting from a PRINT statement. This title contains the page number, the time of day, and the date. Time of day is set at the beginning of the session (TP mode) or at the beginning of the job (batch mode). This title line may be overridden by using a WRITE TITLE statement or may be suppressed by specifying the NOTITLE clause in the PRINT statement.

 Copyright © Software AG 2002

```
PRINT NAME                (default title will be produced)
PRINT NAME
WRITE TITLE 'USER TITLE'  (user title will be produced)
PRINT NOTITLE NAME        (no title will be produced)
```

If the NOTITLE option is used, it applies to all DISPLAY, PRINT and WRITE statements within the same object which write data to the same report.

NOHDR

The PRINT statement itself does not produce any column headers. However, if you use the PRINT statement in conjunction with a DISPLAY statement, you can use the NOHDR option of the PRINT statement to suppress the column headers generated by the DISPLAY statement: the NOHDR option only takes effect if the PRINT statement is executed **after** a DISPLAY statement, the output spans more than one page, and the execution of the PRINT statement causes a new page to be output. Without the NOHDR option, the column headers of the DISPLAY statement would be output on this new page; with NOHDR they will not.

statement-parameters

One or more parameters, enclosed within parentheses, may be specified immediately after the PRINT statement or an element being displayed.

Each parameter specified in this manner will override any previous parameter specified in a GLOBALS command, SET GLOBALS or FORMAT statement. If more than one parameter is specified, the parameters must be separated from one another by one or more blanks. A parameter entry must not be split between two statement lines.

The parameter settings applied here will only be regarded for variable fields, but they have no effect on text-constants. If you would like to set field attributes for a text-constant, they have to be set explicitly for this element.

Example:

```
DEFINE DATA LOCAL
1 VARI (A4)     INIT <'1234'>                  /*     Output
END-DEFINE                                     /*    Produced
*                                              /*    ---------
PRINT           'Text'           VARI          /*    Text 1234
PRINT (PM=I)    'Text'           VARI          /*    Text 4321
PRINT           'Text' (PM=I)    VARI (PM=I)   /*    txeT 4321
PRINT           'Text' (PM=I)    VARI          /*    txeT 1234
END
```

The individual parameters are described in the section Session Parameters of the Natural Reference documentation.

Field Positioning, Text, Attribute Assignment

```
{ [nX] [nT] [/] ... { 'text' [(attributes)]
                      'c'(n) [(attributes)]
                    ['='] operand1 [(parameters)] } } ...
```

Field Positioning Notations

nX

The *n*X notation is used to insert *n* spaces between columns.

Note: (for Mainframes Only)
This notation inserts n spaces between columns. n must not be "0".

```
PRINT NAME 5X SALARY
```

nT

The *n*T notation causes positioning (tabulation) to print position *n*. Backward positioning results in a line advance.

```
PRINT 25T NAME 50T SALARY
```

(causes NAME to print beginning in position 25 and SALARY to print beginning in position 50).

/

A slash causes a line advance when placed between fields or text elements.

```
PRINT NAME / SALARY
```

Text/Attribute Assignment

'text'

text is displayed as text.

```
PRINT 'EMPLOYEE' NAME 'MARITAL/STATUS' MAR-STAT
```

'c' (n)

Identical to '*text*', except that the specified character *c* is displayed *n* times.

```
PRINT '*' (5) '=' NAME
```

 Copyright © Software AG 2002

'='

If '=' is placed immediately before the field, the field name is output immediately before the field value.

```
PRINT '=' NAME
```

attributes

The display and color attributes to be used for text/field display:

1	2
B	BL
C	GR
D	NE
I	PI
N	RE
U	TU
V	YE

1. Display attributes (see the session parameter AD in the Natural Reference documentation).
2. Color attributes (see the session parameter CD in the Natural Reference documentation).

operand1

As *operand1* you specify the field to be printed.

parameters

One or more parameters, enclosed within parentheses, may be specified immediately after *operand1*. Each parameter specified in this manner will override any previous parameter specified in a GLOBALS command, SET GLOBALS or FORMAT statement. If more than one parameter is specified, one or more blanks must be placed between each entry. An entry must not be split between two statement lines.

 Copyright © Software AG 2002

Example

```
/* EXAMPLE 'PRTEX1': PRINT
/*****************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 JOB-TITLE
  2 ADDRESS-LINE (2)
END-DEFINE
LIMIT 1
READ EMPLOY-VIEW BY CITY
/*****************************************************************
WRITE NOTITLE 'EXAMPLE 1:'
           // 'RESULT OF WRITE STATEMENT:'
WRITE       / NAME  ',' FIRST-NAME ':' JOB-TITLE '*' (30)
WRITE       / 'RESULT OF PRINT STATEMENT:'
PRINT       / NAME  ',' FIRST-NAME ':' JOB-TITLE '*' (30)
/*****************************************************************
WRITE      // 'EXAMPLE 2:'
           // 'RESULT OF WRITE STATEMENT:'
WRITE       / NAME 60X ADDRESS-LINE (1:2)
WRITE       / 'RESULT OF PRINT STATEMENT:'
PRINT       / NAME 60X ADDRESS-LINE (1:2)
/*****************************************************************
END-READ
END
```

```
EXAMPLE 1:

RESULT OF WRITE STATEMENT:

SENKO                , WILLIE               : PROGRAMMER
*****************************

RESULT OF PRINT STATEMENT:

SENKO , WILLIE : PROGRAMMER ******************************


EXAMPLE 2:

RESULT OF WRITE STATEMENT:

SENKO
2200 COLUMBIA PIKE    #914

RESULT OF PRINT STATEMENT:

SENKO                                                        2200 COLUMBIA
PIKE  #914
```

PROCESS

Note: This statement is only available with Entire System Server.

```
PROCESS view-name USING operand1 = operand2 GIVING operand3 ...
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S				A	N	P			B						yes	no
Operand2	C	S				A	N	P			B						yes	no
Operand3		S				A	N	P			B						yes	no

Function

The PROCESS statement is used in conjunction with Entire System Server. Entire System Server allows you to use various operating system facilities such as reading and writing files, VTOC and catalog management, JES queues, etc.

See the section Getting Started in the Entire System Server User's Guide for further information on the PROCESS statement and its individual clauses.

USING

The USING clause is used to pass parameters to the Entire System Server processor. This is done by assigning a value (*operand2*) to a field (*operand1*) in a view defined to Entire System Server. See the Entire System Server documentation for view description.

Note:
Multiple specifications of "operand1=operand2" must be separated either by the input delimiter character (as specified with the session parameter ID) or by a comma. A comma must not be used for this purpose, however, if the comma is defined as decimal character (with the session parameter DC).

GIVING

The GIVING clause is used to specify the fields (*operand3*) for which values are to be returned by the Entire System Server processor. Each field must be defined in a view used by Entire System Server.

 Copyright © Software AG 2002

PROCESS COMMAND

Structured Mode Syntax

```
PROCESS COMMAND ACTION

        { CLOSE                                                          }
        {                                                                }
        { { CHECK } USING PROCESSOR-NAME = operand1                      }
        { { EXEC  }       COMMAND-LINE (index [:index]) = operand2       }
        { { TEXT  }                                                      }
        { { HELP  }                                                      }
        {                                                                }
        { GET USING PROCESSOR-NAME = operand1                            }
        {           GETSET-FIELD-NAME = operand3                         }
        {                                                                }
        { SET USING PROCESSOR-NAME = operand1                            }
        {           GETSET-FIELD-NAME = operand3                         }
        {           GETSET-FIELD-VALUE = operand4                        }
```

Reporting Mode Syntax

```
PROCESS COMMAND ACTION

    { CLOSE [GIVING NATURAL-ERROR]

      { CHECK }
      { EXEC  }
      { TEXT  }
      { HELP  }  USING PROCESSOR-NAME = operand1
                       COMMAND-LINE (index [:index]) = operand2
                       GIVING RESULT-FIELD (index [: index])
                       RETURN-CODE [NATURAL-ERROR]

      GET USING PROCESSOR-NAME = operand1
                GETSET-FIELD-NAME = operand3
                GIVING GETSET-FIELD-VALUE [NATURAL-ERROR]

      SET USING PROCESSOR-NAME = operand1
                GETSET-FIELD-NAME = operand3
                GETSET-FIELD-VALUE = operand4 [GIVING NATURAL-ERROR] }
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S				A											no	no
Operand2	C	S	A	G		A	N										no	no
Operand3	C	S				A	N										no	no
Operand4	C	S				A	N	P	I								no	no

Function

Once a Command Processor has been created using the Natural utility SYSNCP, it can be invoked from a Natural program using the PROCESS COMMAND statement.

For details on how to create a Natural Command Processor, please refer to the section Command Processor Maintenance of your Natural User's Guide or the Natural SYSNCP Utility documentation respectively.

Note:
The word "COMMAND" in the PROCESS COMMAND statement is in fact the name of a view. The name of the view that is used need not necessarily be "COMMAND"; however, we recommend the use of "COMMAND" because there exists a DDM with the same name. This DDM must be referenced within the DEFINE DATA statement, for example "COMMAND VIEW OF COMMAND".

 Copyright © Software AG 2002

CLOSE

CLOSE terminates the use of the command processor and releases the command processor buffer. When the command processor is used during a session and is not released with CLOSE, then there exists a buffer named NCPWORK in your thread. The parameters of this buffer may be evaluated by using the system command BUS (which is described in the Natural User's Guide. The runtime part of the command processor requires this buffer; it can be released using the statement PROCESS COMMAND ACTION CLOSE. If any PROCESS COMMAND statement follows this statement, then the command processor buffer will be opened again.

CHECK

CHECK is used as a precautionary measure to determine if a command is executable with the statement PROCESS COMMAND EXEC. It works as follows: for the given processor name, a runtime check is performed in two steps:

- It is checked whether the processor exists in the current library or one of its steplibs;
- The content of the command line COMMAND-LINE (1) is analyzed to determine whether it is acceptable.

In addition, the runtime action definitions "R", "M" and "1-9" are written into RESULT-FIELD (1:9).

If the field NATURAL-ERROR is specified in the view or in the GIVING clause, it returns the error code. If this field is not available and the command analysis fails, a Natural system error occurs.

Note:
As the function of the CHECK option is also performed as part of the EXEC option (see below), it is not necessary to use CHECK before every EXEC.

EXEC

EXEC works exactly the same as CHECK with the addition that the runtime actions are executed as specified in the runtime action editor.

Only COMMAND-LINE (1) is needed. You can use up to 9 occurrences of RESULT-FIELD (however, for optimum performance, you should not use more occurrences than you really need).

Note:
EXEC is the only option which can be used to leave the currently active program. This is the case when the runtime action definition contains a FETCH or STOP statement.

HELP

HELP returns a list of all valid keywords, synonyms, and functions for the purpose of, for example, the creation of online help windows. This list is contained in the field(s) of RESULT-FIELD. The type of help returned is dependent on the content of the command lines. COMMAND-LINE (1) must contain the search criteria. COMMAND-LINE (2), if specified, must contain the start value or a search value. COMMAND-LINE (3), if specified, must contain a start value.

Note:
For optimum performance, the number of occurrences of the field RESULT-FIELD should not exceed the number of lines to be displayed on the screen. At least one occurrence must be used.

Copyright © Software AG 2002

HELP for Keywords

This option returns an alphabetically sorted list of keywords and/or synonyms with their IKNs.

Command Line	Contents
1	Must begin with indicator "K". The types of keywords to be returned: ***** Keywords of all types **1** Keywords with type "1" **2** Keywords with type "2" **3** Keywords with type "3" **P** Keywords with type "P" (parameter) Options: **I** Return IKN in addition to keywords. **T** Show keyword partially in upper case (to show possible abbreviation). **S** Return synonyms in addition to keywords. **X** Return only synonyms of specified keywords. **A** Internal keywords are also returned. **+** Search does not include start value.
2	Start value for the keyword search (optional). By default, the search begins with the start value. However, if you specify the option "+", the search does not include the start value itself, but begins with the next higher value.

The field RESULT-FIELD (1:n) returns the specified list.

Example:

```
Command Line 1:    K*X     Returns all synonyms of all keyword types.
```

```
Command Line 1:    K123S   Returns all keywords of type 1, 2 and 3 including synonyms.
```

HELP for Synonyms

For a given IKN, this option returns the original keyword and all synonyms.

Command Line	Contents
1	Must begin with the indicator "S". Option: **T** Shows keyword partially in upper case (to show possible abbreviation).
2	Internal Keyword Number (IKN) of the keyword in format N4.

 Copyright © Software AG 2002

The field RESULT-FIELD (1) returns the original keyword. The fields RESULT-FIELD (2:*n*) return associated synonyms for this keyword.

Example:

```
Input:
Command Line 1:    S
Command Line 2:    1003
```

```
Output:
Result-Field 1:    Edit
Result-Field 2:    Maintain
Result-Field 3:    Modify
```

HELP for Global Functions

This option returns a list of all global functions.

Command Line	Contents
1	Must begin with the indicator "G". Options: **I** Internal Function Number (IFN) is also returned. **T** Shows keyword partially in upper case (to show possible abbreviation). **S** The keywords returned in RESULT-FIELD will be aligned in columns. **A** Internal keywords are also returned. **1** Only functions containing the given keyword of type 1 are to be returned. **2** Only functions containing the given keyword of type 2 are to be returned. **3** Only functions containing the given keyword of type 3 are to be returned. + Search does not include start value.
2	Start value for global function search. Keywords must be given in sequence "123". By default, the search begins with the start value. However, if you specify the option "+", the search does not include the start value itself, but begins with the next higher value.
3	Must be blank.
4	To search only for global functions with a specific keyword, you specify the keyword here. If you specify a keyword, you also have to specify the keyword type (1, 2 or 3) as option (see above).

The field RESULT-FIELD (1:*n*) returns the specified list.

Example:

```
Input:
Command Line 1:    G
Command Line 2:    ADD
```

```
Output:
Result-Field 1:    ADD CUSTOMER
Result-Field 2:    ADD FILE
Result-Field 3:    ADD USER
```

HELP for Local Functions

This option returns a list of all local functions for a specified location.

Command Line	Contents
1	Must begin with the indicator "L". Options : **I** IFN is also returned. **T** Shows keyword partially in upper case (to show possible abbreviation). **S** The keywords returned in RESULT-FIELD will be aligned in columns. **A** Internal keywords are also returned. **1** Only functions containing given keyword of type 1 are to be returned. **2** Only functions containing given keyword of type 2 are to be returned. **3** Only functions containing given keyword of type 3 are to be returned. **C** Only those functions are returned which are defined for the current location (command line 3 is ignored). **F** Invoke "recursive" listing of local functions; that is, all local commands that lead to the current/specified location will be returned.
2	Start value for local function search (optional). Keywords must be given in sequence "123".
3	The location for which the list is to be returned. Keywords must be given in sequence "123". If no location is specified, the current location of the command processor will be used.
4	Keyword restriction (optional): If you specify a keyword, or an IKN with the format N4, only functions with this keyword will be returned.

The field RESULT-FIELD (1:*n*) returns the specified list.

HELP for IKN

For any given internal keyword numbers (IKN), this option returns the original keyword.

 Copyright © Software AG 2002

Command Line	Contents
1	Must start with "IKN". Options: **A** The internal keyword will be shown. **T** Shows keyword partially in upper case (to show possible abbreviation).
2	The IKN to be translated, in format N4.

The field RESULT-FIELD (1) returns the keyword.

Example:

```
Input:
Command Line 1:    IKN
Command Line 2:    0000002002
```

```
Output:
Result-Field 1:    CUSTOMER
```

HELP for IFN

For any given internal function numbers (IFN), this option returns the keywords of a function.

Command Line	Contents
1	Must start with "IFN". Option: **A** Functions with internal keywords will not be suppressed.
2	The IFN to be translated, in format N10.
3	Further options: **S** Keywords belonging to the IFN will be returned in RESULT-FIELD (1:3). **T** Shows keywords partially in upper case (to show possible abbreviations). **L** IFN will be returned if IFN is used as a location. **C** IFN will be returned if IFN is used as a command.

The field RESULT-FIELD (1) returns the function; if option "S" is used, the function is returned in RESULT-FIELD (1:3).

Example:

```
Input:
Command Line 1:    IFN
Command Line 2:    0001048578
```

```
Output:
Result-Field 1:    DISPLAY INVOICE
```

TEXT

The TEXT option is used to deliver general information about the processor and text associated with a keyword or function. This text is the same as that entered in the keyword editor or action editor of the SYSNCP utility during command processor definition.

Note:
To access texts for keywords and functions, you must have specified "Y" in the field "Catalog user texts" in the processor header maintenance (screen 3) of the SYSNCP utility.

TEXT for General Information

For *general information*, COMMAND-LINE (*), i.e., all command lines, must be blank. Up to nine fields of RESULT-FIELD are returned containing the following information:

RESULT-FIELD	Contents	Format
1	Header 1 for User Text	Text (A40)
2	Header 2 for User Text	Text (A40)
3	"First Entry used as" text	Text (A16)
4	"Second Entry used as" text	Text (A16)
5	"Third Entry used as" text	Text (A16)
6	Number of Entry 1 Keywords	Numeric (N3)
7	Number of Entry 2 Keywords	Numeric (N3)
8	Number of Entry 3 Keywords	Numeric (N3)
9	Number of Cataloged Functions	Numeric (N7)

TEXT for Keyword Information

For *keyword information*, COMMAND-LINE (1) must contain the corresponding keyword; COMMAND-LINE (2) can optionally contain the keyword type (1, 2, 3 or P); COMMAND-LINE (3:6) must be empty.

 Copyright © Software AG 2002

RESULT-FIELD	Contents	Format
1	Keyword comment text	Text (A40)
2	Keyword in full length	Text (A16)
3	Keyword in unique short form	Text (A16)
4	"Keyword used as" entry	Text (A16)
5	Internal keyword number (IKN)	Numeric (N4)
6	Minimum length of keyword	Numeric (N2)
7	Maximum length of keyword	Numeric (N2)
8	Keyword type (1, 2, 3, 1S, 2S, 3S, P)	Text (A2)

TEXT for Function Information

For *function information*, COMMAND-LINE (1:3) must contain the keywords which specify the wanted location. COMMAND-LINE (4:6) contains the keywords which specify the wanted function. For example, if information about the global command ADD USER is to be returned, the command lines 1, 2, 3, and 6 must be blank; the command line 4 must contain "ADD", and the command line 5 must contain "USER".

RESULT-FIELD	Contents	Format
1	Text as defined with the option "T" in runtime action definition.	Text (A40)
2	Internal function number (IFN) of the specified location.	Numeric (N10)
3	Internal function number (IFN) of the specified function.	Numeric (N10)

GET

The GET option is used to read internal command processor information and current command processor settings from the dynamically allocated buffer NCPWORK. The following fields are used:

Field Name	Contents
GETSET-FIELD-NAME (A32)	The name of the variable to be read.
GETSET-FIELD-VALUE (A32)	The value of the specified variable after PROCESS COMMAND ACTION GET is performed.

For a list of possible values for GETSET-FIELD-NAME, see next page.

SET

The SET option is used to modify internal command processor settings in the buffer NCPWORK.

Field Name	Contents
GETSET-FIELD-NAME (A32)	The name of the variable to be modified.
GETSET-FIELD-VALUE (A32)	The value which is to written to the specified variable.

For a list of possible values for GETSET-FIELD-NAME, see next page.

The possible values for GETSET-FIELD-NAME are:

Field Name	Format	G/S*	Content
NAME	A8	G	Name of current processor.
LIBRARY	A8	G	Loaded from library.
FNR	N10	G	Loaded from file.
DBID	N10	G	Loaded from database.
TIMESTMP	A8	G	Time stamp of the current processor.
COUNTER	N10	G	Access counter.
BUFFER-LENGTH	N10	G	Bytes allocated for NCPWORK.
C-DELIMITER	A1	G/S	Multiple command delimiter.
DATA-DELIMITER	A1	G	Delimiter to precede data.
PF-KEY	A1	G/S	PF key may be command (Y/N).
UPPER-CASE	A1	G	Keywords in upper case (Y/N).
UQ-KEYWORDS	A1	G	Keywords unique (Y/N).
IMPLICIT-KEYWORD	A1	G/S	Identifier for implicit keyword entry.
MIN-LEN	N10	G	Minimum length of keywords.
MAX-LEN	N10	G	Maximum length of keywords.
KEYWORD-SEQ	A8	G/S	Keyword sequence.
ALT-KEYWORD-SEQ	A8	G/S	Alternative keyword sequence.
USER-SEQUENCE	A1	G	User may override KEYWORD-SEQ (Y/N).
CURR-LOCATION	N10	G/S	Current location (IFN).
CURR-IKN1	N10	G/S	IKN1 of current location.
CURR-IKN2	N10	G/S	IKN2 of current location.
CURR-IKN3	N10	G/S	IKN3 of current location.
CHECK-LOCATION	N10	G	Last checked location (IFN).
CHECK-IKN1	N10	G	IKN1 of CHECK-LOCATION.
CHECK-IKN2	N10	G	IKN2 of CHECK-LOCATION.
CHECK-IKN3	N10	G	IKN3 of CHECK-LOCATION.
TOP-IKN1	N10	G	IKN1 of topmost keyword.
TOP-IKN2	N10	G	IKN2 of topmost keyword.
TOP-IKN3	N10	G	IKN3 of topmost keyword.
KEY1-TOTAL	N10	G	Number of keywords of type 1.
KEY2-TOTAL	N10	G	Number of keywords of type 2.
KEY3-TOTAL	N10	G	Number of keywords of type 3.
FUNCTIONS-TOTAL	N10	G	Number of cataloged functions.

 Copyright © Software AG 2002

Field Name	Format	G/S*	Content
LOCAL-GLOBAL-SEQ	A8	G/S	Local/global function validation.
ERROR-HANDLER	A8	G/S	General error program.
SECURITY	A1	G	Natural Security installed (Y/N).
SEC-PREFETCH	A1	G	Natural Security data are to be read (Y/N) or have been read (D = done).
PREFIX1	A1	G	Corresponds to the field "Prefix Character 1" of the Processor Header Maintenance 2 screen.
PREFIX2	A1	G	Corresponds to the field "Prefix Character 2" of the Processor Header Maintenance 2 screen.
HEX1	A1	G	Corresponds to the field "Hex. Replacement 1" of the Processor Header Maintenance 2 screen.
HEX2	A1	G	Corresponds to the field "Hex. Replacement 2" of the Processor Header Maintenance 2 screen.
DYNAMIC	A32	G	Dynamic part (:n:) of last error message.
LAST	-	G	Last command placed on top of stack as data.
LAST-ALL	-	G	Last commands placed on top of stack as data.
LAST-COM	-	G	Last command moved to *COM.
MULTI	-	G	Places the last of multiple commands as data on top of the stack.
MULTI-COM	-	G	Places the last of multiple commands in the system variable *COM.

***G** = Can be used with the GET option.
***S** = Can be used with the SET option.

USING Clause

The contents of the fields in the USING clause specify, for example, the processor name and the command line.

Specified in the USING clause are fields to be sent to the command processor.

Option	Field Name			
	PROCESSOR-NAME	**COMMAND-LINE**	**GETSET-FIELD-NAME**	**GETSET-FIELD-VALUE**
CLOSE				
CHECK	mandatory	mandatory		
EXEC	mandatory	mandatory		
TEXT	mandatory	mandatory		
HELP	mandatory	mandatory		
GET	mandatory		mandatory	
SET	mandatory		mandatory	mandatory

GIVING Clause

Note:
This clause can only be used in reporting mode.

Specified in the GIVING clause are fields to be filled by the command processor as a result of the processing of any option.

Option	Field Name			
	Natural-ERROR	RETURN-CODE	RESULT-FIELD	GETSET-FIELD-VALUE
CLOSE	recommended			
CHECK	recommended	mandatory	mandatory	
EXEC	recommended	mandatory	mandatory	
TEXT	recommended	mandatory	mandatory	
HELP	recommended	mandatory	mandatory	
GET	recommended			mandatory
SET	recommended			

Note:
The GIVING clause is not available in structured mode, because there exists an implicit GIVING clause made up of all fields specified in the DEFINE DATA statement, which are usually referenced in the GIVING clause for reporting mode.
This means that in structured mode all fields that are marked as "mandatory" in the table above must be defined in the DEFINE DATA statement.

DDM "COMMAND"

The DDM "COMMAND" has been created specifically for use in conjunction with the PROCESS COMMAND statement:

```
DB:    1 File:    1 - COMMAND                          Default Sequence: ?

 TYL  DB  NAME                              F LENG  S D   REMARKS
 ---  --  -------------------------------- - ----  - -  ------------------------
   1  AA  PROCESSOR-NAME                   A    8  N D DE    USING
 M 1  AB  COMMAND-LINE                     A   80  N D MU/DE USING
   1  AF  GETSET-FIELD-NAME                A   32  N D DE    USING
   1  BA  NATURAL-ERROR                    N  4.0  N         GIVING
   1  BB  RETURN-CODE                      A    4  N         GIVING
 M 1  BC  RESULT-FIELD                     A   80  N   MU    GIVING
   1  BD  GETSET-FIELD-VALUE               A   32  N D       USING; GIVING
 ***** DDM OUTPUT TERMINATED ******
```

Note:
To avoid possible compilation or runtime errors, please make sure that the DDM "COMMAND" is cataloged as type "C" (field "DDM Type" on the SYSDDM Menu) before you use it. (If you re-catalog the DDM, any DBID/FNR specification in SYSDDM will be ignored.)

 Copyright © Software AG 2002

The DDM "COMMAND" contains the following fields:

DDM Field	Explanation
PROCESSOR-NAME	The name of the command processor for which the PROCESS COMMAND statement is issued. The command processor specified must be cataloged.
COMMAND-LINE	The command line to be processed by the command processor (options CHECK, EXEC), or the keyword/command for which user text or help text is to be returned to the program (TEXT, HELP options). Note that this field may extend beyond one line.
GETSET-FIELD-NAME	This field is used with the GET and SET options and is used to specify the name of a constant or variable which is to be read (GET) or written (SET).
RETURN-CODE	This field contains the return code of an action resulting from the option EXEC or CHECK as specified within a runtime action definition (see the Natural utility SYSNCP).
NATURAL-ERROR	This field is used in conjunction with all options. When the field is used in DEFINE DATA, then it returns the Natural error code for the command processor. When the field is absent, normal Natural error handling is active.
RESULT-FIELD	This field contains information resulting from the use of various options as specified within a runtime action definition (see Runtime Actions in the Natural SYSNCP Utility). Please note that this field may be more than one line.
GETSET-FIELD-VALUE	This field is used with the GET and SET options and contains the value of the constant or variable which is specified in the field GETSET-FIELD-NAME (see above).

Security Considerations

With Natural Security, it is possible to restrict the usage of certain keywords and/or functions which are defined in a Command Processor. Keywords and/or functions can be allowed/disallowed for a specific user or group of users. See your Natural Security documentation for details.

Example 1

```
/* EXAM-CLS - Example for PROCESS COMMAND ACTION CLOSE (Structured Mode)
/**********************************************************************
DEFINE DATA LOCAL
  01 COMMAND VIEW OF COMMAND
END-DEFINE
/*
PROCESS COMMAND ACTION CLOSE
/*
DEFINE WINDOW CLS
INPUT WINDOW = 'CLS'
  'NCPWORK has just been released.'
/*
END
```

Example 2

```
/* EXAM-EXS - Example for PROCESS COMMAND ACTION EXEC (Structured Mode)
/**********************************************************************
DEFINE DATA LOCAL
  01 COMMAND VIEW OF COMMAND
     02 PROCESSOR-NAME
     02 COMMAND-LINE (1)
     02 NATURAL-ERROR
     02 RETURN-CODE
     02 RESULT-FIELD (1)
  01 MSG (A65) INIT <'Please enter a command.'>
END-DEFINE
/*
REPEAT
  INPUT (AD=MIT' ' IP=OFF) WITH TEXT MSG
    'Example for PROCESS COMMAND ACTION EXEC (Structured Mode)' (I)
  / 'Command ==>' COMMAND-LINE (1) (AL=64)
  /*******
  PROCESS COMMAND ACTION EXEC
    USING
      PROCESSOR-NAME = 'DEMO'
      COMMAND-LINE (1) = COMMAND-LINE (1)
  /*******
  COMPRESS 'NATURAL-ERROR =' NATURAL-ERROR TO MSG
END-REPEAT
END
```

Note:
You will find other example programs in the library SYSNCP. These programs all begin with "EXAM".

 Copyright © Software AG 2002

PROCESS GUI

Note:
This statement is only available under Windows.

```
PROCESS GUI ACTION action-name WITH { { operand1 }
                                    { { nX       }...
                                    { PARAMETERSclause } [GIVING operand2]
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1*	C	S	A			A	N	P	I	F	B	D	T	L		G		yes	no
Operand2		S					N	P	I									yes	no

* The structure and format actually possible depend on the action to be performed.

Function

The PROCESS GUI statement is used to perform an action. An action in this context is a procedure frequently needed in event-driven applications.

For general information on these standard procedures, see the section **Event-Driven Programming Techniques** of the Natural User's Guide for Windows.

For information on the individual actions available, their parameters, and examples, see the section **Executing Standardized Procedures** of the Natural Dialog Components documentation for Windows.

action-name

As action-name, you specify the name of the action to be invoked.

Passing Parameters to the Action

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-clause

```
PARAMETERS {parameter-name = operand1}... END-PARAMETERS
```

This clause can only be used for the action "ADD", not for any other action.

If the action has optional parameters (i. e. parameters that need not to be specified), you can use the notation *n*X as a placeholder for *n* not specified parameters. Currently, the only actions that can have optional parameters are the methods and the parameterized properties of ActiveX controls.

nX

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 action. This is only possible for actions which are applied to ActiveX controls.

A parameter that is to be skipped must be defined as "optional" in the ActiveX control’s method. If a parameter is defined as "optional", this means that a value can - but need not - be passed from the invoking object to such a parameter.

GIVING operand2

As *operand2*, you can specify a field to receive the response code from the invoked action after the action has been performed.

 Copyright © Software AG 2002

PROCESS REPORTER

Note:
This statement is only available under Windows and Windows NT.

```
PROCESS REPORTER ACTION
        { INITIALIZE
          TERMINATE

          { OPEN
            CLOSE
            REPLACE-TABLE
            SET-PRINTER
            SET-PRINT-OPTIONS
            PRINT
            PREVIEW } WITH { operand1 ...
                             PARAMETERS-clause }

          EDIT [WITH { operand1 ...
                       PARAMETERS-clause } ]
        [GIVING operand2] }
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S				A	N	P	I	F	B	D	T	L				yes	no
Operand2		S					N	P	I									yes	no

Function

The PROCESS REPORTER statement is used to communicate with the Natural reporter from within a program, instructing the reporter to perform a particular action.

For a description of the reporter, please refer to the Natural Reporter online help.

Note:
For actions that apply to a specific report, you may abbreviate the second keyword to REPORT. This is only to enhance the readability of your programs; Natural does not distinguish between the written-out and abbreviated forms of the keyword.

Actions

You can specify one of the following actions to be performed by the reporter:

- **INITIALIZE** - This action initializes and loads the reporter. This must always be the first action to be performed.
- **TERMINATE** - This action terminates and unloads the reporter. This must always be the last action to be performed.
- **OPEN** - This action opens a specified report, and returns a handle which can be used to identify the report for subsequent actions.
- **CLOSE** - This action closes a specified report, after which the report handle can no longer be used.
- **REPLACE-TABLE** - This action replaces the path name of a table.
- **SET-PRINTER** - This action selects a printer to be used for subsequent printing of all reports. The print method for the selected printer must be set to "TTY" in NATPARM.
- **SET-PRINT-OPTIONS** - This action is used to set print options for a specified report.
- **PRINT** - This action prints a specified report on the currently selected printer.
- **PREVIEW** - This action previews a specified report, based on the currently selected printer.
- **EDIT** - If no report is specified, this action shows the main reporter window. If a report is specified, this action shows the main reporter window together with the edit window for the specified report.

WITH Clause

As *operand1*, you specify the parameter(s) to be passed to the action.

Alternatively, you can use the *PARAMETERS-clause*:

PARAMETERS-clause

PARAMETERS {*parameter-name* = *operand1*}... **END-PARAMETERS**

With this clause, you specify the parameter(s) by name (instead of by position).

Parameters for OPEN Action

For this action, you specify as first parameter the name of the report to be opened (without .rpt extension or path specification), and as second parameter the field to receive the handle. The format/length of the first parameter must be compatible with A8, that of the second parameter with I4.

The report is searched for in the logon library's RES subdirectory first, then in the RES subdirectory of each steplib, then in the directory assigned to the environment variable NATGUI_BMP.

Note that the report data is first searched for in the path specified when the report was created (if it exists), then in the directory in which the report was found.

If you use the *PARAMETERS-clause*, the *parameter-name* must be "REPORT-NAME" for the report name, and "REPORT-ID" for the handle field.

Examples:

```
PROCESS REPORT ACTION OPEN WITH 'MYREPORT' #HANDLE
```

 Copyright © Software AG 2002

```
PROCESS REPORT ACTION OPEN WITH
  PARAMETERS
    REPORT-NAME = 'MYREPORT'
    REPORT-ID   = #HANDLE
  END-PARAMETERS
```

Parameters for REPLACE-TABLE Action

For this action, you specify as first parameter the handle identifying the report to which the action is to be applied, as second parameter the work file number, and, optionally, as third parameter the table name. The format/length of the first two parameters must be compatible with I4, that of the third parameter with A8.

If you use the *PARAMETERS-clause*, the *parameter-names* must be "REPORT-ID", "WORK-FILE" and "TABLE-NAME" respectively.

Example:

```
PROCESS REPORT ACTION REPLACE-TABLE WITH
  PARAMETERS
    REPORT-ID = #HANDLE
    WORK-FILE = 5
  END-PARAMETERS
```

Parameter for SET-PRINTER Action

For this action, you specify as *operand1* the logical device name ('LPT1' to 'LPT31') of the printer to be selected. The format/length of *operand1* must be compatible with A8.

If you use the *PARAMETERS-clause*, the *parameter-name* must be "DEVICE-NAME".

Example:

```
PROCESS REPORTER ACTION SET-PRINTER WITH 'LPT1'
```

Parameters for SET-PRINT-OPTIONS Action

For this action, you specify as first parameter the handle identifying the report to which the action is to be applied, followed by the printer options to be set - all of which are optional. If a parameter is omitted, the corresponding option remains unchanged.

The 1st parameter (which must be compatible with format/length I4) is the handle identifying the report to which the action is to be applied.

The 2nd parameter (which must be compatible with format/length I2) is one of the paper-size constants defined in the local data area NGULKEY1. The possible values here are:

- CUSTOM-PAPER (use explicit paper width and height)
- LETTER (8.5 x 11 inches)
- LEGAL (8.5 x 14 inches)
- EXECUTIVE (7.25 x 10.5 inches)
- A4 (210 x 297 mm)
- COM-10-ENVELOPE (4.125 x 9.5 inches)
- DL-ENVELOPE (110 x 220 mm)

- C5-ENVELOPE (162 x 229 mm)
- B5-ENVELOPE (176 x 250 mm)
- MONARCH-ENVELOPE (3.875 x 7.5 inches)

The 3rd and 4th parameters (which must be compatible with format/length I2) are the paper width and height respectively (in twips; 1 twip = 1/1440 inches). These parameters are only used with paper size CUSTOM-PAPER.

The 5th, 6th, 7th and 8th parameters (which must be compatible with format/length I2) specify the left, top, right and bottom margins respectively (in twips).

The 9th parameter (which must be of format L) is the paper orientation: TRUE = landscape, FALSE = portrait. This parameter is not used with paper size CUSTOM-PAPER.

The 10th parameter (which must be of format L) is the fast (text only) print option: TRUE = suppression of graphics, FALSE = no suppression.

The 11th parameter (which must be of format L) determines whether records that consist entirely of blanks are to be suppressed in the output: TRUE = suppression, FALSE = no suppression.

The 12th parameter (which must be of format L) determines whether successive records with identical data are to be ignored: TRUE = ignore, FALSE = do not ignore.

The 13th parameter (which must be of format L) determines whether a printer selection dialog is to be displayed during printing: TRUE = display, FALSE = no display.

The 14th parameter (which must be compatible with format/length I2) is one of the paper-source constants defined in the local data area NGULKEY1. The possible values here are: AUTOMATIC = automatic feed, MANUAL = manual feed.

If you use the *PARAMETERS-clause*, the *parameter-names* must be REPORT-ID, PAPER-SIZE, PAPER-WIDTH, PAPER-HEIGHT, LEFT-MARGIN, TOP-MARGIN, RIGHT-MARGIN, BOTTOM-MARGIN, LANDSCAPE, FAST-PRINT, SUPPRESS- BLANK-LINES, IGNORE-DUPLICATES, SHOW-PRINT-DIALOG and PAPER-SOURCE respectively.

Examples:

```
DEFINE DATA LOCAL
  USING 'NGLUKEY1'
END-DEFINE
...
PROCESS REPORT ACTION SET-PRINT-OPTIONS WITH #HANDLE
  A4 0 0 0 0 0 0 FALSE FALSE FALSE FALSE FALSE AUTOMATIC
```

 Copyright © Software AG 2002

```
DEFINE DATA LOCAL
  USING 'NGLUKEY1'
END-DEFINE
...
PROCESS REPORT ACTION SET-PRINT-OPTIONS WITH PARAMETERS
  REPORT-ID = #HANDLE
  PAPER-SIZE = A4
  PAPER-WIDTH = 0
  PAPER-HEIGHT = 0
  LEFT-MARGIN = 0   TOP-MARGIN = 0
  RIGHT-MARGIN = 0  BOTTOM-MARGIN = 0
  LANDSCAPE = FALSE
  FAST-PRINT = FALSE
  SUPPRESS-BLANK-LINES = FALSE
  IGNORE-DUPLICATES = FALSE
  SHOW-PRINT-DIALOG = FALSE
  PAPER-SOURCE = AUTOMATIC
END-PARAMETERS
```

Parameter for CLOSE, PRINT, PREVIEW, EDIT Actions

For these actions, you specify as *operand1* the handle identifying the report to which the action is to be applied. The format/length of *operand1* must be compatible with I4.

If you use the *PARAMETERS-clause*, the *parameter-name* must be "REPORT-ID".

Examples:

```
PROCESS REPORT ACTION PRINT WITH #HANDLE
PROCESS REPORT ACTION PREVIEW WITH #HANDLE
PROCESS REPORT ACTION CLOSE WITH #HANDLE
PROCESS REPORT ACTION EDIT WITH #HANDLE
PROCESS REPORTER ACTION EDIT
```

GIVING operand2

With the GIVING clause, you can retrieve the response code from the invoked action.

As *operand2*, you specify the field to receive the response code.

The response code is returned in format/length I4.

Response code "0" indicates that the action was successful. Any other response code corresponds to a Natural system error number (NAT*nnnn*).

PROPERTY

```
PROPERTY  property-name

          OF [INTERFACE ] interface-name
          IS operand

END-PROPERTY
```

Function

The PROPERTY statement assigns an object data variable operand as the implementation to a property, outside an interface definition. It is used if the interface definition in question is included from a copycode and is to be implemented in a class-specific way.

It may only be used within the DEFINE CLASS statement and after the interface definitions. The interface and property names specified must be defined in the interface definitions.

 Copyright © Software AG 2002

READ

```
{ READ   } [ ALL        ] [RECORD S] [IN] [FILE] view-name
{ BROWSE } [ (operand1) ]
        [PASSWORD = operand2]
        [CIPHER = operand3]
        [WITH REPOSITION]
        [sequence/range-specification]
        [STARTING WITH ISN = operand4]
        [WHERE logical-condition]
        statement...
END-READ        (structured mode only)
[LOOP]          (reporting mode only)
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S					N	P	I		B						yes	no
Operand2	C	S				A											yes	no
Operand3	C	S					N										yes	no
Operand4	C	S					N	P	I		B						yes	no

Related Statement: FIND | HISTOGRAM

Function

The READ statement is used to read records from a database. The records can be retrieved in physical sequence, in Adabas ISN sequence, or in the value sequence of a descriptor (key) field.

This statement causes a processing loop to be initiated.

Number of Records - operand1/ALL

The number of records to be read may be limited by specifying *operand1* (enclosed in parentheses, immediately after the keyword READ) - either as a numeric constant (0 - 99999999) or as a variable, enclosed within parentheses, immediately after the keyword READ. For example:

```
READ (5) IN EMPLOYEES ...

MOVE 10 TO CNT(N2)
READ (CNT) EMPLOYEES  ...
```

For this statement, the specified limit has priority over a limit set with a LIMIT statement.

If a smaller limit is set with the LT parameter, the LT limit applies.

To emphasize that *all* records are to be read, you can optionally specify the keyword ALL.

Notes:
If you wish to read a 4-digit number of records, specify it with a leading zero: (0nnnn); because Natural interprets every 4-digit number enclosed in parentheses as a line-number reference to a statement.
Operand1 is evaluated when the READ loop is entered. If the value of operand1 is modified within the READ loop, this does not affect the number of records read.

view-name

As *view-name*, you specify the name of a view, which must have been defined either within a DEFINE DATA statement or outside the program in a global or local data area.

In reporting mode, *view-name* may also be the name of a DDM.

PASSWORD and CIPHER Clauses

These clauses are applicable only to Adabas or VSAM databases. They cannot be used with Entire System Server.

The PASSWORD clause is used to provide a password when retrieving data from a file which is password-protected.

The CIPHER clause is used to provide a cipher key when retrieving data from a file which is enciphered.

See the statements FIND and PASSW for further information.

WITH REPOSITION

This option can only be applied to VSAM databases.

With this option, you can reposition to another start value for the database records read within the active READ loop. Processing of the READ statement then continues with the new start value.

The repositioning is triggered by the value of the system variable *COUNTER being reset to "0"; that is, the new start value is used as soon as *COUNTER is "0".

Example:

 Copyright © Software AG 2002

```
DEFINE DATA LOCAL
1 MYVIEW VIEW OF ...
  2 NAME
1 #STARTVAL (A20) INIT <'A'>
1 #ATTR     (C)
END-DEFINE
...
SET KEY PF3
...
READ MYVIEW WITH REPOSITION BY NAME = #STARTVAL
  INPUT (IP=OFF AD=O) 'NAME:' NAME /
    'Enter new start value for repositioning:' #STARTVAL (AD=MT CV=#ATTR) /
    'Press PF3 to stop'
  IF *PF-KEY = 'PF3'
    THEN STOP
  END-IF
  IF #ATTR MODIFIED
    THEN RESET *COUNTER
  END-IF
END-READ
...
```

sequence/range-specification

```
1  [IN] [PHYSICAL] [ ASCENDING | DESCENDING | VARIABLE operand5 ] [SEQUENCE]

2  { BY | WITH } ISN [ [ = | EQ | EQUAL TO | [STARTING] FROM ] operand6 ] [ [ THRU | ENDING AT ] operand7 ]

3  [IN] [LOGICAL] [ ASCENDING | DESCENDING | VARIABLE operand5 ] [SEQUENCE] { BY | WITH } descriptor
                  [ [ = | EQ | EQUAL TO | [STARTING] FROM ] operand6 ] [ [ THRU | ENDING AT ] operand7 ]
```

Options [2] and [3] are not available with Entire System Server.

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand5		S				A												yes	no
Operand6	C	S				A	N	P	I	F	B	D	T	L				yes	no
Operand7	C	S				A	N	P	I	F	B	D	T	L				yes	no

READ IN PHYSICAL SEQUENCE

PHYSICAL SEQUENCE is used to read records in the order in which they are physically stored in a database.

For VSAM databases, READ PHYSICAL can only be applied to ESDS and RRDS.

PHYSICAL is the default sequence.

READ BY ISN

READ BY ISN can only be used for Adabas and VSAM databases; for VSAM databases, it is only valid for ESDS and RRDS.

READ BY ISN is used to read records in the order of Adabas ISNs (internal sequence numbers) or VSAM RBAs (relative byte addresses of ESDS) or RRNs (relative record numbers of RRDS) respectively.

READ IN LOGICAL SEQUENCE

LOGICAL SEQUENCE is used to read records in the order of the values of a descriptor (key) field.

If you specify a descriptor, the records will be read in the value sequence of the descriptor. A descriptor, subdescriptor, superdescriptor or hyperdescriptor may be used for sequence control. A phonetic descriptor, a descriptor within a periodic group, or a superdescriptor which contains a periodic-group field cannot be used.

If you do not specify a descriptor, the default descriptor as specified in the DDM (field "Default Sequence") will be used.

For **DL/I databases**, the descriptor used must be either the sequence field of a root segment, or a secondary index field. If a secondary index field is specified, it must also be specified in the PROCSEQ parameter of a PCB. Natural uses this PCB and the corresponding hierarchical structure to process the database.
As HDAM databases use a randomizing routine to locate root segments, no sensible results will be returned when using READ LOGICAL for HDAM databases; therefore you should use READ PHYSICAL for HDAM databases.

For **VSAM databases**, LOGICAL is only valid for KSDS with primary and alternate keys defined and for ESDS with alternate keys defined.

If the descriptor used for sequence control is defined with null-value suppression (Adabas only), any record which contains a null value for the descriptor will not be read.

If the descriptor is a multiple-value field (Adabas only), the same record will be read multiple times depending on the number of values present.

ASCENDING/DESCENDING/VARIABLE SEQUENCE

This clause only applies to Adabas, VSAM and SQL databases. In a READ PHYSICAL statement, it can only be applied to VSAM databases.

 Copyright © Software AG 2002

With this clause, you can determine whether the records are to be read in ascending sequence or in descending sequence.

- The default sequence is ascending (which may, but need not, be explicity specified by using the keyword ASCENDING).
- If the records are to be read in descending sequence, you specify the keyword DESCENDING.
- if you will determine at runtime whether the records are to be read in ascending or descending sequence, you specify the keyword VARIABLE followed by a variable (*operand5*). The value of *operand5* at the start of the READ processing loop then determines the sequence. *Operand5* must have the format/length A1 and can contain the value "A" (for "ascending") or "D" (for "descending").

Note for Adabas databases:
Descending sequence requires the following Adabas versions (or above): Version 3.1 on UNIX and Windows, Version 3.2 on OpenVMS, and Version 6.1 on mainframe computers.

Note for SQL databases:
On mainframe computers, the VARIABLE option cannot be used for SQL databases.

Example of DESCENDING Option:

```
READ EMPLOYEES IN DESCENDING SEQUENCE BY NAME = 'SMITH'
```

Example of VARIABLE Option:

```
DEFINE DATA LOCAL
1 #DIRECTION (A1) INIT <'A'>   /* 'A' = ASCENDING
1 #EMPVIEW VIEW OF EMPLOYEES
   2 NAME
   ...
END-DEFINE
...
IF *PF-KEY = 'PF7'
  THEN MOVE 'D' TO #DIRECTION
END-IF
READ #EMPVIEW IN VARIABLE #DIRECTION SEQUENCE BY NAME = 'SMITH'
  ...
END-READ
...
```

Further Examples:

See the programs READSCND and REAVSEQ in the library SYSEXRM.

STARTING FROM / ENDING AT

The STARTING FROM and ENDING AT clauses are used to limit reading to a set of records based on a user-specified range of values. The terms THRU and ENDING AT both imply an inclusive range. If a starting value is specified, reading will begin with the value specified. If the starting value does not exist in the file, the next higher value will be used. If no higher value exists, the loop will not be entered.

The keyword "EQ" or "=" only establishes a starting value for the read operation. The ending value must be specified with the ENDING AT option.

If the sequence descriptor is an Adabas hyperdescriptor, the ENDING AT clause must not be specified.

A multiple-value field must not be used with the ENDING AT option.

Note:
Internally, to determine the end of the range to be read, Natural reads one value beyond the ENDING AT value. If you use the last record read for further processing, be aware that this last record is in fact not the last record within the ENDING AT range, but the first record beyond that range (except if there is no further value after the ENDING AT value).

STARTING WITH ISN=operand4

This clause applies only to Adabas and VSAM databases.

Access to Adabas

This clause can be used in conjunction with a READ statement in physical or in logical (ascending/descending) sequence. The value supplied (operand4) represents an Adabas ISN (Internal Sequence Number) and is used to specify a definite record where to start the READ loop.

Logical Sequence

Even if documented with an equal character "=", the READ statement does not return only those records with exactly the start value in the corresponding descriptor field, but starts a logical browse in ascending or descending order, beginning with the start value supplied. If some records have the same contents in the descriptor field, they will be returned in an ISN-sorted sequence.

The STARTING WITH ISN clause is some kind of a "second level" selection criterion that applies only if the start value matches the descriptor value for the first record.

All records with a descriptor value that is the same as the start value and an ISN that is "less equal"("greater equal" for a descending READ) than the start ISN are ignored by Adabas. The first record returned in the READ loop is either

- the first record with descriptor = start value and an ISN "greater" ("less" for a descending READ) than the start ISN,
- or if such a record does not exist, the first record with a descriptor "greater" ("less" for a descending READ) than the start value.

Physical Sequence

The records are returned in the order in which they are physically stored. If a STARTING WITH ISN clause is specified, Adabas ignores all records until the record with the ISN that is the same as the start ISN is reached. The first record returned is the next record following the ISN=start ISN record.

Access to VSAM

This clause can only be used in physical sequence. The value supplied (operand4) represents a VSAM RBA (relative byte address of ESDS) or RRN (relative record number of RRDS), which is to be used as a start value for the read operation.

Examples

This clause may be used for repositioning within a READ loop whose processing has been interrupted, to easily determine the next record with which processing is to continue. This is particularly useful if the next record cannot be identified uniquely by any of its descriptor values. It can also be useful in a distributed client/server application where the reading of the records is performed by a server program while further processing of the records is performed by a client program, and the records are not processed all in one go, but in batches.

 Copyright © Software AG 2002

For an example, see the program REASISND in the library SYSEXRM.

WHERE Clause

With the WHERE clause, you can specify an additional selection criterion in the form of a logical-condition. This criterion is evaluated **after** a record has been read and **before** any further processing (including AT BREAK processing) is performed on the record.

For details on logical condition criteria, see the Natural Reference documentation.

If a LIMIT statement or a processing limit is specified in a READ statement containing a WHERE clause, records which are rejected as a result of the WHERE clause are not counted against the limit.

System Variables

The following Natural system variables are available with the READ statement:

- ***ISN** - Contains the Adabas ISN of the record currently being processed.
  For VSAM databases, *ISN contains either the RRN (for RRDS) or the RBA (for ESDS) of the current record.
  For DL/I and SQL databases and with Entire System Server, *ISN is not available.
- ***COUNTER** - Contains the number of times the processing loop has been entered.

The format/length of these system variables is P10. This format/length cannot be changed.

The usage of the system variables is illustrated in the example on the next page.

Example 1

```
/* EXAMPLE 'REAEX1S': READ (STRUCTURED MODE)
/**************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
1 VEHIC-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
LIMIT 3
/**************************************************
WRITE 'READ IN PHYSICAL SEQUENCE'
READ EMPLOY-VIEW IN PHYSICAL SEQUENCE
 DISPLAY NOTITLE PERSONNEL-ID NAME *ISN *COUNTER
END-READ
/**************************************************
WRITE / 'READ IN ISN SEQUENCE'
READ EMPLOY-VIEW BY ISN
     STARTING FROM 1 ENDING AT 3
 DISPLAY PERSONNEL-ID NAME *ISN *COUNTER
END-READ
/**************************************************
WRITE / 'READ IN NAME SEQUENCE'
READ EMPLOY-VIEW BY NAME
 DISPLAY PERSONNEL-ID NAME *ISN *COUNTER
END-READ
/**************************************************
WRITE / 'READ IN NAME SEQUENCE STARTING FROM ''M'''
READ EMPLOY-VIEW BY NAME
     STARTING FROM 'M'
 DISPLAY PERSONNEL-ID NAME *ISN *COUNTER
END-READ
/**************************************************
END
```

Equivalent reporting-mode example: See the program REAEX1R in the library SYSEXRM.

 Copyright © Software AG 2002

```
PERSONNEL         NAME           ISN       CNT
     ID
  --------- -------------------- --------- ---------

  READ IN PHYSICAL SEQUENCE
  50005600  MORENO                       2         1
  50005500  BLOND                        3         2
  50005300  MAIZIERE                     4         3

  READ IN ISN SEQUENCE
  50005800  ADAM                         1         1
  50005600  MORENO                       2         2
  50005500  BLOND                        3         3

  READ IN NAME SEQUENCE
  60008339  ABELLAN                    479         1
  30000231  ACHIESON                   884         2
  50005800  ADAM                         1         3

  READ IN NAME SEQUENCE STARTING FROM 'M'
  30008125  MACDONALD                  929         1
  20028700  MACKARNESS                 780         2
  40000045  MADSEN                     509         3
```

Copyright © Software AG 2002

Example 2 - Combining READ with FIND

The following program reads records from the EMPLOYEES file in logical sequential order based on the values of the descriptor NAME. A FIND statement is then issued to the VEHICLES file using the personnel number from the EMPLOYEES file as search criterion. The resulting report shows the name (read from the EMPLOYEES file) of each person read and the model of automobile (read from the VEHICLES file) owned by this person. Multiple lines with the same name are produced if the person owns more than one automobile.

```
/* EXAMPLE 'REAEX2': READ AND FIND
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 10
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
          ENTER
       END-NOREC
       DISPLAY NOTITLE (ES=OFF IS=ON ZP=ON AL=15)
               PERSONNEL-ID (RD.) FIRST-NAME (RD.)
               MAKE (FD.) (IS=OFF)
       END-FIND
     END-READ
END
```

```
 PERSONNEL      FIRST-NAME         MAKE
      ID
--------------- --------------- ---------------

20007500        VIRGINIA        CHRYSLER
20008400        MARSHA          CHRYSLER
                                CHRYSLER
20021100        ROBERT          GENERAL MOTORS
20000800        LILLY           FORD
                                MG
20001100        EDWARD          GENERAL MOTORS
20002000        MARTHA          GENERAL MOTORS
20003400        LAUREL          GENERAL MOTORS
30034045        KEVIN           DATSUN
30034233        GREGORY         FORD
11400319        MANFRED
```

 Copyright © Software AG 2002

READ WORK FILE

Structured Mode Syntax

```
READ WORK [FILE] work-file-number [ONCE]
      { RECORD operand1
        [AND] [SELECT] { [OFFSET n] [FILLER nX]... operand2 }... }
      [GIVING LENGTH operand3]
      { AT [END] [OF] [FILE]
            statement...
        END-ENDFILE }
      statement...
END-WORK
```

Reporting Mode Syntax

```
READ WORK [FILE] work-file-number [ONCE]
      { RECORD {operand1 [FILLER nX]}...
        [AND] [SELECT] { [OFFSET n] [FILLER nX]... operand2 }... }
      [GIVING LENGTH operand3]
      [ AT [END] [OF] [FILE] { statement
                               DO statement...DOEND } ]
      statement...
[LOOP]
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S	A	G		A	N	P	I	F	B	D	T	L	C	G		yes	yes
Operand2		S	A	G		A	N	P	I	F	B	D	T	L	C			yes	yes
Operand3		S							I									yes	yes

Related Statements: DEFINE WORK FILE | CLOSE WORK FILE

Function

The READ WORK FILE statement is used to read data from a non-Adabas physical sequential work file. The data are read sequentially from the work file. How they are read is independent of how they were written to the work file.

On mainframe computers, this statement can only be used within a program to be executed under Com-plete, CMS, TSO or TIAM, or in batch mode. The appropriate JCL must be supplied in the execution JCL when a work file is to be read. See the Natural Operations for Mainframes documentation for additional information.

READ WORK FILE causes a processing loop for reading of all records on the work file to be initiated and executed. Automatic break processing may be performed within a READ WORK FILE loop.

Note:
When an end-of-file condition occurs during the execution of a READ WORK FILE statement, Natural automatically closes the work file.

Note for Entire Connection:
If an Entire Connection work file is read, no I/O statement may be placed within the READ WORK FILE processing loop.

work-file-number

The number of the work file (as defined to Natural) to be read.

ONCE Option

ONCE is used to indicate that only one record is to be read. No processing loop is initiated (and therefore the loop-closing keyword END-WORK or LOOP must not be specified). If ONCE is specified, the AT END OF FILE clause should also be used.

If a READ WORK FILE statement specified with the ONCE option is controlled by a user-initiated processing loop, an end-of-file condition may be detected on the work file before the loop ends. All fields read from the work file still contain the values from the last record read. The work file is then repositioned to the first record which will be read upon the next execution of READ WORK FILE ONCE.

Variable Index Range

When reading an array from a work file, you can specify a variable index range for the array. For example:

READ WORK FILE work-file-number #ARRAY (I:J)

RECORD Option

If RECORD is specified, all fields in each record read are made available for processing. An operand list (*operand1*) must be provided corresponding to the layout of the record. A FILLER *n*X entry indicates *n* bytes are to be skipped in the input record. The record as defined in the RECORD clause must be in contiguous storage. FILLER is not permitted in structured mode.

 Copyright © Software AG 2002

In structured mode, or if the record to be used is defined using a DEFINE DATA statement, only one field (or group) may be used. FILLER is not permitted in this case.

No checking is performed by Natural on the data contained in the record. It is the user's responsibility to describe the record layout correctly in order to avoid program abends caused by non-numeric data in numeric fields. Because no checking is performed by Natural, this option is the fastest way to process records from a sequential file. The record area defined by operand1 is filled with blanks before the record is read. Thus, an end-of-file condition will return a cleared area. Short records will have blanks appended.

Note for Entire Connection:
If an Entire Connection work file is read, the RECORD option cannot be used.

SELECT Option - default

If SELECT is specified, only those fields specified in the operand list (*operand2*) will be made available. The position of the field in the input record may be indicated with an OFFSET and/or FILLER specification. OFFSET 0 indicates the first byte of the record. FILLER *n*X indicates that *n* bytes are to be skipped in the input record.

Natural will assign the selected values to the individual fields and check that numeric fields as selected from the record actually contain valid numeric data according to their definition. Because checking of selected fields is performed by Natural, this option results in more overhead for the processing of a sequential file.

If a record does not fill all fields specified in the SELECT option, the following applies:

- For a field which is only filled partially, the section which has not been filled is reset to blanks or zeros.
- Fields which are not filled at all still have the contents they had before.

Note for Entire Connection:
If an Entire Connection work file is read, the OFFSET option is ignored.

Field Lengths

The field lengths in the operand list are determined as follows:

- For A, B, I and F fields, the number of bytes in the input record is the same as the internal length definition.
- For N format fields, the number of bytes in the input record is the sum of internal positions before and after the decimal point. The decimal point and sign do not occupy a byte position in the input record.
- For P, D and T fields, the number of bytes in the input record is the sum of positions before and after the decimal point plus 1 for the sign, divided by 2 rounded upwards.
- For L format fields, 1 byte is used. For C format fields, 2 bytes are used.

Examples of Field Lengths:

Field Definition	Input Record
#FIELD1 (A10)	10 bytes
#FIELD2 (B15)	15 bytes
#FIELD3 (N1.3)	4 bytes
#FIELD4 (N0.7)	7 bytes
#FIELD5 (P1.2)	2 bytes
#FIELD6 (P6.0)	4 bytes

See also Definition of Format and Length in the Reference part of the documentation.

GIVING LENGTH operand3

The GIVING LENGTH clause can be used to retrieve the actual length of the record being read. The length (number of bytes) is returned in *operand3*. *Operand3* must be defined with format/length I4.

AT END OF FILE

The AT END OF FILE clause can only be used in conjunction with the ONCE option. If the ONCE option is used, this clause should be specified to indicate the action to be taken when an end-of-file condition is detected.

If the ONCE option is not used, an end-of-file condition is handled like a normal processing loop termination.

Handling of Large and Dynamic Variables

The RECORD option is not allowed if any dynamic variables are used.

The work file types ASCII, ASCII-COMPRESSED, ENTIRECONNECTION, SAG (binary) and TRANSFER cannot handle dynamic variables and will produce an error. Large variables pose no problem except if the maximum field/record length is exceeded (field length 255 for ENTIRECONNECTION and TRANSFER, record length 32767 for the others). Reading a dynamic variable from a PORTABLE work file leads to resizing to the stored length. Reading a dynamic variable from an UNFORMATTED work file puts the complete rest of the file into the variable (from the current position). If the file exceeds 1 GB, then at most 1 GB is placed into the variable.

 Copyright © Software AG 2002

Example

```
/* EXAMPLE 'RWFEX1': READ WORK FILE
/****************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
1 #RECORD
  2 #PERS-ID (A8)
  2 #NAME (A20)
END-DEFINE
/****************************************
FIND EMPLOY-VIEW WITH CITY = 'STUTTGART'
 WRITE WORK FILE 1
       PERSONNEL-ID
       NAME
END-FIND
/****************************************
/* ...
/****************************************
READ WORK FILE 1
        RECORD
        #RECORD
 DISPLAY NOTITLE #PERS-ID #NAME
END-WORK
/****************************************
END
```

```
PAGE      1                                                    87-03-27  15:46:58

  #PERS-ID        #NAME
  -------- --------------------

  11100328 BERGHAUS
  11100329 BARTHEL
  11300313 AECKERLE
  11300316 KANTE
  11500304 KLUGE
  11500308 DIETRICH
  11500318 GASSNER
  11500343 ROEHM
  11600303 BERGER
  11600320 BLAETTEL
  11500336 JASPER
  11100330 BUSH
  11500328 EGGERT
```

REDEFINE

Note:
This statement is only valid in reporting mode. To redefine a field in structured mode, use the DEFINE DATA statement.

```
REDEFINE {operand1 ( { nX        } ... ) } ...
         {        } { operand2 }
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S	A	G		A	N	P	I	F	B	D	T	L	C			yes	no
Operand2		S	A	G		A	N	P	I	F	B	D	T	L	C			yes	yes

Function

The REDEFINE statement is used to redefine a field. The resulting definition may consist of one or more user-defined variables.

With one REDEFINE statement, several fields may be redefined.

Method of Redefinition

The byte positions of *operand1* are redefined from left to right regardless of format. The format of *operand2* may be different from the format of *operand1*. The bytes as specified in the REDEFINE statement must positionally match the data contained in the field being redefined. If an alphanumeric field is redefined as numeric and does not contain numeric data according to the format specification, an abnormal termination may result when it is used.

Further Redefinition

Fields defined using a REDEFINE statement may be subsequently redefined with another REDEFINE statement.

Filler Notation

The *n*X notation is used to denote filler bytes within the field/variable being redefined. Any trailing *n*X notation is optional.

 Copyright © Software AG 2002

Example 1

The user-defined variable #A (format/length A10) contains the value "123ABCDEFG".

```
REDEFINE #A (#A1(N3) #A2(A7))
```

The value in #A1 is "123". The value in #A2 is "ABCDEFG".

Example 2

The user-defined variable #B (format/length A10) contains the value (shown in hexadecimal) "12345CC1C2C3C4C5C6C7".

```
REDEFINE #B (#B1(P4) #B2(A7))
```

The value in #B1 is "12345C" (in hexadecimal).
The value in #B2 is "C1C2C3C4C5C6C7" (in hexadecimal).

```
REDEFINE #B (#BB1(B2)8X)) or REDEFINE #B(#BB1(B2))
```

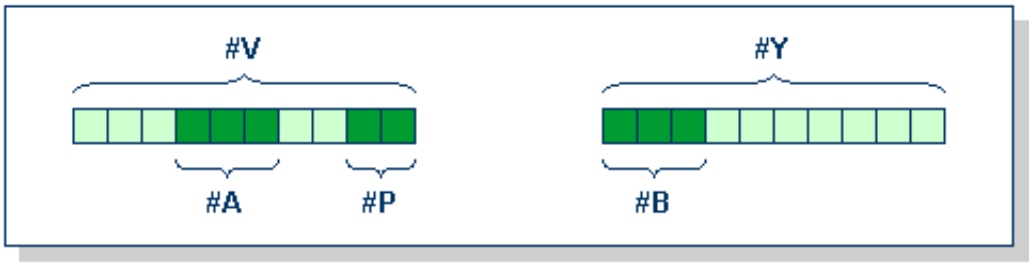
The value in #BB1 is "1234" (in hexadecimal).

Note:
For packed data (P format), the number of decimal positions required must be specified. The following formula can be used to determine the number of bytes that the packed number occupies:

`Number of bytes =` **`(number of decimal positions + 1) / 2`**, rounded upwards to full bytes.

Example 3

```
COMPUTE #V (N8.2) = #Y (N10) = ...
REDEFINE #V (3X  #A(N3)  2X  #P (N2))  #Y (#B(N3)  7X)
```



Example 4

This example redefines the value of the system variable *DATN, which is in the form "YYYYMMDD", and displays the result as three separate fields in the order "day/month/year":

```
MOVE *DATN TO #DATINT (N8)
REDEFINE #DATINT (#YEAR (N4) #MONTH (N2) #DAY (N2))
DISPLAY NOTITLE #DATINT #DAY #MONTH #YEAR
END
```

```
#DATINT  #DAY #MONTH #YEAR
--------- ---- ------ -----

19950108   8    1     1995
```

 Copyright © Software AG 2002

REDUCE

Note:
This statement is not available on mainframe computers.

REDUCE [SIZE OF] DYNAMIC [VARIABLE] *operand 1* **TO** *operand2*

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1		S				A	B										no	no
Operand2	C	S							I								no	no

Related statements: EXPAND

Function

The REDUCE DYNAMIC VARIABLE statement reduces the allocated size of a dynamic variable (operand1) to the size specified (operand2). The allocated memory of the dynamic variable which is beyond the given size is released immediately, i.e., when the statement is executed. If the currently used size (*LENGTH) of the dynamic variable is greater than the given size, *LENGTH is set to the given size and the content of the variable is truncated (but not modified). If the given size is larger than the currently allocated size of the dynamic variable, the statement will be ignored.

operand1

Operand1 is the dynamic variable for which the allocated size is to be reduced.

operand2

Operand2 is used to specify the reduced size. The value specified must be a non-negative numeric value.

Copyright © Software AG 2002

REINPUT

```
REINPUT [FULL]  [(statement-parameters) ] { USING HELP      }
                                          { WITH-TEXT-option }
        [MARK-option]
        [ALARM-option]
```

Related Statement: INPUT

Function

The REINPUT statement is used to return to and re-execute an INPUT statement. It is generally used to display a message indicating that the data input as a result of the previous INPUT statement were invalid.

No WRITE or DISPLAY statements may be executed between an INPUT statement and its corresponding REINPUT statement. The REINPUT statement is not valid in batch mode.

The REINPUT statement, when executed, repositions the program status regarding subroutine, special condition and loop processing as it existed when the INPUT statement was executed (as long as the status of the INPUT statement is still active). If the loop was initiated after the execution of the INPUT statement and the REINPUT statement is within this loop, the loop will be discontinued and then restarted after the INPUT statement has been reprocessed as a result of REINPUT.

If a hierarchy of subroutines was invoked after the execution of the INPUT statement, and the REINPUT is performed within a subroutine, Natural will trace back all subroutines automatically and reposition the program status to that of the INPUT statement.

It is not possible, however, to have an INPUT statement positioned within a loop, a subroutine or a special condition block, and then execute the REINPUT statement when the status under which the INPUT statement was executed has already been terminated. An error message will be produced and program execution terminated when this error condition is detected.

Note:
The execution of a REINPUT statement (without FULL option) does not reset the "MODIFIED" status of a control variable used in the corresponding INPUT statement.

REINPUT FULL

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.

 Copyright © Software AG 2002

Note:
The contents of input-only fields (AD=A) will be deleted again by REINPUT FULL.

statement-parameters

Parameters specified in a REINPUT statement will be applied to all fields specified in the statement.

Any parameter specified at field level (see MARK option) will override any corresponding parameter at statement level.

If AD=P is specified as a statement parameter, all fields - except those used in the MARK option - are protected.

For information on the individual parameters see the section Session Parameters in the Natural Reference documentation.

USING HELP

USING HELP causes the helproutine defined for the INPUT map to be invoked.

USING HELP used in combination with the MARK option (see below) causes the helproutine defined for the first field specified in the MARK option to be invoked. If no helproutine is defined for that field, the helproutine for the map will be invoked.

Example:

```
REINPUT USING HELP MARK 3
```

As a result, the helproutine defined for the third field in the INPUT map will be invoked.

WITH TEXT-option

```
[WITH] [TEXT] { *operand1 } [(attributes)] [,operand3]...7
              { operand2 }
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S					N	P	I		B							yes	no
Operand2	C	S				A												yes	no
Operand3	C	S				A	N	P	I	F	B	D	T	L				yes	no

WITH TEXT is used to provide text which is to be displayed in the message line. This is usually a message indicating what action should be taken to process the screen or to correct an error.

Message Text from Natural Message File - *operand1

Operand1 represents the number of the message text as stored in the Natural message file. The value provided is used to access the Natural message file in order to retrieve a message text.

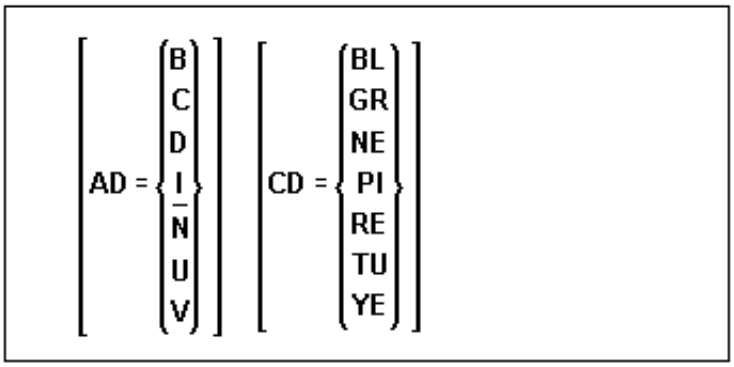
Messages are stored in the Natural message file by library. A maximum of 9999 messages may be stored per library.

See Generating Message Files in the Natural SYSERR Utility documentation for more information.

Message Text - operand2 and Attributes - attributes

Operand2 represents the message to be placed in the message line.

Attributes may be used to assign display and color attributes for *operand1/2*. The following attributes may be specified:



With AD= you specify a display attribute; with CD= you specify a color attribute (see also the session parameters AD and CD in the Natural Reference documentation).

Dynamic Replacement of Message Text - operand3

Operand3 represents a numeric or text constant or the name of a variable.

The values provided are used to replace parts of the message text.

The notation ":*n*:" is used within the message text as a reference to *operand3* contents, where "*n*" represents the occurrence (1 - 7) of *operand3*.

Example:

```
...
MOVE 'MESSAGE-1' TO #FIELD
...
REINPUT 'THE ERROR IS :1:',#FIELD
...
```

As a result, the following message will be output:

THE ERROR IS MESSAGE-1

 Copyright © Software AG 2002

Note:
Multiple specifications of operand3 must be separated from each other by a comma. If the comma is used as a decimal character (as defined with the session parameter DC) and numeric constants are specified as operand3, put blanks before and after the comma so that it cannot be misinterpreted as a decimal character.
Alternatively, multiple specifications of operand3 can be separated by the input delimiter character (as defined with the session parameter ID); however, this is not possible in the case of ID=/ (slash).

Insignificant zeros or blanks will be removed from the field value before it is displayed in a message.

MARK-option

```
MARK [POSITION operand4 [IN]] [FIELD] { { operand5 } [(attributes)] }...
                                       { *fieldname }
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand4	C	S					N	P	I								yes	no
Operand5	C	S	A				N	P	I								yes	no

With the MARK option, you can mark a specific field, that is, specify a field in which the cursor is to be placed when the REINPUT statement is executed. You can also mark a specific position within a field. Moreover, you can make fields input-protected, and change their display and color attributes.

Field to be Marked - operand5

All AD=A or AD=M (that is, non-protected) fields specified in an INPUT statement are sequentially numbered (beginning with 1) by Natural. *Operand5* represents the number of the field in which the cursor is to be positioned.

The **fieldname* notation is used to position to a field (as used in the INPUT statement) using the name of the field as a reference.

If the corresponding INPUT field is an array, a unique index or an index range may be used to reference one or more occurrences of the array.

```
INPUT #ARRAY (A1/1:5)
...
REINPUT (AD=P) 'TEXT' MARK *#ARRAY (2:3)
```

If *operand5* is also an array, the values in *operand5* are used as field numbers for the INPUT array.

```
RESET #X(N2/1:2)
INPUT #ARRAY ...
...
REINPUT (AD=P) 'TEXT' MARK #X (1:2)
```

MARK POSITION

With MARK POSITION, you can have the cursor placed at a specific position - as specified with *operand4* - within a field.

Operand4 must not contain decimal digits.

attributes

```
[ AD = [P] [B] ]  [ CD = { BL } ]
           [C]           { GR }
           [D]           { NE }
           [I]           { PI }
           [N]           { RE }
           [U]           { TU }
           [V]           { YE }
```

With the attribute AD=P, you can make an input field (AD=A or AD=M) input-protected.

Note:
It is not possible via an attribute to make output-only fields (AD=O) available for input.

If AD=P is specified at statement level, all fields except those specified in the MARK option are input-protected.

Moreover, you can change display and color attributes of fields. For information on these attributes, see the session parameters AD and CD in the Natural Reference documentation.

ALARM-option

```
[AND] [SOUND] ALARM
```

This option causes the sound alarm feature of the terminal to be activated when the REINPUT statement is executed. The appropriate hardware must be available to be able to use this feature.

Example 1

 Copyright © Software AG 2002

```
/* EXAMPLE 'REIEX1': REINPUT
/*****************************************************
/* IF FUNCTION = A AND PARM = X
/*    ROUTINE-A IS TO BE EXECUTED.
/* IF FUNCTION = B AND PARM = X
/*    ROUTINE-B IS TO BE EXECUTED.
/* IF FUNCTION = C THRU D
/*    ROUTINE-CD IS TO BE EXECUTED.
/* FOR ALL OTHER CASES,
/*    REINPUT STATEMENT IS TO BE EXECUTED.
/*****************************************************
DEFINE DATA LOCAL
1 #FUNCTION (A1)
1 #PARM (A1)
END-DEFINE
/*****************************************************
INPUT #FUNCTION #PARM
/*****************************************************
DECIDE FOR FIRST CONDITION
           WHEN #FUNCTION = 'A' AND #PARM = 'X'
             PERFORM ROUTINE-A
           WHEN #FUNCTION = 'B' AND #PARM = 'X'
             PERFORM ROUTINE-B
           WHEN #FUNCTION = 'C' THRU 'D'
             PERFORM ROUTINE-CD
           WHEN NONE
             REINPUT 'PLEASE ENTER A VALID FUNCTION'
                     MARK *#FUNCTION
END-DECIDE
/*****************************************************
END
```

```
#FUNCTION a #PARM y
```

```
PLEASE ENTER A VALID FUNCTION
  #FUNCTION a #PARM y
```

Example 2

```
/* EXAMPLE 'REIEX2': REINPUT WITH ATTRIBUTE ASSIGNMENT
/*****************************************************
DEFINE DATA LOCAL
1 #A (A20)
1 #B (N7.2)
1 #C (A5)
1 #D (N3)
END-DEFINE
/*****************************************************
INPUT (AD=A)
  #A  #B  #C  #D
/*****************************************************
IF #A = ' ' OR #B = 0
 REINPUT (AD=P) 'RETYPE VALUES'
                MARK *#A (AD=I CD=RE)
                     *#B (AD=U CD=PI)
END-IF
/*****************************************************
END
```

Example 3

```
/* EXAMPLE 'REIEX3': REINPUT FULL WITH POSITION
/*****************************************************
DEFINE DATA LOCAL
1 #A (A20)
1 #B (N7.2)
1 #C (A5)
1 #D (N3)
END-DEFINE
/*****************************************************
INPUT (AD=M)
  #A  #B  #C  #D
IF #A = ' '
  COMPUTE #B = #B + #D
  RESET #D
END-IF
/*****************************************************
IF #A = SCAN 'TEST' OR = ' '
 REINPUT FULL 'RETYPE VALUES' MARK POSITION 5 IN *#A
END-IF
/*****************************************************
END
```

 Copyright © Software AG 2002

```
#A                      #B        0.00 #C       #D    0



  RETYPE VALUES
```

REJECT

For more information about this statement, see the statement ACCEPT/REJECT.

 Copyright © Software AG 2002

RELEASE

```
RELEASE { STACK
        { SETS [set-name...]
        { VARIABLES
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Set-name	C	S				A											no	no

Related Statements: STACK | FIND with RETAIN option | DEFINE DATA GLOBAL

Function

The RELEASE statement is used to:

- delete the entire contents of the Natural stack;
- release sets of ISNs retained via a FIND statement that contained a RETAIN clause (applicable to Adabas databases only);
- reset global and application-independent variables.

RELEASE STACK

RELEASE STACK causes all data/commands currently in the Natural stack to be deleted.

RELEASE SET

RELEASE SET is applicable to Adabas databases only.

RELEASE SET *set-name* causes a specific single ISN set to be released.

```
RELEASE SET 'CITY-SET'

MOVE 'CITY-SET' TO #SET(A32)
RELEASE SET #SET
```

If only RELEASE SETS, without a *set-name*, is specified, all ISN sets retained with a FIND statement with a RETAIN clause will be released.

RELEASE VARIABLES

RELEASE VARIABLES causes all variables defined in the current global data area to be reset to their initial values. Also, it eliminates all AIVs (application-independent variables), thus making them no longer available.

The variables are reset/eliminated either when the execution of the level 1 program is finished, or when the program invokes another program via a FETCH or RUN statement.

Example

```
 * EXAMPLE 'RELEX1': FIND (RETAIN CLAUSE) AND RELEASE
****************************************************
DEFINE DATA LOCAL
 1 EMPLOY-VIEW VIEW OF EMPLOYEES
   2 CITY
   2 BIRTH
   2 NAME
 1 #BIRTH (D)
END-DEFINE
*
MOVE EDITED '19400101' TO #BIRTH (EM=YYYYMMDD)
*
FIND NUMBER EMPLOY-VIEW WITH BIRTH GT #BIRTH
                        RETAIN AS 'AGESET1'
IF *NUMBER = 0
  STOP
END-IF
*
FIND EMPLOY-VIEW WITH 'AGESET1' AND CITY = 'NEW YORK'
  DISPLAY NOTITLE NAME CITY BIRTH (EM=YYYY-MM-DD)
END-FIND
*
RELEASE SET 'AGESET1'
END
```

```
         NAME                 CITY            DATE
                                               OF
                                             BIRTH
-------------------- -------------------- ----------

RUBIN                NEW YORK             1945-10-27
WALLACE              NEW YORK             1945-08-04
```

 Copyright © Software AG 2002

REPEAT

Syntax 1

```
REPEAT
    statement... [ {UNTIL} logical-condition ]
                   {WHILE}

END-REPEAT   (structured mode only)
LOOP         (reporting mode only)
```

Syntax 2

```
REPEAT
    [ {UNTIL} logical-condition ] statement...
      {WHILE}

END-REPEAT   (structured mode only)
LOOP         (reporting mode only)
```

Related Statement: FOR

Function

The REPEAT statement is used to initiate a processing loop.

If no logical condition is specified, the loop must be exited by an ESCAPE, STOP or TERMINATE statement specified within the loop. If a logical condition is specified, the condition determines when the execution of the loop is to be terminated.

Using syntax 1, the statements are executed one or more times.

Using syntax 2, the statements are executed zero or more times.

The placement of the condition (either at the beginning or at the end of the loop) determines when it is to be evaluated.

For further information on logical conditions, see the section Logical Condition Criteria in the Natural Reference documentation.

UNTIL

The processing loop will be continued until the logical condition becomes true.

WHILE

The processing loop will be continued as long as the logical condition is true.

Example 1

```
/* EXAMPLE 'RPTEX1S': REPEAT (STRUCTURED MODE)
/************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
1 #PERS-NR (A8)
END-DEFINE
/************************************************
REPEAT
 INPUT 'ENTER A PERSONNEL NUMBER:' #PERS-NR
 IF #PERS-NR = ' '
  ESCAPE BOTTOM
 END-IF
 FIND EMPLOY-VIEW WITH PERSONNEL-ID = #PERS-NR
  IF NO RECORD FOUND
   REINPUT 'NO RECORD FOUND'
  END-NOREC
  DISPLAY NOTITLE NAME
 END-FIND
END-REPEAT
/************************************************
END
```

```
ENTER A PERSONNEL NUMBER: 11500304
```

```
       NAME
--------------------

KLUGE
```

Equivalent reporting-mode example: See the program RPTEX1R in the library SYSEXRM.

Example 2:

 Copyright © Software AG 2002

```
/* EXAMPLE 'RPTEX2S': REPEAT (WHILE AND UNTIL OPTIONS)
/*****************************************************
DEFINE DATA LOCAL
1 #X (I1) INIT <0>
1 #Y (I1) INIT <0>
END-DEFINE
/*****************************************************
REPEAT WHILE #X <= 5
 ADD 1 TO #X
 WRITE NOTITLE '=' #X
END-REPEAT
/*****************************************************
SKIP 1
REPEAT
 ADD 1 TO #Y
 WRITE '=' #Y
 UNTIL #Y = 6
END-REPEAT
/*****************************************************
END
```

```
#X:    1
#X:    2
#X:    3
#X:    4
#X:    5
#X:    6

#Y:    1
#Y:    2
#Y:    3
#Y:    4
#Y:    5
#Y:    6
```

Equivalent reporting-mode example: See the program RPTEX2R in the library SYSEXRM.

REQUEST DOCUMENT

Note:
This statement is available under UNIX, OpenVMS and Windows.

```
REQUEST DOCUMENT FROM operand1
[
 WITH
   [USER operand2]
   [PASSWORD operand2]
   [HEADER [[NAME] operand4 [VALUE] operand5]...]
   [DATA { ALL operand6
         [[NAME] operand7 [VALUE] operand8]... }]
]
{RETURN
   [HEADER [ALL operand9]
           [[NAME] operand10 [VALUE] operand11]..]

   [PAGE operand12]
}
RESPONSE operand13
[GIVING operand14]
```

 Copyright © Software AG 2002

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S				A											no	no
Operand2	C	S				A											no	no
Operand3	C	S				A											no	no
Operand4	C	S				A											no	no
Operand5	C	S				A	N	P	I	F		D	T	L			no	no
Operand6	C	S				A	N	P	I	F	B	D	T	L			no	no
Operand7	C	S				A											no	no
Operand8	C	S				A	N	P	I	F		D	T	L			no	no
Operand9		S				A											no	yes
Operand10	C	S				A											no	yes
Operand11		S				A	N	P	I	F	B	D	T	L			no	yes
Operand12		S				A					B						no	yes
Operand13		S							I								no	yes
Operand14		S							I								no	no

Function

The REQUEST DOCUMENT statement gives you the means to access an external system.

Restrictions for Cookies

Under the HTTP Protocol, a server uses cookies to maintain state information on the client workstation.

REQUEST DOCUMENT is implemented using internet option settings. This means that, depending on the security settings, Cookies will be used.

If the internet option setting "Disabled" is set, no cookies will be sent, even if a cookie header (operand 4/5) is sent.

For Server environments, do not use the internet option setting "Prompt". This setting leads to a "hanging" server, because no client will be able to answer the prompt.

In the Windows environment, cookies are handled automatically by the Windows API. This means that, if cookies are enabled in the browser, all incoming cookies will be saved and sent automatically with the next request.

Note:
In a UNIX or OpenVMS environment, the following profile parameters have to be considered: NOPROX, PROXPORT, PROX.

For information on these parameters, refer to the profile parameter descriptions in the Natural Reference documentation.

operand1

Operand1 is the URL to access a document.

The information below is only valid if operand1 begins with "http://" or, in a Windows environment, also with "https://".

operand2

Operand2 is the name of the user that is used for the request.

operand3

Operand3 is the password of the user that is used for the request.

operand4/5

Operand4 is the name of a HEADER variable sent with this request.

Operand5 is the value of a HEADER variable sent with this request.

Note:
Operand4 and operand5 can only be used in conjunction with each other.

Header Name for Operand4

Header names are not allowed to contain CR/LF (carriage return/line feed) or ":" (colon). This will not be checked by the REQUEST DOCUMENT statement. For valid header names, please see the HTTP specifications. For compatibility with the web interface, header names can be written with "_" (underscore) instead of "-" (dash). (Internally, "_" is replaced by"-").

Header Value for Operand5

Header values are not allowed to contain CR/LF. This will not be checked by the REQUEST DOCUMENT statement. For valid header values and formats, please see the HTTP specifications.

General Information

For a HTTP request, some headers are required, eg: Request-Method or Content-Type.

These headers will be automatically generated depending on the parameters given with the REQUEST DOCUMENT statement.

Automatically Generated Headers (operand 4/5)

Request-Method

The following values are supported for operand5: "HEAD", "POST", "GET", and "PUT".

The following table shows the automatic calculation of Request-Method depending on the given operands:

 Copyright © Software AG 2002

Request-Method / Operand	HEAD	POST	GET	PUT
WITH HEADER **(operands 4/5)**	optional	optional	optional	optional
WITH DATA **(operands 7/8)**	not specified	specified	not specified	only with option ALL (operand 6)
RETURN HEADER **(operands 9 to 11)**	specified	optional	optional	optional
RETURN PAGE **(operand 12)**	not specified	specified	specified	optional

Content-Type

If the request method is POST, a content-type header has to be delivered with the HTTP request. If no content-type is set explicitly, the automatically generated value of operand5 is `"application/x-www-form-urlencoded"`.

Note:

It is possible to overwrite the automatically generated headers. Natural will not check them for errors. Unexpected errors may occur.

operand6

Operand6 is a complete document that is to be sent. This value is needed for the HTTP request method PUT.

operand7/8

Operand7 is the name of a DATA variable to be sent with this request. This value is needed for the HTTP request method POST (URL-encoding necessary, especially "&", "=", "%").

Operand8 is the value of a DATA variable to be sent with this request. This value is needed for HTTP request method POST (URL-encoding necessary, especially "&", "=", "%").

Note:
Operand7 and operand8 can only be used in conjunction with each other.

Restriction

If operand 7/8 is given, and the communication is "http://" or "https://" by default, the request method **POST** (see table above) with content type **application/x-www-form-urlencoded** is used.

During the request, the operands 7/8 will be separated by "=" and "&" characters. Therefore the operands are not allowed to contain "=", "&" and, because of URL-encoding, "%" characters. These characters are considered "unsafe" and need to be encoded as:

Character	URL-Encoding Syntax
%	%25
&	%26
=	%3D

General Note for URL-Encoding

When sending POST data with the content type **application/x-www-form-urlencoded**, certain characters must be represented by means of URL-encoding, which means substituting the character with %hexadecimal-character-code. The full details of when and why URL-encoding is necessary, are discussed in RFC 1630, RFC 1738 and RFC 1808. Some basic details are given here. All non-ASCII characters (i.e., valid ISO 8859/1 characters that are not also ASCII characters) must be URL-encoded, e.g., the file *köln.html* would appear in an URL as *k%F6ln.html*.

Some characters are considered to be "unsafe" when web pages are requested by e-mail.

These characters are:

Character	URL-Encoding Syntax
the tab character	%09
the space character	%20
[	%5B
\	%5C
]	%5D
^	%5E
‘	%60
{	%7B
|	%7C
}	%7D
~	%7E

When writing URLs, you should URL-encode these characters.

Some characters have special meanings in URLs, such as the colon (:) that separates the URL scheme from the rest of the URL, the double slash (//) that indicates that the URL conforms to the Common Internet Scheme syntax and the percent sign (%). Generally, when these characters appear as parts of file names, they must be URL-encoded to distinguish them from their special meaning in URLs (this is a simplification, read the RFCs for full details).

These characters are:

 Copyright © Software AG 2002

Character	URL-Encoding Syntax
"	%22
#	%23
%	%25
&	%26
+	%2B
,	%2C
/	%2F
:	%3A
<	%3C
=	%3D
>	%3E
?	%3F
@	%40

operand9

Operand9 contains all header values delivered with the HTTP response.

The first line contains the status information and all following lines contain the headers as pairs of name and value. The names always end in a colon (:) and the values end in a carriage return (CR). (Internally, all "CR/LF"s are transformed to "CR"s.)

operand10/11

Operand10 is the name of a HEADER received with this request. The HEADER is needed for HTTP.

Operand11 is the value of a HEADER received with this request. The HEADER is needed for HTTP.

Note:
Operand10 and operand11 can only be used in conjunction with each other.

Return Header Name for Operand10

For compatibility with the web interface, header names can be written with "_" instead of "-". Internally, "_" is replaced by "-".

If operand10 is a blank string, the status information is returned.

```
HTTP/1.0 200 OK
```

operand12

Operand12 is the document returned for this request.

operand13

Operand13 is the response number of the request (e.g. 200).

Overview of Response Numbers - for HTTP/HTTPs Requests

 Copyright © Software AG 2002

Status	Value	Response
STATUS CONTINUE	100	OK to continue with request
STATUS SWITCH_PROTOCOLS	101	Server has switched protocols in upgrade header
STATUS OK	200	Request completed
STATUS CREATED	201	Object created, reason = new URL
STATUS ACCEPTED	202	Async completion (TBS)
STATUS PARTIAL	203	Partial completion
STATUS NO_CONTENT	204	No info to return
STATUS RESET_CONTENT	205	Request completed, but clear form
STATUS PARTIAL_CONTENT	206	Partial GET fulfilled
STATUS AMBIGUOUS	300	Server could not decide what to return
STATUS MOVED	301	Object permanently moved
STATUS REDIRECT	302	Object temporarily moved
STATUS REDIRECT_METHOD	303	Redirection w/o new access method
STATUS NOT_MODIFIED	304	If-modified-since was not modified
STATUS USE_PROXY	305	Redirection to proxy, location header specifies proxy to use
STATUS REDIRECT_KEEP_VERB	307	HTTP/1.1: keep same verb
STATUS BAD_REQUEST	400	Invalid syntax
STATUS DENIED	401	Access denied
STATUS PAYMENT_REQ	402	Payment required
STATUS FORBIDDEN	403	Request forbidden
STATUS NOT_FOUND	404	Object not found
STATUS BAD_METHOD	405	Method is not allowed
STATUS NONE_ACCEPTABLE	406	No response acceptable to client found
STATUS PROXY_AUTH_REQ	407	Proxy authentication required
STATUS REQUEST_TIMEOUT	408	Server timed out waiting for request
STATUS CONFLICT	409	User should resubmit with more info
STATUS GONE	410	The resource is no longer available
STATUS LENGTH_REQUIRED	411	The server refused to accept request w/o a length
STATUS PRECOND_FAILED	412	Precondition given in request failed
STATUS REQUEST_TOO_LARGE	413	Request entity was too large
STATUS URL_TOO_LONG	414	Request URL too long
STATUS UNSUPPORTED_MEDIA	415	Unsupported media type
STATUS SERVER_ERROR	500	Internal server error
STATUS NOT_SUPPORTED	501	"Required" not supported
STATUS BAD_GATEWAY	502	Error response received from gateway
STATUS SERVICE_UNAVAIL	503	Temporarily overloaded
STATUS GATEWAY_TIMEOUT	504	Timed out waiting for gateway
STATUS VERSION_NOT_SUP	505	HTTP version not supported

Copyright © Software AG 2002

Response 301 - 303 (Redirection)

Redirection means that the requested URL has moved. As a response, the Return Header with the name LOCATION will be displayed. This header contains the URL where the requested page has moved to. A new REQUEST DOCUMENT request can be used to retrieve the page moved.

HTTP browsers redirect automatically to the new URL, but the REQUEST DOCUMENT statement does not handle redirection automatically.

Response 401 (Denied)

The response "Access Denied" means that the requested page can only be accessed if a valid user ID and password are provided with the request. As a response, the Return Header with the name WWW-AUTHENTICATE will be delivered with the realm needed for this request.

HTTP browsers normally display a dialog with user ID and password, but with the REQUEST DOCUMENT statement, no dialog is displayed.

operand14

Operand14 contains the Natural error if the request could not be performed.

Examples

Note:

There is an example dialog V5-RDOC for this statement in the example library SYSEXV.

General Request

```
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
```

Simple Get Request (no data)

```
REQUEST DOCUMENT FROM "http://pcnatweb:8080"
  RETURN
   PAGE #Resultxml
  RESPONSE #rc
```

Simple Head Request (no return page)

```
REQUEST DOCUMENT FROM "http://pcnatweb"
   RESPONSE #rc
```

 Copyright © Software AG 2002

Simple Post Request (default)

```
REQUEST DOCUMENT FROM "http://pcnatweb/cgi-bin/nwwcgi.exe/sysweb/nat-env"
  WITH
   DATA
   NAME 'XMLData'       VALUE #Queryxml
   NAME 'repServerName' VALUE 'NT2'
  RETURN
   PAGE #Resultxml
  RESPONSE #rc
```

Simple Put Request (with data all)

```
REQUEST DOCUMENT FROM "http://pcnatweb/test.txt"
  WITH
   DATA ALL  #document
  RETURN
   PAGE #Resultxml
  RESPONSE #rc
```

RESET

```
RESET [INITIAL] operand1 ...
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S	A	G	M	A	N	P	I	F	B	D	T	L	C	G	O	yes	yes

Function

The RESET statement is used to set the value of an operand(s) to a null value, or to an initial value as defined in a DEFINE DATA statement.

If *operand1* is a DYNAMIC variable, it will be reset to a null value with the length the variable currently has at the time the RESET statement is executed. The current length of a DYNAMIC variable can be ascertained by using the system variable *LENGTH. For general information on DYNAMIC variables, see your Natural User's Guide.

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

INITIAL

RESET (without INITIAL) sets the value of each specified field (operand1) to a null value.

RESET INITIAL sets each specified field to the initial value as defined for the field in the DEFINE DATA statement. If no initial value is defined for a field, it will be reset to a default initial value (see below).

If you apply RESET INITIAL to an array, it must be applied to the entire array (as defined in the DEFINE DATA statement); a RESET INITIAL of individual array occurrences is not possible.

RESET INITIAL of fields resulting from a redefinition is not possible either.

RESET INITIAL cannot be applied to database fields.

RESET INITIAL cannot be applied to DYNAMIC variables.

Default Initial Values

If you specify no INIT or CONST value in the DEFINE DATA statement, a field will be initialised with a default initial value depending on its format.

 Copyright © Software AG 2002

Example

```
/* EXAMPLE 'RSTEX1': RESET
/*******************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME (A10)
1 #BINARY  (B4) INIT <1>
1 #INTEGER (I4) INIT <5>
1 #NUMERIC (N2) INIT <25>
END-DEFINE
/*******************************************************
LIMIT 1
READ EMPLOY-VIEW
/*******************************************************
 WRITE NOTITLE 'VALUES BEFORE RESET STATEMENT:'
 WRITE / '=' NAME '=' #BINARY '=' #INTEGER '=' #NUMERIC
/*******************************************************
 RESET NAME #BINARY #INTEGER #NUMERIC
 WRITE /// 'VALUES AFTER  RESET STATEMENT:'
 WRITE / '=' NAME '=' #BINARY '=' #INTEGER '=' #NUMERIC
/*******************************************************
 RESET INITIAL #BINARY #INTEGER #NUMERIC
 WRITE /// 'VALUES AFTER  RESET INITIAL STATEMENT:'
 WRITE / '=' NAME '=' #BINARY '=' #INTEGER '=' #NUMERIC
/*******************************************************
END-READ
END
```

```
VALUES BEFORE RESET STATEMENT:

  NAME: MORENO     #BINARY: 00000001 #INTEGER:           5 #NUMERIC:  25



  VALUES AFTER  RESET STATEMENT:

  NAME:            #BINARY: 00000000 #INTEGER:           0 #NUMERIC:   0



  VALUES AFTER  RESET INITIAL STATEMENT:

  NAME:            #BINARY: 00000001 #INTEGER:           5 #NUMERIC:  25
```

RETRY

Note:
This statement can only be used when accessing Adabas databases.

```
RETRY
```

Related Statements: FIND | READ

Function

The RETRY statement is used within an ON ERROR statement block (see ON ERROR statement). It is used to reattempt to obtain a record which is in hold status for another user.

When a record to be held is already in hold status for another user, Natural issues error message 3145. See also the session parameter WH in the Natural Reference documentation.

The RETRY statement must be placed in the object that causes the error 3145.

For details on records hold logic, see the section Database Access of the Natural Programming Guide.

 Copyright © Software AG 2002

Example

```
/* EXAMPLE 'RTYEX1S': RETRY (STRUCTURED MODE)
/*******************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
1 #RETRY (A1) INIT <' '>
END-DEFINE
/*******************************************
FIND EMPLOY-VIEW WITH NAME = 'ALDEN'
 DELETE
 END TRANSACTION
/*******************************************
 ON ERROR
  IF *ERROR-NR = 3145
   INPUT NO ERASE 10/1
                 'RECORD IS IN HOLD' /
                 'DO YOU WISH TO RETRY ' /
                 #RETRY '(Y)ES OR (N)O '
   IF #RETRY = 'Y'
    RETRY
   ELSE
    STOP
   END-IF
  END-IF
 END-ERROR
/*******************************************
 AT END OF DATA
  WRITE NOTITLE *NUMBER 'RECORDS DELETED'
 END-ENDDATA
END-FIND
/*******************************************
END
```

Equivalent reporting-mode example: See the program RTYEX1R in the library SYSEXRM.

RUN

```
RUN [REPEAT] operand1 [operand2 [(parameter)]]...40
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S				A												yes	no
Operand2	C	S	A	G		A	N	P	I	F	B	D	T	L		G		yes	no

Function

The RUN statement is used to read a Natural source program from the Natural system file and then execute it.

REPEAT

RUN REPEAT causes the program not to prompt the user for input until the program has finished executing even if multiple output screens (produced by INPUT statements) are produced.

This feature may be used if the program is to display multiple screens of information without having the user respond to each screen.

Program Name - operand1

The name of the program can be specified as an alphanumeric constant or as the content of an alphanumeric variable. If a variable is used, it must be 8 characters in length.

The program may be stored in the current library or in a concatenated library (default steplib is SYSTEM). If the program is not found, an error message is issued.

The program is read into the source program work area and overlays any current source program.

Parameters - operand2

The RUN statement may also be used to pass parameters to the program to be run. A parameter may be defined with any format. The parameters are converted to a format suitable for a corresponding INPUT field. All parameters are placed on the top of the Natural stack.

The parameters can be read using an INPUT statement. The first INPUT statement issued will result in the insertion of all parameters into the fields specified in the INPUT statement. The INPUT statement must have the sign specification (SG=ON) for parameter fields defined with numeric format.

 Copyright © Software AG 2002

If more parameters are passed than are read by the next INPUT statement, the extra parameters are ignored. The number of parameters may be obtained with the system variable *DATA.

Note:
If operand2 is a time variable (format T), only the time component of the variable content is passed, but not the date component.

parameter

If *operand2* is a date variable, you can specify the session parameter DF as *parameter* for this variable. The session parameter DF is described in the Natural Reference documentation.

Dynamic Source Text Creation/Execution

The RUN statement may be used to dynamically compile and execute a program for which the source or parts thereof are created dynamically.

Dynamic source text creation is performed by placing source text into global variables and then referring to these variables by using "&" instead of "+" as the first character of the variable name in the source text. The content of the global variable will be interpreted as source text when the program is invoked using the RUN statement.

A global variable with index must not be used within a program that is invoked via a RUN statement.

It is not allowed to place a comment or an INCLUDE statement in a global variable.

Example

Program containing RUN statement:

```
/* EXAMPLE 'RUNEX1': RUN (WITH DYNAMIC SOURCE PROGRAM CREATION)
/*************************************************************
/* GLOBAL AREA 'GDA1' CONTAINS '+CRITERIA'
/* +CRITERIA (A80)
/* PROGRAM CREATING SOURCE
DEFINE DATA GLOBAL USING GDA1
LOCAL
1 #NAME (A20)
1 #CITY (A20)
END-DEFINE
/*************************************************************
INPUT 'Please specify the search values:' /
      'Name:' #NAME /
      'City:' #CITY
RESET +CRITERIA
/*************************************************************
IF #NAME = ' ' AND #CITY = ' '
  REINPUT 'Enter at least 1 value'
END-IF
/*************************************************************
IF #NAME NE ' '
  COMPRESS 'NAME' ' =''' #NAME '''' INTO +CRITERIA LEAVING NO
END-IF
IF #CITY NE ' '
  IF +CRITERIA NE ' '
    COMPRESS +CRITERIA 'AND' INTO +CRITERIA
  END-IF
  COMPRESS +CRITERIA ' CITY =''' #CITY '''' INTO +CRITERIA
END-IF
/*************************************************************
RUN 'FIND-EMP'
/*************************************************************
END
```

Program FIND-EMP executed by RUN statement:

```
/* PROGRAM EXECUTED WITH RUN STATEMENT ('FIND-EMP')
/***************************************************
DEFINE DATA GLOBAL USING GDA1
 LOCAL
 1 EMPLOY-VIEW VIEW OF EMPLOYEES
 2 NAME
 2 CITY
END-DEFINE
/***************************************************
FIND NUMBER EMPLOY-VIEW WITH &CRITERIA
                        RETAIN AS 'EMP-SET'
  DISPLAY *NUMBER
END
```

 Copyright © Software AG 2002

SEND EVENT

Note:
This statement is only available under Windows and Windows NT.

```
SEND EVENT operand1 TO [DIALOG-ID] { operand2 }
                                   { *DIALOG-ID }

[ { WITH { operand3 [(AD = { M } )] }...
         { nX       [      { O }  ]
         {                 { A }  }
  { USING [DIALOG] 'dialog-name' WITH PARAMETERS-clause } ]
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S				A												yes	no
Operand2		S							I									yes	no
Operand3	C	S	A			A	N	P	I	F	B	D	T	L	C	G	O	yes	no

Function

You use this statement to trigger a user-defined event within a Natural application.

Operands

Operand1 is the name of the event to be sent.

Operand2 is the identifier of the dialog receiving the user event. *Operand2* must be defined with format/length I4.

AD=

If operand3 is a variable, you can mark it in one of the following ways:

AD=O	non-modifiable
AD=M	modifiable
AD=A	input only

The default setting for AD= is AD=M.

Operand3 cannot be explicitly specified if operand3 is a constant. AD=O always applies to constants.

AD=M

By default, the passed value of a parameter can be changed in the dialog and the changed value passed back to the invoking object, where it overwrites the original value.

Exception: For a field defined with BY VALUE in the dialogs's parameter data area, no value is passed back.

AD=O

If you mark a parameter with AD=O, the passed value can be changed in the dialog, but the changed value cannot be passed back to the invoking object; that is, the field in the invoking object retains its original value.

Note:
Internally, AD=O is processed in the same way as BY VALUE (see the section parameter-data-definition of the DEFINE DATA statement).

AD=A

If you mark a parameter with AD=A, its value will not be passed to the dialog, but it will receive a value from the dialog. AD=A fields will be reset to empty before the event is sent.

For a field defined with BY VALUE in the dialog's parameter data area, the invoking object cannot receive a value. In this case, AD=A only causes the field to be reset to empty before the event is sent.

Passing Parameters to the Dialog

It is possible to pass parameters to the dialog.

As *operand3* you specify the parameter(s) to be passed to the dialog.

With the *PARAMETERS-clause*, parameters may be passed selectively.

nX

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 dialog.
A parameter that is to be skipped must be defined with the keyword OPTIONAL in the dialog's DEFINE DATA PARAMETER statement. OPTIONAL means that a value can - but need not - be passed from the invoking object to such a parameter.

PARAMETERS-clause

```
PARAMETERS {parameter-name = operand3}... END-PARAMETERS
```

Note:
You can only use the PARAMETERS-clause if the specified target dialog (dialog-name) is cataloged.

 Copyright © Software AG 2002

Dialog-name is the name of the dialog receiving the user event.

Note:
If the value of a parameter marked with AD=O and passed "by reference" is changed in a dialog, this will lead to a runtime error.

Further Information and Examples

See the section Event-Driven Programming Techniques in the Natural User’s Guide for Windows.

SEND METHOD

```
SEND [METHOD] operand 1 TO [OBJECT] operand2
                                  [ WITH { operand3 [ (AD = { M } ) ] } ]
                                                     { O }
                                                     { A }
                                         { nX                    } ...
                                  [RETURN operand4]
                                  [GIVING operand5]
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S				A												yes	no
Operand2		S															O	no	no
Operand3	C	S	A	G		A	N	P	I	F	B	D	T	L	C	G	O	yes	no
Operand4		S	A			A	N	P	I	F	B	D	T	L	C	G	O	yes	no
Operand5		S			N				I									yes	no

The formats C and G can only be passed to methods of local classes. For more information, see the section Local Classes.

Function

The SEND METHOD statement is used to invoke a particular method of an object.

Note: Optional parameters (*n*X notation) are available with Version 4.1.1 and all subsequent releases. The AD option is available with Version 4.1.2 and all subsequent releases.

 Copyright © Software AG 2002

Method-Name - operand1

Operand1 is the name of a method which is supported by the object specified in *operand2*.

Since the method names can be identical in different interfaces of a class, the method name in *operand1* can also be qualified with the interface name to avoid ambiguity.

In the following example, the object #O3 has an interface Iterate with the method Start. The following statements apply:

```
* Specifying only the method name.
  SEND "Start" TO #O3
  * Qualifying the method name with the interface name.
  SEND "Iterate.Start" TO #O3
```

If no interface name is specified, Natural searches the method name in all the interfaces of the class. If the method name is found in more than one interface, a runtime error occurs.

Object Handle - operand2

The handle of the object to which the method call is to be sent.

Operand2 must be defined as an object handle (HANDLE OF OBJECT). The object must already exist.

To invoke a method of the current object inside a method, use the system variable *THIS-OBJECT.

Parameter - operand3

As *operand3* you can specify parameters specific to the method.

In the following example, the object #O3 has the method PositionTo with the parameter Pos. The method is called in the following way:

```
  SEND "PositionTo" TO #O3 WITH Pos
```

Methods can have optional parameters. Optional parameters need not to be specified when the method is called. To omit an optional parameter, use the placeholder 1X. To omit n optional parameters, use the placeholder *n*X.

In the following example, the method SetAddress of the object #O4 has the parameters FirstName, MiddleInitial, LastName, Street and City, where MiddleInitial, Street and City are optional. The following statements apply:

```
* Specifying all parameters.
  SEND "SetAddress" TO #O4 WITH FirstName MiddleInitial LastName Street City
  * Omitting one optional parameter.
  SEND "SetAddress" TO #O4 WITH FirstName 1X LastName Street City
  * Omitting all optional parameters.
  SEND "SetAddress" TO #O4 WITH FirstName 1X LastName 2X
```

Omitting a non-optional (mandatory) parameter results in a runtime error.

AD=

If *operand3* is a variable, you can mark it as non-modifiable (AD=O), as modifiable (AD=M) or as for input only (AD=A). The default is AD=M.

If *operand3* is a constant, AD cannot be explicitly specified. For constants AD=O always applies.

AD=M

This is the default, the passed value of a parameter can be changed in the method and the changed value is passed back to the caller, where it overwrites the original value.

If a method is implemented in Natural and the parameter is defined with BY VALUE in the methodŸs parameter data area, no value is passed back.

If a method is not implemented in Natural, the behavior depends on the method implementation. The parameter is then passed BY REFERENCE. Whether the external component accepts a by reference or by value parameter is described in the documentation of the external component. It can also be viewed in the Natural Component Browser.

AD=O

If you mark a parameter with AD=O, the passed value can be changed in the method, but the changed value is not passed back to the invoking object. The caller retains its original value.

If a method is implemented in Natural, the parameter is treated like it was defined BY VALUE in the methodŸs parameter data area (see the section PARAMETER Clause in the description of the INTERFACE statement).

If a method is not implemented in Natural, the behavior depends on the method implementation. The parameter is then passed BY VALUE. Whether the external component accepts a by reference or by value parameter is described in the documentation of the external component. It can also be viewed in the Natural Component Browser.

AD=A

If you mark a parameter with AD=A, its value will not be passed to the method, but it will receive a value from the method. AD=A fields will be reset to empty before the method is invoked.

For a field defined with BY VALUE in the method’s parameter data area the caller cannot receive a value. In this case, AD=A only causes the field to be reset to empty before the method is invoked.

If a method is not implemented in Natural, the behavior depends on the method implementation. The parameter is then passed as an initialized variant. Whether the external component is able to return a value is described in the documentation of the external component. It can also be viewed in the Natural Component Browser.

Parameter - nX

This notation is not available on mainframe computers.

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

For a method implemented in Natural, a parameter that is to be skipped must be defined with the keyword OPTIONAL in the method subprogram’s DEFINE DATA PARAMETER statement. OPTIONAL means that a value can - but need not - be passed from the invoking object to such a parameter.

 Copyright © Software AG 2002

RETURN - operand4

If the RETURN clause is omitted and the method has a return value, the return value is discarded.

If the RETURN clause is specified, *operand4* contains the return value of the method. If the method execution fails, *operand4* is reset to its initial value.

Note:
For classes written in Natural, the return value of a method is defined by entering one additional parameter in the parameter data area of the method and by marking it with 'BY VALUE RESULT'. (For more information, see the section PARAMETER Clause.) Therefore the parameter data area of a method that is written in Natural and that has a return value always contains one additional field next to the method parameters. This is to be considered when you call a method of a Natural written class and want to use the parameter data area of the method in the SEND statement.

GIVING - operand5

If the GIVING clause is not specified, the Natural run time error processing is triggered if an error occurs.

If the GIVING clause is specified, *operand5* contains the Natural message number if an error occurred, or zero on success.

SEPARATE

```
SEPARATE { operand1                                  }
         { SUBSTRING (operand1,operand2,operand3)    }
         [LEFT [JUSTIFIED]] INTO operand4 ...
         [ IGNORE                  ]
         [ REMAINDER operand5      ]
         [ WITH [RETAINED] { [ANY] DELIMITERS }  ]
         [                 { INPUT DELIMITERS }  ]
         [                 { DELIMITERS operand6 } ]
         [[GIVING] NUMBER [IN] operand7]
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S				A											yes	no
Operand2	C	S					N	P	I								yes	no
Operand3	C	S					N	P	I								yes	no
Operand4		S	A	G		A											yes	yes
Operand5		S				A											yes	yes
Operand6	C	S				A											yes	no
Operand7		S					N	P	I								yes	yes

Related Statements: COMPRESS | EXAMINE

Function

The SEPARATE statement is used to separate the content of an alphanumeric operand into two or more alphanumeric operands (or into multiple occurrences of an alphanumeric array).

Source Operand - operand1

Operand1 is the alphanumeric constant or variable whose content is to be separated.

Trailing blanks in *operand1* are removed before the value is processed (even if the blank is used as a delimiter character; see also the DELIMITER option).

 Copyright © Software AG 2002

SUBSTRING

Normally, the whole content of a field is separated, starting from the beginning of the field. The SUBSTRING option allows you to separate only a certain part of the field. After the field name (*operand1*) in the SUBSTRING clause you specify first the starting position (*operand2*) and then the length (*operand3*) of the field portion to be separated. For example, if a field #A contained "CONTRAPTION", SUBSTRING(#A,5,3) would contain "RAP".

Note:
If you omit operand2, the starting position is assumed to be "1". If you omit operand3, the length is assumed to be from the starting position to the end of the field.

LEFT JUSTIFIED

This option causes leading blanks which may occur between the delimiter character and the next non-blank character to be removed from the target operand.

Target Operand - operand4

Operand4 represents the target operands. If an array is specified as target operand, it is filled occurrence by occurrence with the separated values.

The number of target operands corresponds to the number of delimiter characters (including trailing delimiter characters) in *operand1*, plus 1.

If *operand4* is a DYNAMIC variable, its length may be modified by the SEPARATE operation. The current length of a DYNAMIC variable can be ascertained by using the system variable *LENGTH. For general information on DYNAMIC variables, see your Natural User's Guide.

IGNORE / REMAINDER

If you do not specify enough target fields for the source value to be separated into, you will receive an appropriate error message.

To avoid this, you have two options:

- If you specify IGNORE, Natural will ignore it if there are not enough target operands to receive the source value.
- If you specify REMAINDER *operand5*, that section of the source value which could not be placed into target operands will be placed into *operand5*. You may then use the content of *operand5* for further processing, for example in a subsequent SEPARATE statement.

DELIMITER Option

```
[ WITH [RETAINED] { [ANY] DELIMITERS }
                  { INPUT DELIMITERS }
                  { DELIMITERS operand6 } ]
```

Delimiter characters within *operand1* indicate the positions at which the value is to be separated.

- If you omit the DELIMITER option (or specify WITH ANY DELIMITER), a blank and any character which is neither a letter nor a numeric character will be treated as delimiter character.
- WITH INPUT DELIMITER indicates that the blank and the default delimiter character (as specified with the session parameter ID) is to be used as delimiter character.
- WITH DELIMITER *operand6* indicates that each of the specified characters (*operand6*) is to be treated as delimiter character. If *operand6* contains trailing blanks, these will be ignored.

WITH RETAINED DELIMITERS

Normally, the delimiter characters themselves are not moved into the target operands.

When you specify RETAINED, however, each delimiter (that is, either default delimiters and blanks, or the delimiter specified with *operand6*) will also be placed into a target operand.

Example:

The following SEPARATE statement would place "150" into #B, "+" into #C, and "30" into #D:

```
...
MOVE '150+30' TO #A
SEPARATE #A INTO #B #C #D WITH RETAINED DELIMITER '+'
...
```

GIVING NUMBER

The GIVING NUMBER option causes the number of filled target operands (including those filled with blanks) to be returned in *operand7*. The number actually obtained is the number of delimiters plus 1.

If you use the IGNORE option, the maximum possible number returned in *operand7* will be the number of target operands (*operand4*).

If you use the REMAINDER option, the maximum possible number returned in *operand7* will be the number of target operands (*operand4*) plus *operand5*.

 Copyright © Software AG 2002

Example 1

```
/* EXAMPLE 'SEPEX1': SEPARATE
/*******************************************************************
DEFINE DATA LOCAL
1 #TEXT1 (A6) INIT <'AAABBB'>
1 #TEXT2 (A7) INIT <'AAA BBB'>
1 #TEXT3 (A7) INIT <'AAA-BBB'>
1 #TEXT4 (A7) INIT <'A.B/C,D'>
1 #FIELD1A (A6)
1 #FIELD1B (A6)
1 #FIELD2A (A3)
1 #FIELD2B (A3)
1 #FIELD3A (A3)
1 #FIELD3B (A3)
1 #FIELD4A (A3)
1 #FIELD4B (A3)
1 #FIELD4C (A3)
1 #FIELD4D (A3)
1 #NBT (N1)
1 #DEL (A5)
END-DEFINE
/*****************************************************************
WRITE NOTITLE 'EXAMPLE A (SOURCE HAS NO BLANKS)'
SEPARATE #TEXT1 INTO #FIELD1A #FIELD1B GIVING NUMBER #NBT
WRITE     / '=' #TEXT1 5X '=' #FIELD1A 4X '=' #FIELD1B 4X '=' #NBT
/*****************************************************************
WRITE NOTITLE /// 'EXAMPLE B (SOURCE HAS EMBEDDED BLANK)'
SEPARATE #TEXT2 INTO #FIELD2A #FIELD2B GIVING NUMBER #NBT
WRITE     / '=' #TEXT2 4X '=' #FIELD2A 7X '=' #FIELD2B 7X '=' #NBT
/*****************************************************************
WRITE NOTITLE /// 'EXAMPLE C (USING DELIMITER ''-'')'
SEPARATE #TEXT3 INTO #FIELD3A #FIELD3B WITH DELIMITER '-'
WRITE     /    '=' #TEXT3 4X '=' #FIELD3A 7X '=' #FIELD3B
/*****************************************************************
MOVE ',/' TO #DEL
WRITE NOTITLE /// 'EXAMPLE D USING DELIMITER' '=' #DEL
SEPARATE #TEXT4 INTO #FIELD4A #FIELD4B
                     #FIELD4C #FIELD4D WITH DELIMITER #DEL
WRITE     /    '=' #TEXT4 4X '=' #FIELD4A 7X '=' #FIELD4B
          /              19X '=' #FIELD4C 7X '=' #FIELD4D
/*****************************************************************
END
```

```
EXAMPLE A (SOURCE HAS NO BLANKS)

  #TEXT1: AAABBB     #FIELD1A: AAABBB    #FIELD1B:           #NBT:  1



  EXAMPLE B (SOURCE HAS EMBEDDED BLANK)

  #TEXT2: AAA BBB    #FIELD2A: AAA       #FIELD2B: BBB       #NBT:  2



  EXAMPLE C (USING DELIMITER '-')

  #TEXT3: AAA-BBB    #FIELD3A: AAA       #FIELD3B: BBB



  EXAMPLE D USING DELIMITER #DEL: ,/

  #TEXT4: A.B/C,D    #FIELD4A: A.B       #FIELD4B: C
                     #FIELD4C: D         #FIELD4D:
```

Example 2

```
   /* EXAMPLE 'SEPEX2': SEPARATE (USING AN ARRAY)
   /*********************************************************
   DEFINE DATA LOCAL
   1 #INPUT-LINE (A60) INIT <'VALUE1,   VALUE2,VALUE3'>
   1 #FIELD (A20/1:5)
   1 #NUMBER (N2)
   END-DEFINE
   /*********************************************************
   SEPARATE #INPUT-LINE LEFT JUSTIFIED INTO #FIELD (1:5)
                        GIVING NUMBER IN #NUMBER
   WRITE NOTITLE #INPUT-LINE //
                 #FIELD (1)  /
                 #FIELD (2)  /
                 #FIELD (3)  /
                 #FIELD (4)  /
                 #FIELD (5)  /
                 #NUMBER
   /*********************************************************
   END
```

```
VALUE1,   VALUE2,VALUE3

  VALUE1
  VALUE2
  VALUE3


    3
```

 Copyright © Software AG 2002

Example 3

```
/* EXAMPLE 'SEPEX3': SEPARATE (REMAINDER, RETAIN)
/*******************************************************
DEFINE DATA LOCAL
1 #INPUT-LINE (A60) INIT <'VAL1,   VAL2, VAL3,VAL4'>
1 #FIELD (A10/1:4)
1 #REM (A30)
END-DEFINE
WRITE TITLE LEFT 'INP:' #INPUT-LINE /
            '#FIELD (1)' 13T '#FIELD (2)' 25T '#FIELD (3)'
        37T '#FIELD (4)' 49T 'REMAINDER'
       /    '----------' 13T '----------' 25T '----------'
        37T '----------' 49T '------------------------------'
/*******************************************************
SEPARATE #INPUT-LINE INTO #FIELD (1:2)
         REMAINDER #REM WITH DELIMITERS ','
WRITE #FIELD(1) 13T #FIELD(2) 25T #FIELD(3) 37T #FIELD(4) 49T #REM
/*******************************************************
RESET #FIELD(*) #REM
SEPARATE #INPUT-LINE INTO #FIELD (1:2)
         IGNORE WITH DELIMITERS ','
WRITE #FIELD(1) 13T #FIELD(2) 25T #FIELD(3) 37T #FIELD(4) 49T #REM
/*******************************************************
RESET #FIELD(*) #REM
SEPARATE #INPUT-LINE INTO #FIELD (1:4) IGNORE
        WITH RETAINED DELIMITERS ','
WRITE #FIELD(1) 13T #FIELD(2) 25T #FIELD(3) 37T #FIELD(4) 49T #REM
/*******************************************************
RESET #FIELD(*) #REM
SEPARATE SUBSTRING(#INPUT-LINE,1,50) INTO #FIELD (1:4)
         IGNORE WITH DELIMITERS ','
WRITE #FIELD(1) 13T #FIELD(2) 25T #FIELD(3) 37T #FIELD(4) 49T #REM
/*******************************************************
END
```

```
INP: VAL1,   VAL2, VAL3,VAL4
  #FIELD (1)  #FIELD (2)  #FIELD (3)  #FIELD (4)  REMAINDER
  ----------  ----------  ----------  ----------  ------------------------------
  VAL1           VAL2                             VAL3,VAL4
  VAL1           VAL2
  VAL1        ,            VAL2       ,
  VAL1           VAL2      VAL3       VAL4
```

SET CONTROL

```
SET CONTROL operand1
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1	C	S				A											yes	no

Function

The SET CONTROL statement is used to perform terminal commands from within a program.

The terminal commands (*operand1*) are specified without the control character and can be specified as a text constant or as the content of an alphanumeric variable.

For further information on terminal commands, see the Natural Reference documentation.

Example 1

```
...
SET CONTROL 'L'
...
```

Switches to lower case (equivalent to the terminal command %L).

Example 2

```
...
SET CONTROL 'HDEST'
...
```

Activates hardcopy output to destination "DEST" (equivalent to the terminal command %H*destination*).

 Copyright © Software AG 2002

SET GLOBALS

Note:
This statement is permitted in reporting mode only.

Instead of this statement, you can use the system command GLOBALS.

```
SET GLOBALS parameter ...
```

Function

The SET GLOBALS statement is used to set values for session parameters.

The parameters are evaluated either when the program that contains the SET GLOBALS statement is compiled, or when it is executed; this depends on the individual parameters.

The parameter settings specified with SET GLOBALS remain in effect until the end of the Natural session, unless they are overridden with a subsequent SET GLOBALS statement (or GLOBALS system command).

Exception: On mainframe computers, a SET GLOBALS statement in a subordinate program (that is, a subroutine, subprogram, or program invoked with FETCH RETURN) only applies until control is returned from the subordinate program to the invoking object; then the parameter values set for the invoking object apply again.

Parameters

If you specify multiple parameters, you have to separate them from one another by one or more blanks. The parameters can be specified in any order.

For information on the individual parameters, see the section Session Parameters in the Natural Reference documentation.

Example

```
SET GLOBALS LS=74 LT=5000
...
```

SET KEY

Syntax 1 - Affecting All Keys

```
                 ALL
                 ON
                 OFF
SET KEY {                 }
         COMMAND {ON}
                 {OFF}
         NAMED OFF
```

Syntax 2 - Affecting Individual Keys

```
                 PAn                                  ON
                 PFn                                  OFF
SET KEY {{       CLR       }  [ = {                 DISABLED  }  ]}...
         {DYNAMIC operand1}      {COMMAND {ON}}
                                          {OFF}
```

 Copyright © Software AG 2002

Syntax 3 - Affecting Individual Keys

```
SET KEY { { PAn | PFn | CLR | DYNAMIC operand1 } [ = { PGM | operand2 | HELP | DATA operand3 } ] [ NAMED { operand4 | OFF } ]
          ENTR NAMED { operand4 | OFF } } ...
```

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand1		S				A											yes	no
Operand2	C	S				A											yes	no
Operand3	C	S				A											yes	no
Operand4	C	S				A											yes	no

Function

The SET KEY statement is used to assign functions to video terminal PA (program attention) keys, PF (program function) keys, and the CLEAR key.

When a SET KEY statement is executed, Natural receives control of the keys during program execution and uses the values assigned to the keys.

The Natural system variable *PF-KEY identifies which key was pressed last.

Notes:
If a user presses a key to which no function is assigned, either a warning message will be issued prompting the user to press a valid key, or the value "ENTR" will be placed into the Natural system variable *PF-KEY, i.e. Natural will react as if the ENTER key had been pressed (this depends on the Natural profile parameter IKEY as set by the Natural administrator).
On mainframe computers, processing of PA and PF keys is also affected by the Natural profile parameter KEY as set by the Natural administrator.

Making Keys Program-Sensitive

Making a key program-sensitive means that the key will be available for interrogation by the currently active program. If a key is made program-sensitive, pressing the key has the same effect as pressing ENTER. All data that have been entered on the screen are transferred to the program.

Note:
PA keys and the CLEAR key, when made program-sensitive, do not cause any data to be transferred from the screen.

The program-sensitivity remains in effect only for the execution of the current program. See also the section SET KEY Statements on Different Program Levels.

Examples:

SET KEY ALL

This statement causes all keys to be made program-sensitive. All function assignments to any keys are overwritten.

SET KEY PF2
SET KEY PF2=PGM

Each of these statements cause PF2 to be made program-sensitive.

SET KEY OFF

This statement de-activates all key settings. (Under Com-plete on mainframe computers, control of all keys is returned to the TP monitor.)

SET KEY ON

This statement re-activates the functions assigned to all keys that had an assignment and re-activates the program-sensitivity of keys that were made program-sensitive before they were de-activated.

SET KEY PF2=OFF

This statement de-activates PF2. (Under Com-plete on mainframe computers, control of the PF2 key is returned to the TP monitor.)

SET KEY PF2=ON

This statement re-activates the function assigned to PF2 before it was de-activated or made program-sensitive. If no function had been assigned to PF2, it will be made program-sensitive again.

Assigning Commands/Programs

You can assign a command or program name to a key. When the key is pressed, the current program is terminated and the command/program assigned to the key is invoked via the Natural stack. When assigning a command/program, you can also pass parameters to the command/program (see third example below).

You can also assign a terminal command to a key. When the key is pressed, the terminal command assigned to the key is executed.

When operand2 is specified as a constant, it must be enclosed within apostrophes.

Examples:

The command "SAVE" is assigned to PF4:

SET KEY PF4 = ’SAVE’

The value contained in the variable #XYZ is assigned to PF4:

SET KEY PF4 = #XYX

 Copyright © Software AG 2002

The command "LIST", including the LIST parameters "MAP" and "*", is assigned to PF6:

SET KEY PF6 = ’LIST MAP *’

The terminal command "%%" is assigned to PF2:

SET KEY PF2=’%%’

The assignment remains in effect until it is overwritten by another SET KEY statement, until the user logs on to another application, or until the end of the Natural session. See also the section SET KEY Statements on Different Program Levels.

Note:
Before a program invoked via a key is executed, Natural internally issues a BACKOUT TRANSACTION statement.

Assigning Input DATA

You can assign a data string (*operand3*) to a key. When the key is pressed, the data string is placed into the input field in which the cursor is currently positioned, and the data are transferred to the executing program (as if ENTER had been pressed).

When *operand3* is specified as a constant, it must be enclosed within apostrophes.

Example:

SET KEY PF12=DATA ’YES’

For the validity of a DATA assignment, the same applies as for a command assignment, that is, it remains in effect until it is overwritten by another SET KEY statement, until the user logs on to another application, or until the end of the Natural session. See also the section SET KEY Statements on Different Program Levels.

COMMAND OFF/ON

With COMMAND OFF, you can temporarily de-activate any function (command, program, or data) assigned to a key. If the key had been program-sensitive before the function was assigned, COMMAND OFF will make it program-sensitive again.

With a subsequent COMMAND ON, you can re-activate the assigned function again.

Examples:

SET KEY PF4=COMMAND OFF

The function that has been assigned to PF4 is temporarily de-activated; if PF4 had been program-sensitive before the function was assigned, it is now made program-sensitive again.

SET KEY PF4=COMMAND ON

The function assigned to PF4 is re-activated again.

SET KEY COMMAND OFF

All functions assigned to all keys are temporarily de-activated; those keys which had been program-sensitive before functions were assigned to them, are now made program-sensitive again.

SET KEY COMMAND ON

All functions assigned to all keys are re-activated again.

With SET KEY PF*nn*=' ' you can delete the function assigned to a key and at the same time deactivate the program sensitivity of the key.

Assigning HELP

You can assign "HELP" to a key. When the key is pressed, the helproutine assigned to the field in which the cursor is currently positioned will be invoked.

The effect is the same as when entering the help character in the field to invoke help. (The help character - default is "?" - is determined by the Natural profile parameter HI as set by the Natural administrator.)

Example:

SET KEY PF1=HELP

For the validity of a HELP assignment, the same applies as for program-sensitivity, that is, it remains in effect only for the execution of the current program. See also the section SET KEY Statements on Different Program Levels.

DYNAMIC

Instead of specifying a specific key with the SET KEY statement, you can use the DYNAMIC option with a variable (*operand1*), and assign a value (PF*n*, PA*n*, CLR) to this variable in the program. This allows you to specify a function and make it dependent on the program logic which key this function is assigned to.

Example:

```
...
IF ...
   MOVE 'PF4' TO #KEY
ELSE
   MOVE 'PF7' TO #KEY
END-IF
...
SET KEY DYNAMIC #KEY = 'SAVE'
...
```

DISABLED

Graphical user interface (GUI) elements, such as push-buttons, menus, and bitmaps, are implemented as PF keys. With the DISABLED option, you can disable the use the of a GUI element assigned to a PF key. The GUI element will then be displayed in grey and is not available for selection.

To cancel the effect of SET KEY PF*nn*=DISABLED, you use SET KEY PF*nn*=ON.

Example:

SET KEY PF10=DISABLED

 Copyright © Software AG 2002

Disables the use of the GUI element assigned to PF10.

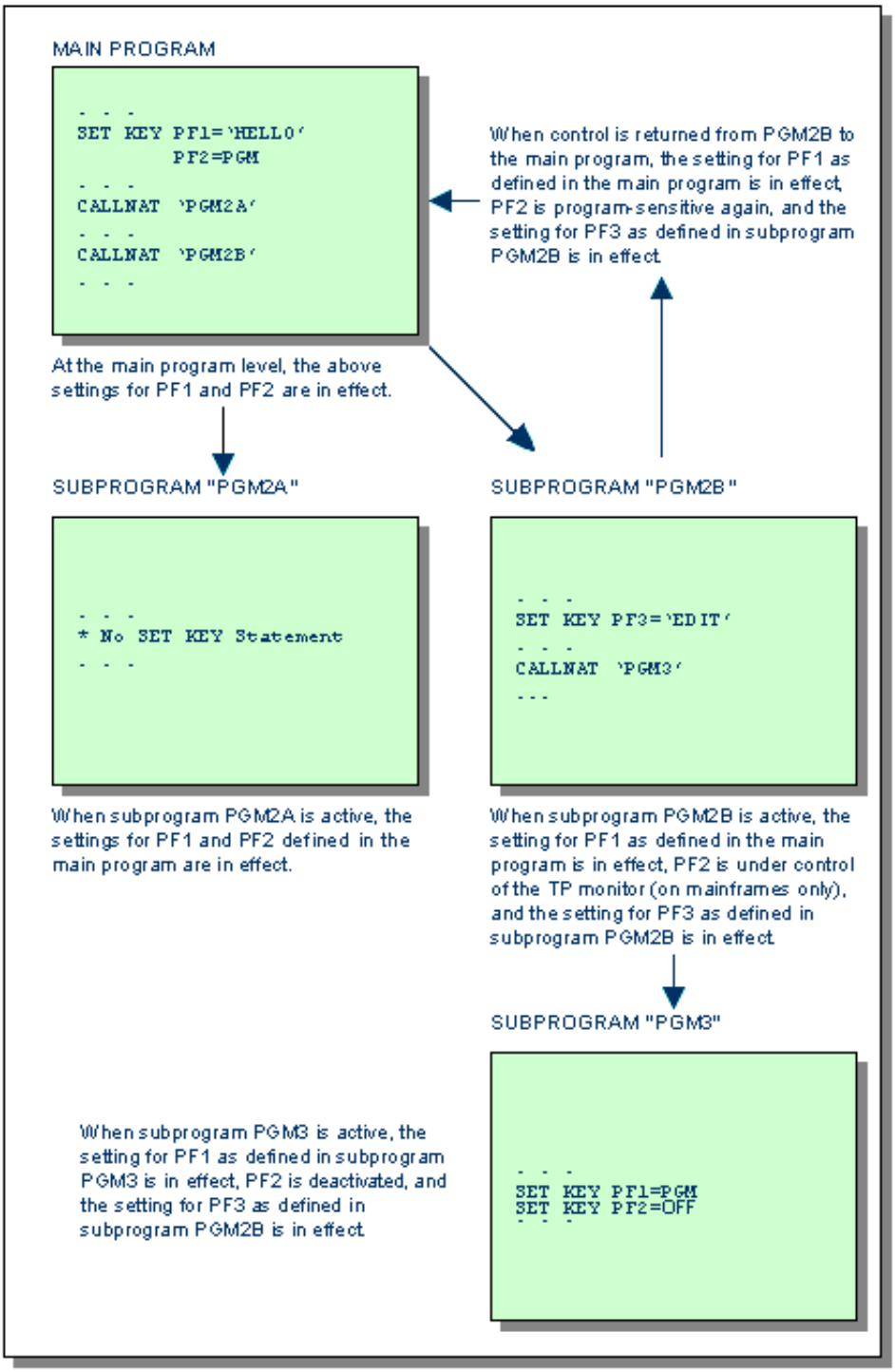
The DISABLED option can only be used within a processing rule.

SET KEY Statements on Different Program Levels

When an application contains SET KEY statements at different levels, the following applies:

- When keys are made program-sensitive, the program-sensitivity also applies to all lower level (called) programs, unless these programs contain further SET KEY statements. When control is returned to a higher level program, the SET KEY assignments made at the higher level come into effect again.
- For keys which are defined as HELP keys, the same applies as for keys which are program-sensitive.
- When a function (program, command, terminal command, or data string) is assigned to a key, this assignment is valid at all higher and lower levels - regardless of the level at what the assignment is made - until another function is assigned to the key or it is made program-sensitive, or until the user logs on to another application or the Natural session is terminated.

Example of SET KEY Statements on Different Program Levels:



Assigning Names

With the NAMED clause, you can assign a name (*operand4*) to a key. The name will then be displayed in the PF-key lines on the screen; this allows the users to identify the functions assigned to the keys:

 Copyright © Software AG 2002

```
              ?   Help
               .   Exit
              ---- ---------------------------------------------------
      Code ..: ?   Library ..: *_______
                   Object ...: *_____________________________________
                   DBID .....: 0__    FILENR ...: 0__


  Command ===>
 Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
       Help        Exit  Last        Flip                                Canc
```

The display of the PF-key lines is activated with the session parameter KD (see the Natural Reference documentation). You can control the way in which the PF-key lines are displayed by using the terminal command %Y (see the Natural Reference documentation).

The maximum length of a name to be assigned to a key is 10 characters. In normal tabular PF-key line format (%YN), only the first 5 characters are displayed.

When *operand4* is specified as a constant, it must be enclosed within apostrophes (see examples).

You cannot assign a name to a key without assigning a function to it or making it program-sensitive. To the ENTER key, however, you can only assign a name, but no function.

With NAMED OFF, you delete the name assigned to a program-sensitive key.

Examples:

SET KEY ENTR NAMED ’EXEC’

The name "EXEC" is assigned to the ENTER key.

SET KEY PF3 NAMED ’EXIT’

PF3 is made program-sensitive, and the name "EXIT" is assigned to PF3.

SET KEY PF3 NAMED OFF

PF3 is made program-sensitive, and the name that has been assigned to PF3 is deleted.

SET KEY NAMED OFF

All names that have been assigned to any program-sensitive keys are deleted.

SET KEY PF4=’AP1’ NAMED ’APPL1’

The program "AP1" and the name "APPL1" are assigned to PF4.

When you use normal tabular PF-key line format (%YN), the following applies:

- If you omit the NAMED clause when assigning a command/program to a key, the command/program name will be displayed in the PF-key line; if the command/program name is longer than 5 characters, "CMND" will be displayed.
- If you omit the NAMED clause when assigning input data to a key, "DATA" will be displayed in the PF-key line.

When you use sequential PF-key line format (%YS or %YP), only those keys to which names have been assigned will be displayed in the PF-key line; that is, if you omit the NAMED clause when assigning a command/program/data to a key, the key will not be displayed in the PF-key line.

Example

```
/* EXAMPLE 'SKYEX1': SET KEY
/*****************************************************************
DEFINE DATA LOCAL
1 #PF4 (A56)
1 #FCT (A8)
END-DEFINE
/*****************************************************************
MOVE 'LIST FILES' TO #PF4
/*****************************************************************
SET KEY PF1 PF2
SET KEY PF3 = 'MENU'
        PF4 = #PF4
        PF5 = 'LIST FILE EMPLOYEES'
/****************************************************************
INPUT 10X 'THE FOLLOWING FUNCTION KEYS ARE AVAILABLE:' //
      10X 'PF1: EMPLOYEES UPDATE PROGRAM' /
      10X 'PF2: EMPLOYEES READ   PROGRAM' /
      10X 'PF3: RETURN TO MENU          ' /
      10X 'PF4: LIST FILES              ' /
      10X 'PF5: LIST FILE EMPLOYEES     ' ///
      10X '     OR YOU MAY ENTER A PROGRAM TO BE EXECUTED:' #FCT
/****************************************************************
IF #FCT NE ' '
 FETCH #FCT
END-IF
IF *PF-KEY = 'PF1'
 FETCH 'UPDPERS'
END-IF
IF *PF-KEY = 'PF2'
 FETCH 'READPERS'
END-IF
/****************************************************************
END
```

```
THE FOLLOWING FUNCTION KEYS ARE AVAILABLE:

  PF1: EMPLOYEES UPDATE PROGRAM
  PF2: EMPLOYEES READ   PROGRAM
  PF3: RETURN TO MENU
  PF4: LIST FILES
  PF5: LIST FILE EMPLOYEES


       OR YOU MAY ENTER A PROGRAM TO BE EXECUTED:
```

 Copyright © Software AG 2002

SET TIME

```
{SETTIME}
{SET TIME}
```

Function

The SETTIME statement is used in conjunction with the Natural system variable *TIMD to measure the time it takes to execute a specific section of a program.

The SETTIME statement is placed at a specific position in the program, and *TIMD will contain the amount of time elapsed since the execution of the SETTIME statement.

*TIMD must always contain a reference to the SETTIME statement, either by using the source-code line number of the SETTIME statement or by assigning a label to the SETTIME statement which can then be used as a reference.

Example

```
/* EXAMPLE 'STIEX1': SETTIME
/************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
END-DEFINE
/************************************************************
ST. SETTIME
    WRITE 10X 'START TIME:' *TIME
    READ (100) EMPLOY-VIEW BY NAME
    END-READ
    WRITE NOTITLE 10X 'END TIME:  ' *TIME
    WRITE         10X 'ELAPSED TIME TO READ 100 RECORDS'
                  '(HH:SS:SS.T) :' *TIMD (ST.) (EM=99:99:99'.'9)
/************************************************************
END
```

```
        START TIME: 16:34:17.3
        END TIME:   16:34:24.0
        ELAPSED TIME TO READ 100 RECORDS (HH:II:SS.T) : 00:00:06.7
```

SET WINDOW

```
SET WINDOW { 'window-name' }
           {     OFF      }
```

Related Statements: DEFINE WINDOW | INPUT WINDOW='*window-name*'

Function

The SET WINDOW statement is used to activate and de-activate a window.

With SET WINDOW '*window-name*', you activate the specified window, which means that all subsequent statements refer to that window until either the window is de-activated or another window is activated. The specified window must have been defined with a DEFINE WINDOW statement.

With SET WINDOW OFF, you de-activate the currently active window.

Any SET WINDOW '*window-name*' or INPUT WINDOW='*window-name*' statement de-activates the window which has currently been active and activates the window specified in the statement. This means that only one window can be active at a time.

Note:
If you use SET WINDOW to activate a window which is defined with SIZE AUTO, the data on the screen **before** the window is activated determine the size of the window.

Example

See DEFINE WINDOW statement.

 Copyright © Software AG 2002

SKIP

```
SKIP [(rep)] operand1 [LINES]
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S					N	P	I									yes	no

Related Statements: DISPLAY | PRINT | WRITE

Function

The SKIP statement is used to generate one or more blank lines in an output report.

Report Specification - rep

The notation (*rep*) may be used to specify the identification of the report for which the SKIP statement is applicable. A value in the range 0 - 31 or a logical name which has been assigned using the DEFINE PRINTER statement may be specified.

If (*rep*) is not specified, the SKIP statement will apply to the first report (report 0).

Number of Lines to be Skipped - operand1

Operand1 represents the number (1 - 250) of blank lines to be generated. This number may be specified as a numeric constant or as the content of a numerical variable.

If *operand1* exceeds the page size of the report, the SKIP statement will result in a newpage condition.

Additional Considerations

If the execution of a SKIP statement would cause the page size to be exceeded, exceeding lines will be ignored (except in an AT TOP OF PAGE statement).

A SKIP statement is only executed if something has already been output on the page (output from an AT TOP OF PAGE statement is not taken into account here).

Copyright © Software AG 2002

Example

```
/* EXAMPLE 'SKPEX1': SKIP
/*****************************************************
LIMIT 7
READ EMPLOYEES BY CITY STARTING FROM 'W'
 AT BREAK OF CITY
  SKIP 2
 DISPLAY NOTITLE CITY (IS=ON) COUNTRY (IS=ON) NAME
/*****************************************************
END
```

```
      CITY          COUNTRY         NAME
-------------------- ------- --------------------

WASHINGTON            USA     REINSTEDT
                              PERRY


WEITERSTADT           D       BUNGERT
                              UNGER
                              DECKER


WEST BRIDGFORD        UK      ENTWHISTLE


WEST MIFFLIN          USA     WATSON
```

 Copyright © Software AG 2002

SORT

Structured Mode Syntax

```
END-ALL
[AND]
SORT [THEM | RECORDS] [BY] { operand1 [ASCENDING | DESCENDING] }...10
     USING-clause
     [GIVE-clause]
     statement...
END-SORT
```

* If a statement label is specified, it must be placed *before* the keyword SORT, but *after* END-ALL (and AND).

Reporting Mode Syntax

```
SORT [THEM | RECORDS] [BY] { operand1 [ASCENDING | DESCENDING] }...10
     [USING-clause]
     [GIVE-clause]
     statement...
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S				A	N	P	I	F	B	D	T					no	no

Related Statement: FIND with SORTED BY option

Function

The SORT statement is used to perform a sort operation, sorting the records from all processing loops that are active when the SORT statement is executed.

For the sort operation, Natural's internal sort program is used. On mainframe computers, it is also possible to use another, external sort program. The sort program to be used is determined by the Natural administrator in the macro NTSORT of the Natural parameter module (see also the Natural Operations for Mainframes documentation).

For the use of an external sort program, additional JCL is required; ask your Natural administrator for additional information.

Note:
Under OpenVMS and UNIX, the records that are to be sorted will be stored intermediately in the directory as specified under "TMP_PATH" in the configuration file "Natural.INI".

Restrictions

The SORT statement must be contained in the same object as the processing loops whose records it sorts.

Nested SORT statements are not allowed.

On mainframe computers, the total length of a record to be sorted must not exceed 10240 bytes.

Processing Loops

In reporting mode, the SORT statement closes all active processing loops and initiates a new processing loop.

In structured mode, the SORT statement must be preceded by END-ALL, which serves to close all active processing loops. The SORT statement itself initiates a new processing loop, which must be closed with END-SORT.

Sort Criteria - operand1

Operand1 represents the fields/variables to be used as the sort criteria. 1 to 10 database fields (descriptors and non-descriptors) and/or user-defined variables may be specified. A multiple-value field or a field contained within a periodic group may be used. A group or an array is not permitted.

The default sort sequence is ASCENDING. If you wish the values to be sorted in descending sequence, specify DESCENDING. ASCENDING/DESCENDING may be specified for each sort field.

USING-clause

```
{USING operand2 ...}
{  USING KEYS      }
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand2		S	A			A	N	P	I	F	B	D	T	L	C			no	no

The USING clause indicates the fields which are to be written to intermediate sort storage. It is required in structured mode and optional in reporting mode. However, it is strongly recommended to also use it in reporting mode so as to reduce memory requirements.

 Copyright © Software AG 2002

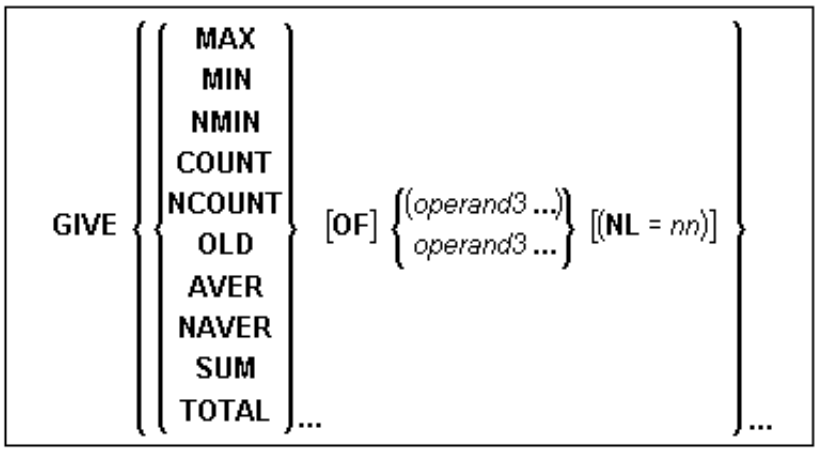
If you specify USING KEYS, only the sort key fields, as specified with *operand1*, will be written to intermediate sort storage.

With USING *operand2* you can specify additional fields that are to be written to intermediate sort storage - in addition to the sort key fields (as specified with *operand1*).

If you omit the USING clause in reporting mode, all database fields of processing loops initiated before the SORT statement, as well as all user-defined variables defined before the SORT statement, will be written to intermediate sort storage.

If, after sort execution, a reference is made to a field which was not written to sort intermediate storage, the value for the field will be the last value of the field before the sort.

GIVE-clause



Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
Operand3		S	A			*											yes	no

* depends on function

The GIVE clause is used to specify Natural system functions (MAX, MIN, etc.) that are to be evaluated in the first phase of the SORT statement. These system functions may be referenced in the third phase (see SORT Statement Processing). A reference to a system function after the SORT statement must be preceded by an asterisk, for example, *AVER (SALARY).

For details on the individual system functions, see the Natural Reference documentation.

(NL=nn)

This option may be used to prevent an arithmetic overflow during the evaluation of system functions; it is described under Arithmetic Overflows in AVER, NAVER, SUM or TOTAL in the section System Functions of the Natural Reference documentation.

This option applies to AVER, NAVER, SUM and TOTAL only and will be ignored for any other system function.

SORT Statement Processing

A program containing a SORT statement is executed in three phases.

1st Phase - Selecting the Records to be Sorted

The statements before the SORT statement are executed. Data as described in the USING clause will be written to intermediate sort storage.

In reporting mode, any variables to be used as accumulators following the sort must not be defined before the SORT statement. In structured mode, they must not be included in the USING clause. Fields written to intermediate sort storage cannot be used as accumulators because they are read back with each individual record during the 3rd processing phase. Consequently, the accumulation function would not produce the desired result because with each record the field would be overwritten with the value for that individual record.

The number of records written to intermediate storage is determined by the number of processing loops and the number of records processed per loop. One record on the internal intermediate storage is created each time the SORT statement is encountered in a processing loop.
In the case of nested loops, a record is only written to intermediate storage if the inner loop is executed. If in the example below a record is to be written to intermediate storage even if no records are found for the inner (FIND) loop, the FIND statement must contain an IF NO RECORDS FOUND clause.

```
READ ...
  ...
  FIND ...
...
END-ALL
SORT ...
  DISPLAY ...
END-SORT
...
```

2nd Phase - Sorting the Records

The records are sorted.

3rd Phase -Processing the Sorted Records

The statements after the SORT statement are executed for all records on the intermediate storage in the specified sorting sequence.
Database fields to be referenced after a SORT statement must be correctly referenced using the appropriate statement label or reference number.

Example

 Copyright © Software AG 2002

```
/* EXAMPLE 'SRTEX1R': SORT (REPORTING MODE)
/*************************************************************
LIMIT 3
FIND EMPLOYEES WITH CITY = 'BOSTON'
OBTAIN SALARY(1:2)
COMPUTE #TOTAL-SALARY (P11) = SALARY (1) + SALARY (2)
ACCEPT IF #TOTAL-SALARY GT 0
/*************************************************************
SORT BY PERSONNEL-ID USING #TOTAL-SALARY SALARY(*) CURR-CODE
     GIVE AVER(#TOTAL-SALARY)
/*************************************************************
AT START OF DATA
 DO
  WRITE NOTITLE
        '*' (40)
        'AVG CUMULATIVE SALARY:' *AVER (#TOTAL-SALARY) /
  MOVE *AVER (#TOTAL-SALARY) TO #AVG (P11)
 DOEND
COMPUTE #AVER-PERCENT (N3.2) = #TOTAL-SALARY / #AVG * 100
ADD #TOTAL-SALARY TO #TOTAL-TOTAL (P11)
DISPLAY NOTITLE PERSONNEL-ID SALARY (1) SALARY (2)
                #TOTAL-SALARY CURR-CODE (1)
                'PERCENT/OF/AVER' #AVER-PERCENT
AT END OF DATA
 WRITE / '*' (40) 'TOTAL SALARIES PAID: ' #TOTAL-TOTAL
/*************************************************************
END
```

```
PERSONNEL   ANNUAL     ANNUAL   #TOTAL-SALARY CURRENCY PERCENT
     ID       SALARY     SALARY                    CODE     OF
                                                         AVER
  --------- ---------- ---------- ------------- -------- -------

  **************************************** AVG CUMULATIVE SALARY:        41900

  20007000       16000      15200        31200  USD        74.00
  20019200       18000      17100        35100  USD        83.00
  20020000       30500      28900        59400  USD       141.00

  **************************************** TOTAL SALARIES PAID:        125700
```

The previous example is executed as follows:

First Phase:

- Records with CITY=BOSTON are selected from the EMPLOYEES file.
- The first 2 occurrences of SALARY are accumulated in the field #TOTAL-SALARY.
- Only records with #TOTAL-SALARY greater than 0 are accepted.
- The records are written to the sort intermediate storage. The database arrays SALARY (first 2 occurrences) and CURR-CODE (first occurrence), the database field PERSONNEL-ID, and the user-defined variable #TOTAL-SALARY are written to the intermediate storage.
- The average of #TOTAL-SALARY is evaluated.

Second Phase:

- The records are sorted.

Third Phase:

- The sorted intermediate storage is read.
- At the at-start-of-data condition, the average of #TOTAL-SALARY is displayed.
- #TOTAL-SALARY is added to #TOTAL-TOTAL and the fields PERSONNEL-ID, SALARY(1), SALARY(2), #AVER-PERCENT and #TOTAL-SALARY are displayed.
- At the end-of-data condition, the variable #TOTAL-TOTAL is written.

 Copyright © Software AG 2002

STACK

```
STACK [TOP] { COMMAND operand1 [operand2 [(parameter)]]... }
            { [DATA] [FORMATTED] {operand2 [(parameter)]}... }
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S	A	G	N	A												yes	no
Operand2	C	S	A	G	N	A	N	P	I	F	B	D	T	L		G		yes	yes

Related Statements: INPUT | RELEASE

Function

The STACK statement is used to place any of the following into the Natural stack:

- the name of a Natural program or Natural system command to be executed;
- data to be used during the execution of an INPUT statement.

For further information on the stack, see the section Further Programming Aspects of the Natural Programming Guide.

TOP

If you specify TOP, the data/program/command will be placed at the top of the Natural stack. Otherwise, they are placed at the bottom of the stack.

Example:

The following statement causes the content of the variable #FIELDA to be placed as data on top of the stack:

```
STACK TOP #FIELDA
```

DATA

DATA (which is also the default) causes data to be placed in the stack which are to be used as input data for an INPUT statement.

Delimiter characters or input assign characters contained within the data values will be processed as delimiters. For details on how data from the stack are processed by an INPUT statement, please refer to the description of the INPUT statement.

Example:

The following statements cause the contents of the variables #FIELD1 and #FIELD2 to be placed in the stack:

```
MOVE 'ABC' TO #FIELD1
MOVE 'XYZ' TO #FIELD2
STACK #FIELD1 #FIELD2
```

These variables will be passed as data to the next INPUT statement in the Natural program, using delimiter mode:

```
INPUT #FIELD1  #FIELD2
```

Note:
If operand2 is a time variable (format T), only the time component of the variable content is placed in the stack, but not the date component.

FORMATTED

FORMATTED causes all data to be passed on a field-by-field basis to the next INPUT statement; no key assignments or delimiter characters will be interpreted.

Examples:

The following statements cause "ABC,DEF" to be placed in #FIELD1 and "XYZ" in #FIELD2:

```
MOVE 'ABC,DEF' TO #FIELD1
MOVE 'XYZ'     TO #FIELD2
STACK TOP DATA FORMATTED #FIELD1 #FIELD2
...
INPUT #FIELD1 #FIELD2
```

Assuming the input delimiter character to be the comma (ID=,), the following statements - without the keyword FORMATTED - cause "ABC" to be placed in #FIELD1 and "DEF" in #FIELD2:

 Copyright © Software AG 2002

```
MOVE 'ABC,DEF' TO #FIELD1
STACK TOP DATA #FIELD1
...
INPUT #FIELD1 #FIELD2
```

COMMAND operand1

To place a command (or program name) in the stack, you specify the keyword COMMAND followed by the command (*operand1*). Natural will execute the command instead of displaying the NEXT prompt and prompting the user for input.

Example:

The following statement causes the command RUN to be placed at the top of the stack. Natural will execute this command at the point where the NEXT prompt would normally be issued.

```
STACK TOP COMMAND 'RUN'
```

COMMAND operand1 operand2...

Together with a command (*operand1*), you may also place data (*operand2*) in the stack. These data will then be processed by the next INPUT statement after the command has been executed.

Data stacked with a command are always stacked unformatted.

Note:
If the data to be stacked include empty alphanumeric fields (i.e., blanks), these blanks will be interpreted as delimiters between values and thus not processed correctly by the corresponding INPUT statement. Therefore, if you wish to stack empty alphanumeric fields as data with a command, you have to use two STACK statements: one "STACK DATA operand2..." to stack the data, and one "STACK COMMAND operand1" to stack the command.

parameter

If *operand2* is a date variable, you can specify the session parameter DF as parameter for this variable. The session parameter DF is described in the Natural Reference documentation.

 Copyright © Software AG 2002

Example

```
/* EXAMPLE 'STKEX1': STACK
/*****************************************
INPUT 'PLEASE SELECT DESIRED FUNCTION:' //
  10X 'LIST FILES  (F)'  /
  10X 'LIST PROGRAMS (P)' /
  10X 'FUNCTION:' #RSP (A1)
/*****************************************
IF NOT (#RSP = 'F' OR = 'P')
 REINPUT 'PLEASE ENTER A CORRECT FUNCTION'
/*****************************************
IF #RSP = 'F'
 DO STACK TOP COMMAND 'LIST FILES *' STOP DOEND
/*****************************************
IF #RSP = 'P'
 DO STACK TOP COMMAND 'LIST PROGRAMS *' STOP DOEND
/*****************************************
END
```

```
PLEASE SELECT DESIRED FUNCTION:

            LIST FILES  (F)
            LIST PROGRAMS (P)
            FUNCTION: p
```

```
16:51:17          ***** NATURAL LIST COMMAND *****                    03-27-87
  USER: TM       LIST                                               LIB: RJNV2RM
  C Name     Pgm. Type   SM S/C Vers Level Userid   Time     Date
  - -------- ----------- -- --- ---- ----- -------- ----- --------
    SISXXX   Program     S  S   2.1  0000  RJ       12:06 87:03:13
    SKPEX1   Program     R  S   2.1  0000  RJ       19:24 87:03:23
    SKYEX1   Program     S  S   2.1  0000  RJ       19:02 87:03:23
    SRTEX1   Program     R  S   2.1  0000  RJ       10:11 87:03:13
    SRTEX1R  Program     R  S   2.1  0000  RJ       09:44 87:03:24
    SRTEX1S  Program     S  S   2.1  0000  RJ       09:48 87:03:24
    SRTEX2   Program     S  S   2.1  0000  RJ       10:35 87:03:13
    STIEX1   Program     S  S   2.1  0000  RJ       19:21 87:03:23
    STKEX1   Program     R  S   2.1  0000  RJ       09:54 87:03:24
    STKEX1R  Program     R  S   2.1  0000  RJ       10:46 87:03:13
    STOEX1   Program     S  S   2.1  0000  RJ       11:28 87:03:13
    STOEX1R  Program     R  S   2.1  0000  RJ       10:45 87:03:24
    STOEX1S  Program     S  S   2.1  0000  RJ       10:14 87:03:24
    STOEX1T  Program     R  S   2.1  0000  RJ       19:35 87:03:24
    STOEX1U  Program     R  S   2.1  0000  RJ       19:42 87:03:24
```

STOP

```
STOP
```

Function

The STOP statement is used to terminate the execution of a program and return to the command input prompt.

One or more STOP statements may be inserted anywhere within a Natural program.

The STOP statement will terminate the execution of the program immediately. Independent of the positioning of a STOP statement in a subroutine, any end-page condition specified in the main program will be invoked for final end-page processing during execution of the STOP statement.

Example

```
/* EXAMPLE 'STPEX1': STOP
/************************************************
INPUT 'PLEASE SELECT DESIRED FUNCTION:' //
  10X 'LIST FILES  (F)'  /
  10X 'LIST PROGRAMS (P)' /
  10X 'FUNCTION:' #RSP (A1)
/************************************************
IF #RSP = ' '
 STOP
/************************************************
IF NOT (#RSP = 'F' OR = 'P')
 REINPUT 'PLEASE ENTER A CORRECT FUNCTION'
/************************************************
IF #RSP = 'F'
 DO
  STACK TOP COMMAND 'LIST FILES *'
  STOP
 DOEND
/************************************************
IF #RSP = 'P'
 DO
  STACK TOP COMMAND 'LIST PROGRAMS *'
  STOP
 DOEND
/************************************************
END
```

 Copyright © Software AG 2002

STORE

Structured Mode Syntax

```
STORE [RECORD] [IN] [FILE] view-name
      [PASSWORD = operand1]
      [CIPHER = operand2]
      [ [USING] NUMBER operand3 ] [(r)]
      [ [GIVING]                ]
```

Reporting Mode Syntax

```
STORE [RECORD] [IN] [FILE] view-name
      [PASSWORD = operand1]
      [CIPHER = operand2]
      [ [USING] NUMBER operand3 ]
      [ [GIVING]               ]
      { [USING] SAME [RECORD] [AS] [STATEMENT [(r)]] }
      { [SET  ] [operand4 = operand5]...             }
      { [WITH ]                                      }
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S				A												yes	no
Operand2	C	S					N											yes	no
Operand3		S					N	P										no	yes
Operand4		S	A			A	N	P	I	F	B	D	T	L				no	no
Operand5	C	S	A			A	N	P	I	F	B	D	T	L				yes	no

Related Statements: UPDATE | DELETE | END TRANSACTION | BACKOUT TRANSACTION

Function

The STORE statement is used to add a record to a database.

Notes for DL/I databases:
This statement may be used to add a segment occurrence.
If the dataset is defined with a primary key, a value for the primary key field must be provided.
In the case of a GSAM database, records must be added at the end of the database (due to GSAM restrictions).

Note for SQL databases:
This statement may be used to add a row to a table. The PASSWORD, CIPHER, and GIVING NUMBER clauses cannot be used. The STORE statement corresponds with the SQL statement INSERT.

Note for VSAM databases:
If the dataset is defined with a primary key, a value for the primary key field must be provided.

view-name

As *view-name*, you specify the name of a view, which must have been defined either in a DEFINE DATA statement or outside the program in a global or local data area.

In reporting mode - if no DEFINE DATA statement is used - *view-name* may also be the name of a DDM.

PASSWORD/CIPHER

These clauses are applicable only for an Adabas or VSAM database.

The PASSWORD clause is used to provide a password when updating data from a file which is password-protected.

The CIPHER clause is used to provide a cipher key when updating data from a file which is enciphered.

See the statements FIND and PASSW for further information.

USING/GIVING NUMBER

This clause can only be used for an Adabas or VSAM database.

This option is used to store a record with a user-supplied Adabas ISN. If a record with the specified ISN already exists, an error message will be returned and the execution of the program will be terminated unless ON ERROR processing was specified.

Note for VSAM databases:
This clause is only valid for VSAM RRDS, in which case a user-supplied RRN (relative record number) corresponds to the ISN as described above.

SET/WITH

SET/WITH can be used in reporting mode to specify the fields for which values are being provided. Any field defined in the file that is not specified in the SET clause will contain a null value in the new record.

 Copyright © Software AG 2002

This clause is not permitted if a DEFINE DATA statement is used, because in that case the STORE statement always refers to the entire view as defined in the DEFINE DATA statement.

DL/I Considerations

Values must be provided for the segment sequence field, and for all sequence fields of the ancestors.

Only I/O (sensitive) fields may be provided.

A segment of variable length is stored with the minimum length necessary to contain all fields as specified with the STORE statement. The segment length will never be less than the minimum size specified in the SEGM macro of the DBD.

If a multiple-value field or a periodic group is defined as variable in length, at the end of the segment only the occurrences as specified in the STORE statement are written to the segment and define the segment length.

USING SAME

USING SAME can be used in reporting mode to indicate that the same field values as read in the statement referenced by the STORE statement (FIND, GET, READ) are to be used to add a new record.

This clause is not permitted if a DEFINE DATA statement is used, because in that case the STORE statement always refers to the entire view as defined in the DEFINE DATA statement.

System Variable *ISN

The Natural system variable *ISN contains the Adabas ISN or VSAM RBA/RRN respectively assigned to the new record as a result of the STORE statement execution. A subsequent reference to *ISN must include the statement number of the related STORE statement.

For VSAM databases, *ISN is available only for ESDS and RRDS files.

*ISN is not available for DL/I and SQL databases.

Example

```
 /* EXAMPLE 'STOEX1': STORE
*********************************************
RESET #BIRTH-D (D)
*
REPEAT
  INPUT 'ENTER A PERSONNEL ID AND NAME (OR ''END'' TO END)' //
        'PERSONNEL-ID : ' #PERSONNEL-ID (A8)   //
        'NAME         : ' #NAME         (A20)  /
        'FIRST-NAME   : ' #FIRST-NAME   (A15)
*********************************************
*  VALIDATE ENTERED DATA
*
  IF #PERSONNEL-ID = 'END' OR #NAME = 'END' THEN STOP
  IF #NAME = ' '  THEN
    REINPUT WITH TEXT 'ENTER A LAST-NAME' MARK 2 AND SOUND ALARM
  IF #FIRST-NAME = ' ' THEN
    REINPUT WITH TEXT 'ENTER A FIRST-NAME' MARK 3 AND SOUND ALARM
*********************************************
*  ENSURE PERSON IS NOT ALREADY ON FILE
*
  FIND NUMBER EMPLOYEES WITH PERSONNEL-ID =  #PERSONNEL-ID
  IF *NUMBER > 0 THEN
    REINPUT 'PERSON WITH SAME PERSONNEL-ID ALREADY EXISTS'
            MARK 1 AND SOUND ALARM
*********************************************
*  GET FURTHER INFORMATION
*
  INPUT
    'ADDITIONAL PERSONNEL DATA'                      ////
    'PERSONNEL-ID             :' #PERSONNEL-ID (AD=IO) /
    'NAME                     :' #NAME         (AD=IO) /
    'FIRST-NAME               :' #FIRST-NAME   (AD=IO) ///
    'MARITAL STATUS           :' #MAR-STAT (A1)        /
    'DATE OF BIRTH (YYYYMMDD) :' #BIRTH    (A8)        /
    'CITY                     :' #CITY     (A20)       /
    'COUNTRY (3 CHARACTERS)   :' #COUNTRY  (A3)        //
    'ADD THIS RECORD (Y/N)    :' #CONF     (A1) (AD=M)
*********************************************
*   ENSURE REQUIRED FIELDS CONTAIN VALID DATA
*
  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 1
  IF NOT (#BIRTH = MASK(YYYYMMDD) AND #BIRTH = MASK(1582-2699))
    REINPUT TEXT 'ENTER CORRECT DATE' MARK 2
  IF #CITY  = ' '
    REINPUT TEXT 'ENTER A CITY NAME' MARK 3
  IF #COUNTRY = ' '
    REINPUT TEXT 'ENTER A COUNTRY CODE' MARK 4
  IF NOT (#CONF = 'N' OR= 'Y')
    REINPUT TEXT 'ENTER Y (YES) OR N (NO)' MARK 5
  IF #CONF = 'N'
    ESCAPE TOP
*********************************************
* continued on next page
```

 Copyright © Software AG 2002

```
* continued from previous page
*********************************************
*  ADD THE RECORD
*
  MOVE EDITED #BIRTH TO #BIRTH-D (EM=YYYYMMDD)
*
  STORE RECORD IN EMPLOYEES
  WITH  PERSONNEL-ID = #PERSONNEL-ID
        NAME         = #NAME
        FIRST-NAME   = #FIRST-NAME
        MAR-STAT     = #MAR-STAT
        BIRTH        = #BIRTH-D
        CITY         = #CITY
        COUNTRY      = #COUNTRY
  END OF TRANSACTION
*
  WRITE NOTITLE 'RECORD HAS BEEN ADDED'
  RESET INITIAL #CONF
*
LOOP
END
```

```
ENTER A PERSONNEL ID AND NAME (OR 'END' TO END)

PERSONNEL-ID : 90001100

NAME         : JONES
FIRST-NAME   : EDWARD
```

```
ADDITIONAL PERSONNEL DATA



PERSONNEL-ID             : 90001100
NAME                     : JONES
FIRST-NAME               : EDWARD


MARITAL STATUS           : m
DATE OF BIRTH (YYYYMMDD) : 690511
CITY                     : wan chai
COUNTRY (3 CHARACTERS)   : hkg

ADD THIS RECORD (Y/N)    : y
```

SUBTRACT

Syntax 1

```
SUBTRACT [ROUNDED] operand1 ...FROM operand2
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S	A		N		N	P	I	F		D	T					yes	no
Operand2		S	A		M		N	P	I	F		D	T					yes	no

Syntax 2

```
SUBTRACT [ROUNDED] operand1 ...FROM operand2 GIVING operand3
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S	A		N		N	P	I	F		D	T					yes	no
Operand2	C	S	A		N		N	P	I	F		D	T					yes	no
Operand3		S	A		M	A	N	P	I	F	B	D	T					yes	yes

Related Statement: COMPUTE

Function

The SUBTRACT statement is used to subtract the values of two or more operands.

Operands

As for the formats of the operands, see also the section Performance Considerations for Mixed Formats in the Natural Reference documentation.

If a database field is used as the result field, the SUBTRACT operation only results in an update to the internal value that is used within the program. The value for the field in the database remains unchanged.

 Copyright © Software AG 2002

Result Field

If the GIVING clause is used, *operand2* will not be modified and the result will be stored in *operand3*. If the GIVING clause is not used, the result will be stored in *operand2*.

ROUNDED

If you specify the keyword ROUNDED, the result will be rounded. See the section Rules for Arithmetic Assignment in the Natural Reference documentation for information on rounding.

Example

```
/* EXAMPLE 'SUBEX1': SUBTRACT
/*********************************************************************
DEFINE DATA LOCAL
  1 #A (P2) INIT <50>
  1 #B (P2)
  1 #C (P1.1) INIT <2.4>
END-DEFINE
/*********************************************************************
SUBTRACT 6 FROM #A
WRITE NOTITLE 'SUBTRACT 6 FROM #A              '  10X '=' #A
/*********************************************************************
SUBTRACT 6 FROM 11 GIVING #A
WRITE         'SUBTRACT 6 FROM 11 GIVING #A    '  10X '=' #A
/*********************************************************************
SUBTRACT 3 4 FROM #A GIVING #B
WRITE         'SUBTRACT 3 4 FROM #A GIVING #B  '  10X '=' #A '=' #B
/*********************************************************************
SUBTRACT -3 -4 FROM #A GIVING #B
WRITE         'SUBTRACT -3 -4 FROM #A GIVING #B'  10X '=' #A '=' #B
/*********************************************************************
SUBTRACT ROUNDED 2.06 FROM #C
WRITE         'SUBTRACT ROUNDED 2.06 FROM #C   '  10X '=' #C
/*********************************************************************
END
```

```
SUBTRACT 6 FROM #A                        #A:  44
SUBTRACT 6 FROM 11 GIVING #A              #A:   5
SUBTRACT 3 4 FROM #A GIVING #B            #A:   5 #B:  -2
SUBTRACT -3 -4 FROM #A GIVING #B          #A:   5 #B:  12
SUBTRACT ROUNDED 2.06 FROM #C             #C:  0.3
```

SUSPEND IDENTICAL SUPPRESS

```
SUSPEND IDENTICAL [SUPPRESS] [(rep)]
```

Related Statements: DISPLAY | WRITE

Function

The SUSPEND IDENTICAL SUPPRESS statement is used to suspend the parameter IS=ON (which suppresses the output of identical field values) for the processing of one record. See also the session parameter IS in the Natural Reference documentation.

Report Specification - rep

The notation (*rep*) may be used to specify the identification of the report for which the SUSPEND IDENTICAL SUPPRESS statement is applicable. A value in the range 0 - 31 or a logical name which has been assigned using the DEFINE PRINTER statement may be specified.

If (*rep*) is not specified, the SUSPEND IDENTICAL SUPPRESS statement will be applicable to the first report (report 0).

Example

Program with SUSPEND IDENTICAL SUPPRESS:

 Copyright © Software AG 2002

```
/* EXAMPLE 'SISEX1': SUSPEND IDENTICAL SUPPRESS
/************************************************************
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 '***NO CAR***' TO MAKE
       END-NOREC
       DISPLAY NOTITLE
               NAME (RD.) (IS=ON) FIRST-NAME (RD.) (IS=ON)
               MAKE (FD.)
      END-FIND
    END-READ
END
```

```
       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              ***NO CAR***
JOUSSELIN            DANIEL               RENAULT
JUBE                 GABRIEL              ***NO CAR***
JUNG                 ERNST                ***NO CAR***
JUNKIN               JEREMY               ***NO CAR***
KAISER               REINER               ***NO CAR***
```

Same as Previous Program, but without SUSPEND IDENTICAL SUPPRESS:

```
/* EXAMPLE 'SISEX2': SUSPEND IDENTICAL SUPPRESS
/* (SIMILAR TO EXAMPLE 'SISEX1' EXCEPT THAT SUSPEND IDENTICAL SUPPRESS
/* STATEMENT IS NOT USED. COMPARE OUTPUT OF EXAMPLES SISEX1 AND SISEX2
/* TO SEE EFFECT OF SUSPEND IDENTICAL).
/**********************************************************************
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 STATEMENT REMOVED *********************
FD.   FIND VEH-VIEW 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
    END-READ
END
```

```
       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***
```

 Copyright © Software AG 2002

TERMINATE

```
TERMINATE [operand1 [operand2]]
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S					N	P										yes	no
Operand2*	C	S	A			A	N	P	I	F	B	D	T	L	C			yes	yes

* Under OpenVMS, UNIX and Windows, operand2 must be in alphanumeric format.

Function

The TERMINATE statement is used to terminate a Natural session.

A TERMINATE statement may be placed anywhere within a Natural program.

When a TERMINATE statement is executed, no end-of-page or end-loop processing will be performed.

operand1

Operand1 may be used to pass a return code to the program receiving control when Natural terminates. For example, a return code setting may be passed to the operating system. This is done differently, depending on the platform under which Natural is running:

UNIX

via "argc,argv";

Windows

via the operating-system function ERRORLEVEL;

Mainframes

via register 15.

The value supplied for operand1 must be in the range 0 - 255.

operand2

Note:
Operand2 is only available on mainframes.

Operand2 may be used to pass additional information to the program which receives control after the termination.

Program Receiving Control after Termination

After the termination of the Natural session, the program whose name is specified with the profile parameter PROGRAM (see your Natural Installation and Operations documentation) will receive control.

Under OpenVMS and UNIX, Natural passes the three following parameters (if specified) to that program: operand1, operand2 and the value of the profile parameter PRGPAR (see the Natural Installation and Operations documentation for OpenVMS and UNIX).

Under OpenVMS, if the PROGRAM parameter is not set, the OpenVMS DCL will receive control after the termination, and operand1 (if specified) will be passed to the DCL.

Under UNIX, if the PROGRAM parameter is not set, the UNIX command shell will receive control after the termination, and operand1 (if specified) will be passed to the command shell.

Example

```
/* EXAMPLE 'TEREX1': TERMINATE
/*****************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 SALARY (1)
1 #PNUM (A8)
1 #PASS (A8)
END-DEFINE
/*****************************************
INPUT 'ENTER PASSWORD:' #PASS
IF #PASS NE 'USERPASS'
 TERMINATE
END-IF
/*****************************************
INPUT 'ENTER PERSONNEL NUMBER:' #PNUM
FIND EMPLOY-VIEW WITH PERSONNEL-ID = #PNUM
 DISPLAY NAME SALARY (1)
END-FIND
/*****************************************
END
```

 Copyright © Software AG 2002

UPDATE

Structured Mode Syntax

```
UPDATE [RECORD] [IN] [STATEMENT] [(r)]
```

Reporting Mode Syntax

```
UPDATE [RECORD] [IN] [STATEMENT] [(r)]
       [ SET  ] { SAME [RECORD]          }
       [ WITH ] {{operand1 = operand2}...}
       [USING ]
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S	A			A	N	P	I	F	B	D	T	L				no	no
Operand2	C	S	A			A	N	P	I	F	B	D	T	L				yes	no

Related Statements: STORE | DELETE | END TRANSACTION | BACKOUT TRANSACTION

Function

The UPDATE statement is used to update one or more fields of a record in a database. The record to be updated must have been previously selected with a FIND, GET or READ statement (or, for Adabas only, with a STORE statement).

Considerations for DL/I Databases

The UPDATE statement can be used to update a segment in a DL/I database. If necessary, the segment length is increased to accommodate all fields specified with the UPDATE statement.

If a multiple-value field or a periodic group is defined as variable in length, only the occurrences as specified in the UPDATE statement are written to the segment.

The DL/I AIX field name cannot be used in an UPDATE statement. AIX fields, however, may be updated by referring to the source field which comprises the AIX field.

Copyright © Software AG 2002

DL/I sequence fields cannot be updated because of DL/I restrictions.

Due to GSAM restrictions, the UPDATE statement cannot be used for GSAM databases.

Considerations for SQL Databases

The UPDATE statement can be used to update a row in a database table. It corresponds with the SQL statement UPDATE WHERE CURRENT OF cursor-name (positioned UPDATE), which means that only the row which was read last can be updated.

On mainframe computers, only columns (fields) that have been modified within the program, as well as columns that `might` have been (but not necessarily actually have been) modified outside the program (for example, as input fields in maps), are updated. On all other platforms, all columns are updated.

With most SQL databases, a row that was read with a FIND SORTED BY or with a READ LOGICAL statement cannot be updated.

Considerations for VSAM Databases

VSAM primary keys cannot be updated because of VSAM restrictions.

The DL/I AIX field name cannot be used in an UPDATE statement. AIX fields, however, may be updated by referring to the source field which comprises the AIX field.

Restrictions

The UPDATE statement must not be entered on the same line as the statement used to select the record to be updated.

The UPDATE statement cannot be applied to Entire System Server views.

Statement Reference - r

The notation "(*r*)" is used to indicate the statement in which the record to be modified was read. *r* may be specified as a source-code line number or as a statement label.

If no reference is specified, the UPDATE statement will reference the innermost active READ or FIND processing loop. If no READ or FIND loop is active, it will reference the last preceding GET (or STORE) statement.

Note:
The UPDATE statement must be placed within the READ or FIND loop it references.

USING SAME

This clause is not permitted if a DEFINE DATA statement is used, because in that case the UPDATE statement always refers to the entire view as defined in the DEFINE DATA statement.

USING SAME can be used in reporting mode to indicate that the same fields as read in the statement referenced by the UPDATE statement are to be used for the update function. In this case, the most recent value assigned to each database field will be used to update the field. If no new value has been assigned, the old value will be used.

 Copyright © Software AG 2002

If the field to be updated is an array range of a multiple-value field or periodic group and you use a variable index for this array range, the latest range will be updated. This means that if the index variable is modified after the record has been read and before the UPDATE USING SAME (reporting mode) or UPDATE (structured mode) statement respectively is executed, the range updated will not be the same as the range read.

SET/WITH operand1 = operand2

This clause can be used in reporting mode to specify the fields to be updated and the values to be used.

This clause is not permitted if a DEFINE DATA statement is used, because in that case the UPDATE statement always refers to the entire view as defined in the DEFINE DATA statement.

Note for DL/I databases:
If the SET/WITH clause is used, only I/O (sensitive) fields can be provided. A segment sequence field cannot be updated (DELETE and STORE must be used instead).

Hold Status

The use of the UPDATE statement causes each record read for processing in the corresponding FIND or READ statement to be placed in hold status.

Record hold logic is explained in the Natural Programming Guide.

Example

```
/* EXAMPLE 'UPDEX1S': UPDATE (STRUCTURED MODE)
/*********************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
1 #NAME (A20)
END-DEFINE
/*********************************************************
INPUT 'ENTER A NAME:' #NAME (AD=M)
IF #NAME = ' '
 STOP
END-IF
/*********************************************************
FIND EMPLOY-VIEW WITH NAME = #NAME
 IF NO RECORDS FOUND
  REINPUT WITH 'NO RECORDS FOUND'  MARK 1
 END-NOREC
 INPUT 'NAME:' NAME (AD=O) /
       'FIRST NAME:' FIRST-NAME (AD=M) /
       'CITY:' CITY (AD=M)
  UPDATE
  END TRANSACTION
 END-FIND
/*********************************************************
END
```

```
ENTER A NAME: BROWN
```

```
NAME: BROWN
FIRST NAME: KENNETH
CITY: DERBY
```

Equivalent reporting-mode example: See the program UPDEX1R in the library SYSEXRM.

 Copyright © Software AG 2002

UPLOAD

This statement is only available with Natural Connection. It is described in the Natural Connection documentation.

WRITE

Syntax 1 - Dynamic Formatting

```
WRITE [(rep)] [NOTITLE] [NOHDR]

      [(statement-parameters)]

      { [ nX                  ]                                          }
      { [ nT                  ]  { 'text' [(attributes)]                 }
      { [ x/y                 ]  { 'c'(n) [(attributes)]                 }
      { [ T* field-name       ]  { ['='] operand1 [(parameters)]         }
      { [ P* field-name       ]  }                                       }
      { [ /                   ]...                                       }...
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S	A	G	N	A	N	P	I	F	B	D	T	L		G	O	yes	no

Related Statements: DISPLAY | WRITE TITLE | WRITE TRAILER | INPUT

Function

The WRITE statement is used to produce output in free format.

The WRITE statement differs from the DISPLAY statement in the following respects:

- Line overflow is supported. If the line width is exceeded for a line, the next field (or text) is written on the next line. Fields or text elements are not split between lines.
- No default column headers are created. The length of the data determines the number of positions printed for each field.
- A range of values/occurrences for an array is output horizontally rather than vertically.

 Copyright © Software AG 2002

Report Specification - rep

The notation (*rep*) is used to specify the number of the report if multiple reports are to be produced by the program. A value in the range 0 - 31 or a logical name which has been assigned using the DEFINE PRINTER statement may be specified. If (*rep*) is not specified, the statement will apply to the first report (report 0).

NOTITLE

Natural generates a single title line for each page resulting from a WRITE statement. This title contains the page number, the time of day, and the date. Time of day is set at the beginning of program execution.

This title line may be overridden by using a WRITE TITLE statement, or it may be suppressed by specifying the NOTITLE clause in the WRITE statement.

Examples:

```
Default title will be produced:
WRITE NAME
```

```
User title will be produced:
WRITE NAME
WRITE TITLE 'USER TITLE'
```

```
No title will be produced:
WRITE NOTITLE NAME
```

If the NOTITLE option is used, it applies to all DISPLAY, PRINT and WRITE statements within the same object which write data to the same report.

Page overflow is checked **before** execution of a WRITE statement. No new page with title or trailer information is generated **during** the execution of a WRITE statement.

NOHDR

The WRITE statement itself does not produce any column headers. However, if you use the WRITE statement in conjunction with a DISPLAY statement, you can use the NOHDR option of the WRITE statement to suppress the column headers generated by the DISPLAY statement: the NOHDR option only takes effect if the WRITE statement is executed **after** a DISPLAY statement, the output spans more than one page, and the execution of the WRITE statement causes a new page to be output. Without the NOHDR option, the column headers of the DISPLAY statement would be output on this new page; with NOHDR they will not.

statement-parameters

One or more parameters, enclosed within parentheses, may be specified immediately after the WRITE statement.

Each parameter specified in this manner will override any previous parameter specified in a GLOBALS command, SET GLOBALS or FORMAT statement.

If more than one parameter is specified, one or more blanks must be present between each entry. An entry may not be split between two statement lines.

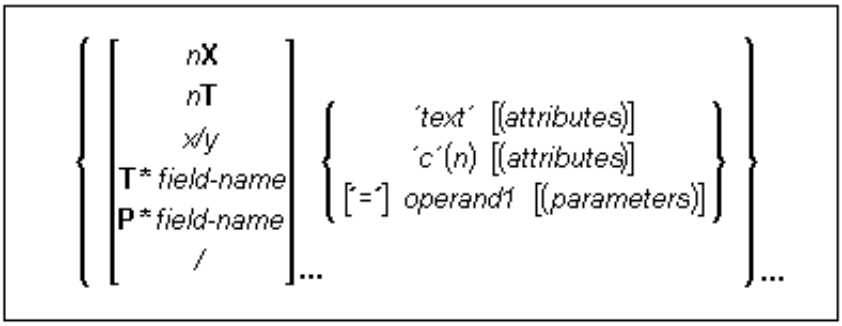
The parameter settings appiled here will only be regarded for variable fields, but they have no effect on text-constants. If you would like to set field attributes for a text-constant, they have to be set explicitly for this element.

Example:

```
DEFINE DATA LOCAL
1 VARI (A4)     INIT <'1234'>                  /*     Output
END-DEFINE                                     /*    Produced
*                                              /*    ---------
WRITE           'Text'           VARI          /*    Text 1234
WRITE (PM=I)    'Text'           VARI          /*    Text 4321
WRITE           'Text' (PM=I)    VARI (PM=I)   /*    txeT 4321
WRITE           'Text' (PM=I)    VARI          /*    txeT 1234
END
```

For details on the individual parameters, see the section Session Parameters in the Natural Reference documentation.

Output Format



Field Positioning Notations

*n*X

Note: (for Mainframes Only)
This notation inserts n spaces between columns. n must not be "0".

Example: WRITE NAME 5X SALARY

*n*T

The *n*T notation causes positioning (tabulation) to print position "*n*". Backward positioning is not permitted.

Example: WRITE 25T NAME 50T SALARY

 Copyright © Software AG 2002

(causes NAME to print beginning in position 25 and SALARY to print beginning in position 50).

x/y

Causes the next element to be placed *x* lines below the output of the last statement, beginning in column *y*. *y* must not be "0". Backward positioning in the same line is not permitted.

T**field-name*

The notation T* is used to position to a specific print position of a field used in a previous DISPLAY statement. Backward positioning is not permitted.

P**field-name*

The notation P* is used to position to a specific print position *and line* of a field used in a previous DISPLAY statement. It is most often used in conjunction with vertical printing mode. Backward positioning is not permitted.

Equal Sign '='

When placed before a field, '=' results in the display of the field heading (as defined in the DEFINE DATA statement or in the DDM) followed by the field contents.

Slash '/'

When placed between fields or text elements, "/" causes positioning to the beginning of the next print line.

Example: WRITE NAME / SALARY

Multiple "/" notations may be used to cause multiple line advances.

Text/Attribute Assignment

'text'

text is displayed.

Example: WRITE 'EMPLOYEE' NAME 'MARITAL/STATUS' MAR-STAT

'c'(n)

Identical to *'text'* except that the specified character *c* is displayed *n* times.

Example: WRITE '*' (5) '=' NAME

attributes

Indicates the display and color attributes to be used for text/field display. The following *attributes* can be used:

```
[B]  [BL]
[C]  [GR]
[D]  [NE]
[I]  [PI]
[N]  [RE]
[U]  [TU]
[V]  [YE]
 1    2
```

1. Display attributes (see the session parameter AD in the Natural Reference documentation).
2. Color attributes (see the session parameter CD in the Natural Reference documentation).

WRITE 'TEXT' (BGR)
WRITE 'TEXT' (B)
WRITE 'TEXT' (BBLC)

operand1

The field to be written.

Note for DLII databases:
The DL/I AIX fields can be displayed only if a PCB is used with the AIX specified in the parameter PROCSEQ. If not, an error message is returned by Natural at runtime.

parameters

One or more parameters, enclosed within parentheses, may be specified immediately after *operand1*. Each parameter specified in this manner will override any previous parameter specified in a GLOBALS command, SET GLOBALS or FORMAT statement. If more than one parameter is specified, one or more blanks must be present between each entry. An entry may not be split between two statement lines.

Syntax 2 - Using Predefined Map

```
WRITE [(rep)] [NOTITLE] [NOHDR] [USING] {FORM} operand1 [operand2 ...]
                                        {MAP}
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S				A												no	no
Operand2		S	A	G	N	A	N	P	I	F	B	D	T	L				yes	no

 Copyright © Software AG 2002

FORM/MAP

This option may be used to indicate that a form/map layout previously defined using the Natural map editor is to be used.

A map layout used in a WRITE statement does not automatically create a new page each time the map is output.

The LS parameter setting must be 1 byte greater than the LS setting defined in the map.

operand1

The name of the form/map to be used.

operand2

The field to be written.

If *operand1* is a constant and *operand2* is omitted, the fields are taken from the map source at compilation time.

NOTITLE/NOHDR

NOTITLE and NOHDR are described under Syntax 1 of the WRITE statement.

Example 1

```
/* EXAMPLE 'WRTEX1': WRITE (USING '=', 'TEXT', '/')
/*****************************************************
LIMIT 1
READ  EMPLOYEES BY NAME
/*****************************************************
WRITE NOTITLE '=' NAME '=' FIRST-NAME '=' MIDDLE-I //
              'L O C A T I O N' /
              'CITY:' CITY    /
              'COUNTRY:' COUNTRY //
/*****************************************************
END
```

```
NAME: ABELLAN             FIRST-NAME: KEPA                MIDDLE-I:

  L O C A T I O N
  CITY: MADRID
  COUNTRY: E
```

Example 2

```
/* EXAMPLE 'WRTEX2:' WRITE (USING NX, NT NOTATION)
/**************************************************
LIMIT 4
READ EMPLOYEES BY NAME
 WRITE NOTITLE 5X NAME  50T JOB-TITLE
/**************************************************
END
```

```
     ABELLAN                                    MAQUINISTA
     ACHIESON                                   DATA BASE ADMINISTRATOR
     ADAM                                       CHEF DE SERVICE
     ADKINSON                                   SALES PERSON
```

 Copyright © Software AG 2002

Example 3

```
/* EXAMPLE 'WRTEX3': WRITE (USING T* NOTATION)
/*******************************************************
LIMIT 5
READ EMPLOYEES BY CITY STARTING FROM 'ALBU'
 DISPLAY NOTITLE CITY NAME SALARY (1)
 AT BREAK CITY
/*******************************************************
  WRITE / 'CITY AVERAGE:' T*SALARY (1) AVER(SALARY(1)) //
/*******************************************************
END
```

```
      CITY                 NAME           ANNUAL
                                            SALARY
-------------------- -------------------- ----------

ALBUQUERQUE          HAMMOND                   22000
ALBUQUERQUE          ROLLING                   34000
ALBUQUERQUE          FREEMAN                   34000
ALBUQUERQUE          LINCOLN                   41000

CITY AVERAGE:                                  32750


ALFRETON             GOLDBERG                   4700

CITY AVERAGE:                                   4700
```

Example 4

```
* EXAMPLE 'WRTEX4': WRITE (USING P* NOTATION)
*********************************************
LIMIT 3
READ EMPLOYEES BY CITY FROM 'N'
 DISPLAY NOTITLE NAME CITY
         VERT AS 'BIRTH/SALARY' BIRTH (EM=YYYY-MM-DD) SALARY (1)
 SKIP 1
 AT BREAK CITY
*
  WRITE / 'CITY AVERAGE:' P*SALARY (1) AVER(SALARY(1)) //
*
LOOP
END
```

```
         NAME                 CITY           BIRTH
                                             SALARY
-------------------- -------------------- ----------

WILCOX               NASHVILLE            1970-01-01
                                               38000

MORRISON             NASHVILLE            1949-07-10
                                               36000


CITY AVERAGE                                   37000


BOYER                NEMOURS              1955-11-23
                                              195900


CITY AVERAGE                                  195900
```

Example 5

```
/* EXAMPLE 'WRTEX5': WRITE (USING '=', STATEMENT/ELEMENT PARAMETERS)
/*********************************************************************
LIMIT 2
READ EMPLOYEES BY NAME
  WRITE NOTITLE (AL=16 NL=8)
       '=' PERSONNEL-ID '=' NAME '=' PHONE (AL=10 EM=XXX-XXXXXXX)
/*********************************************************************
END
```

```
PERSONNEL ID: 60008339         NAME: ABELLAN          TELEPHONE: 435-672
PERSONNEL ID: 30000231         NAME: ACHIESON         TELEPHONE: 523-341
```

 Copyright © Software AG 2002

WRITE TITLE

```
WRITE [(rep)] TITLE [LEFT [JUSTIFIED]] [UNDERLINED]
      [(statement-parameters)]
      { [ nX ]  { 'text' [(attributes)]                 }  }
      { [ nT ]  { 'c'(n) [(attributes)]                 }  }
      { [x/y ]...{ [' = '] operand1 [(parameters)]      }  }...
      [SKIP operand2 [LINES]]
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S	A	G	N	A	N	P	I	F	B	D	T	L		G	O	yes	no
Operand2	C	S					N	P	I		B							yes	no

Related Statements: WRITE | DISPLAY | WRITE TRAILER

Function

The WRITE TITLE statement is used to override the default page title with a page title of your own. It is executed whenever a new page is initiated.

This statement is non-procedural (that is, its execution depends on an event, not on where in a program it is located).

Note:
If a report is produced by statements in different objects, the WRITE TITLE statement is only executed if it is contained in the same object as the statement that causes a new page to be initiated.

Restrictions

WRITE TITLE may be specified only once per report.

WRITE TITLE cannot be specified within a special condition statement block.

WRITE TITLE cannot be specified within a subroutine.

Report Specification - rep

If multiple reports are to be produced, the notation (*rep*) may be used to specify the identification of the report for which the WRITE TITLE statement is applicable. A value in the range 0 - 31 or a logical name which has been assigned using the DEFINE PRINTER statement may be specified.

If (*rep*) is not specified, the WRITE TITLE statement applies to the first report (report 0).

Justification and Underlining

By default, page titles are centered and not underlined. LEFT JUSTIFIED and/or UNDERLINED may be specified to override these defaults. If UNDERLINED is specified, the underlining character (system default or specified with the UC parameter in a FORMAT statement) is printed underneath the title and runs the width of the line size (LS parameter).

Natural first applies all spacing or tab specifications and creates the line before centering the whole line. For example, a notation of "10T" as the first element would cause the centered header to be positioned five positions to the right.

statement-parameters

One or more parameters, enclosed within parentheses, may be specified immediately after WRITE TITLE. Each parameter specified in this manner will override any previous parameter specified in a GLOBALS command, SET GLOBALS or FORMAT statement.

If more than one parameter is specified, one or more blanks must be present between each entry. An entry may not be split between two statement lines.

For a description of each parameter, see the section Session Parameters in the Natural Reference documentation.

operand1

Operand1 represents the field(s) to be displayed within the title. The format notations and spacing elements to be used are identical to those used with the WRITE statement (see the WRITE statement for more information).

SKIP - operand2

SKIP may be used to cause lines to be skipped immediately after the title line. The number of lines to be skipped may be specified as a numeric constant or as the content of a numeric variable.

Note:
SKIP after WRITE TITLE is always interpreted as the SKIP clause of the WRITE TITLE statement, and not as an independent statement. If you wish an independent SKIP statement after a WRITE TITLE statement, use a semicolon (;) to separate the two statements from one another.

 Copyright © Software AG 2002

Example

```
/* EXAMPLE 'WTIEX1': WRITE TITLE
/************************************************
FORMAT LS=70
/************************************************
WRITE TITLE LEFT JUSTIFIED UNDERLINED
      *TIME 3X 'PEOPLE LIVING IN NEW YORK CITY'
           11X 'PAGE:' *PAGE-NUMBER
SKIP1
/************************************************
FIND EMPLOYEES WITH CITY = 'NEW YORK'
 DISPLAY NAME FIRST-NAME 3X JOB-TITLE
/************************************************
END
```

```
15:57:19.1   PEOPLE LIVING IN NEW YORK CITY          PAGE:      1
  ---------------------------------------------------------------------

          NAME              FIRST-NAME                 CURRENT
                                                       POSITION
  -------------------- --------------------   -------------------------

  RUBIN                SYLVIA                 SECRETARY
  WALLACE              MARY                   ANALYST
```

WRITE TRAILER

```
WRITE [(rep)] TRAILER [LEFT [JUSTIFIED]] [UNDERLINED]
      [(statement-parameters)]
      { [ nX ]  { 'text' [(attributes)]          }  }
      { [ nT ]  { 'c'(n) [(attributes)]          }  }
      { [x/y] ...{ [ = ] operand1 [(parameters)] }  }...
      [SKIP operand2 [LINES]]
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1		S	A	G	N	A	N	P	I	F	B	D	T	L		G	O	yes	no
Operand2	C	S					N	P	I		B							yes	no

Related Statements: WRITE | DISPLAY | WRITE TITLE

Function

The WRITE TRAILER statement is used to output text or the contents of variables at the bottom of a page.

This statement is non-procedural (that is, its execution depends on an event, not on where in a program it is located).

Restrictions

WRITE TRAILER may be specified only once per report.

WRITE TRAILER cannot be specified within a special condition statement block.

WRITE TRAILER cannot be specified within a subroutine.

Processing

This statement is executed when an end-of-page or end-of-data condition is detected, or when a SKIP or NEWPAGE statement causes a page advance. It is not executed as a result of an EJECT statement.

The end-of-page condition is checked only after the processing of an entire DISPLAY/WRITE statement. If a DISPLAY/WRITE statement produces multiple lines of output, overflow of the physical page may occur before the end-of-page condition is reached.

 Copyright © Software AG 2002

Note:
If a report is produced by statements in different objects, the WRITE TRAILER statement is only executed if it is contained in the same object as the statement that causes the end-of-page condition.

Logical Page Size

The logical page size (specified with the session parameter PS) should be less than the physical page size to ensure that the trailer information appears at the bottom of the same page.

Report Specification - rep

The notation (*rep*) may be used to specify the identification of the report for which the WRITE TRAILER statement is applicable. A value in the range 0 - 31 or a logical name which has been assigned using the DEFINE PRINTER statement may be specified.

If (*rep*) is not specified, the WRITE TRAILER statement applies to the first report (report 0).

Justification and Underlining

By default, the trailer lines are centered and not underlined. LEFT JUSTIFIED and UNDERLINED may be specified to override these defaults. If UNDERLINED is specified, the underlining character (either default or specified with the session parameter UC) is printed underneath the trailer and runs the width of the line size (session parameter LS).

Natural first applies all spacing or tab specifications and creates the line before centering the whole line. For example, a notation of "10T" as the first element would cause the centered header to be positioned five positions to the right.

statement-parameters

One or more parameters, enclosed within parentheses, may be specified immediately after WRITE TRAILER. Each parameter specified in this manner will override any previous parameter specified in a GLOBALS command, SET GLOBALS or FORMAT statement.

If more than one parameter is specified, one or more blanks must be present between each entry. An entry may not be split between two statement lines.

For a description of each parameter, see the section Session Parameters of the Natural Reference documentation.

operand1

Operand1 represents the field/fields to be output as trailer information.

Format notations and spacing elements are identical to those used with the WRITE statement (see the WRITE statement for more information).

SKIP - operand2

SKIP may be used to cause lines to be skipped immediately after the trailer line. The number of lines to be skipped (*operand2)* may be specified as a numeric constant or as the content of a numeric variable.

Note:
SKIP after WRITE TRAILER is always interpreted as the SKIP clause of the WRITE TRAILER statement, and not as an independent statement. If you wish an independent SKIP statement after a WRITE TRAILER statement, use a semicolon (;) to separate the two statements from one another.

Example

```
/* EXAMPLE 'WTLEX1': WRITE TRAILER
/*******************************************
FORMAT PS=15
WRITE TITLE LEFT JUSTIFIED UNDERLINED
      *TIME 3X 'PEOPLE LIVING IN BARCELONA'
            14X 'PAGE:' *PAGE-NUMBER
      SKIP 1
/*******************************************
WRITE TRAILER LEFT JUSTIFIED UNDERLINED
    / 'CITY OF BARCELONA REGISTER'
/*******************************************
LIMIT 10
FIND EMPLOYEES WITH CITY = 'BARCELONA'
 DISPLAY NAME FIRST-NAME 3X JOB-TITLE
/*******************************************
END
```

```
15:57:19.1   PEOPLE LIVING IN BARCELONA              PAGE:      1
  ---------------------------------------------------------------------------

          NAME              FIRST-NAME                CURRENT
                                                      POSITION
  -------------------- --------------------   -------------------------

  DEL CASTILLO         ANGEL                  EJECUTIVO DE VENTAS
  GARCIA               M. DE LAS MERCEDES     SECRETARIA
  GARCIA               ENDIKA                 DIRECTOR TECNICO
  MARTIN               ASUNCION               SECRETARIA
  MARTINEZ             TERESA                 SECRETARIA
  YNCLAN               FELIPE                 ADMINISTRADOR
  FERNANDEZ            ELOY                   OFICINISTA
  TORRES               ANTONI                 OBRERA

  CITY OF BARCELONA REGISTER
  ------------------------------------------------------------------------------
```

```
        NAME              FIRST-NAME                CURRENT
                                                    POSITION
  -------------------- --------------------   -------------------------

  RODRIGUEZ            VICTORIA               SECRETARIA
  GARCIA               GERARDO                INGENIERO DE PRODUCCION

  CITY OF BARCELONA REGISTER
  ------------------------------------------------------------------------------
```

 Copyright © Software AG 2002

WRITE WORK FILE

```
WRITE WORK [FILE] work-file-number [VARIABLE] operand1 ...
```

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S	A	G		A	N	P	I	F	B	D	T	L	C	G		yes	no

Related Statements: DEFINE WORK FILE | READ WORK FILE | CLOSE WORK FILE

Function

The WRITE WORK FILE statement is used to write records to a physical sequential work file.

On mainframe computers, this statement can only be used in batch mode, or under Com-plete, CMS, TSO and TIAM.

It is possible to create a work file in one program or processing loop and to read the same file in a subsequent independent processing loop or in a subsequent program using the READ WORK FILE statement.

work-file-number

The work file number (as defined to Natural) to be used.

VARIABLE

Note:
This option is only available on mainframe computers.

It is possible to write records with different fields to the same work file with different WRITE WORK FILE statements. In this case, the VARIABLE entry must be specified in all WRITE WORK FILE statements. The records on the external file will be written in variable format. Natural will write all output files as variable-blocked (unless you specify a record format and block size in the execution JCL).

Fields - operand1

With *operand1* you specify the fields to be written to the work file. These fields may be database fields, user-defined variables, and/or fields read from another work file using the READ WORK FILE statement.

A database array may be referenced with one single range of indices which indicates the occurrences that are to be written to the work file. Groups from database files may be referenced using the group name. All fields belonging to that group will be written to the work file individually.

Variable Index Range

When writing an array to a work file, you can specify a variable index range for the array. For example:

WRITE WORK FILE work-file-number VARIABLE #ARRAY (I:J)

External Representation of Fields

Fields written with a WRITE WORK FILE statement are represented in the external file according to their internal definition. No editing is performed on the field values.

For fields of format A and B, the number of bytes in the external file is the same as the internal length definition as defined in the Natural program. No editing is performed and a decimal point is not represented in the value.

For fields of format N, the number of bytes on the external file is the sum of internal positions before and after the decimal point. The decimal point is not represented on the external file.

For fields of format P, the number of bytes on the external file is the sum of positions before and after the decimal point, plus 1 for the sign, divided by 2, rounded upward to a full byte.

Note:
No format conversion is performed for fields that are written to a work file.

Examples of Field Representation:

Field Definition	Output Record
#FIELD1 (A10)	10 bytes
#FIELD2 (B15)	15 bytes
#FIELD3 (N1.3)	4 bytes
#FIELD4 (N0.7)	7 bytes
#FIELD5 (P1.2)	2 bytes
#FIELD6 (P6.0)	4 bytes

Note:
When the system functions AVER, NAVER, SUM or TOTAL for numeric fields (format N or P) are written to a work file, the internal length of these fields is increased by one digit (for example, SUM of a field of format P3 is increased to P4). This has to be taken into consideration when reading the work file.

Handling of large and dynamic variables

The RECORD option is not allowed if any dynamic variables are used.

The work file types ASCII, ASCII-COMPRESSED, ENTIRECONNECTION, SAG (binary) and TRANSFER cannot handle dynamic variables and will produce an error. Large variables pose no problem except if the maximum field/record length is exceeded (field length 255 for ENTIRECONNECTION and TRANSFER, record length 32767 for the others). The work file type PORTABLE stores the field information within the work file so that dynamic variables are resized during READ if the field size in the record is different from the current size.

 Copyright © Software AG 2002

Example

```
/* EXAMPLE 'WWFEX1': WRITE WORK FILE
/************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
END-DEFINE
/************************************
FIND EMPLOY-VIEW WITH CITY = 'LONDON'
 WRITE WORK FILE 1
       PERSONNEL-ID
       NAME
END-FIND
/************************************
END
```

Old Statements

The following is a list of old Natural statements.

They can be used in conjunction with NATURAL EXPERT and PREDICT CASE.

- DLOGOFF/DLOGON
- SHOW
- IMPORT
- EXPORT
- INVESTIGATE

For more information about these statements, please see the NATURAL EXPERT and PREDICT CASE documentation.

 Copyright © Software AG 2002

SQL Statements Overview

In addition to the Natural Statements, Natural also provides SQL statements for use in Natural programs so that SQL can be used directly.

The following SQL Statements are available:

CALLDBPROC | COMMIT | DELETE | INSERT | PROCESS SQL | READ RESULT SET | ROLLBACK | SELECT | UPDATE

Note:
Concerning the portability of Natural applications, please bear in mind that the Natural SQL statements can only be used for SQL-eligible database systems, whereas Natural DML statements, like FIND or READ, can be used for all database systems supported by Natural.

This section covers the following topics:

- Common Set and Extended Set

Prior to the description of the actual statements, the following items - which apply to several SQL statements - are described:

- Basic Syntactical Items
- Natural View Concept
- Scalar Expressions
- Search Conditions
- Select Expressions

A further possibility of issuing SQL statements, the so-called Flexible SQL is described, which allows you to use arbitrary SQL syntax.

Copyright © Software AG 2002

Common Set and Extended Set

The SQL statements available within the Natural programming language comprise two different syntax sets:

- **Common Set**
  The Common Set basically corresponds to the standard SQL syntax definitions and is provided for each SQL-eligible database system supported by Natural.
- **Extended Set**
  The Extended Set, in addition, provides special enhancements to the Common Set to support specific features of the various supported database systems. The supported part of the Extended Set differs with each of these database systems.

The following sections mainly describe the Natural SQL Common Set. The statement syntax adheres as far as possible to the syntax described in the relevant literature on SQL; please refer to this literature for further details. For details on the Natural SQL Extended Set, see the documentation of the Natural interface specific to the database system you use.

Those syntax parts which are supported by the SQL *Extended Set* only are highlighted with *grey shading* in the syntax diagrams within this section.

 Copyright © Software AG 2002

Basic Syntactical Items

This section describes basic syntactical items, which are not explained any further within the individual statement descriptions. These items are:

- Constants
- Names
- Parameters

Constants

The constants used in the syntactical descriptions of the Natural SQL statements are *constant* and *integer*.

Note:
If the character for decimal point notation (session parameter DC) is set to a comma (,), any specified numeric constant must not be followed directly by a comma, but must be separated from it by a blank character; otherwise an error or wrong results occur.

Wrong Examples: Right Examples:

```
VALUES (1,'A')    leads to a syntax error      VALUES (1 ,'A')
VALUES (1,2,3)    leads to wrong results       VALUES (1 ,2 ,3)
```

constant

The item *constant* always refers to a Natural constant.

integer

The item *integer* always represents an integer constant.

Names

The names used in the syntactical descriptions of the Natural SQL statements are *authorization-identifier*, *ddm-name*, *view-name*, *column-name*, *table-name* and *correlation-name*.

authorization-identifier

The item *authorization-identifier*, which is also called creator name, is used to qualify database tables and views.

ddm-name

The item *ddm-name* always refers to the name of a Natural DDM as created with the Natural utility SYSDDM.

view-name

The item *view-name* always refers to the name of a Natural view as defined in the DEFINE DATA statement.

column-name

The item *column-name* always refers to the name of a physical database column.

table-name

```
[authorization-identifier.] ddm-name
```

The item *table-name* in this section is used to reference both SQL base tables and SQL viewed tables. A Natural DDM must have been created for a table to be used. The name of such a DDM must be the same as the corresponding database table name or view name.

authorization-identifier

There are two ways of specifying the *authorization-identifier* of a database table or view.

One way corresponds to the standard SQL syntax, in which the *authorization-identifier* is separated from the table name by a period. Using this form, the name of the DDM must be the same as the name of the database table without the *authorization-identifier*.

Example:

```
DEFINE DATA LOCAL
01 PERS VIEW OF PERSONNEL
   02 NAME
   02 AGE
END-DEFINE
SELECT *
  INTO VIEW PERS
  FROM SQL.PERSONNEL
...
```

 Copyright © Software AG 2002

Alternatively, you can define the *authorization-identifier* as part of the DDM name. The DDM name then consists of the *authorization-identifier* and the database table name separated by a hyphen (-). The hyphen between the *authorization-identifier* and the table name is converted internally into a period.

Note:
This form of DDM name can also be used with a FIND or READ statement, because it conforms to the DDM naming conventions applicable to these statements.

Example:

```
DEFINE DATA LOCAL
01 PERS VIEW OF SQL-PERSONNEL
   02 NAME
   02 AGE
END-DEFINE
SELECT *
  INTO VIEW PERS
  FROM SQL-PERSONNEL
...
```

If the *authorization-identifier* has been specified neither explicitly nor within the DDM name, it is determined by the SQL database system.

In addition to being used in SELECT statements, *table-names* can also be specified in DELETE, INSERT and UPDATE statements.

Examples:

```
...
DELETE FROM SQL.PERSONNEL
  WHERE AGE IS NULL
 ...

...
INSERT INTO SQL.PERSONNEL (NAME,AGE)
  VALUES ('ADKINSON',35)
 ...

...
UPDATE SQL.PERSONNEL
SET SALARY = SALARY * 1.1
WHERE AGE > 30
...
```

correlation-name

The item *correlation-name* represents an alias name for a table-name. It can be used to qualify column names; it also serves to implicitly qualify fields in a Natural view when used with the INTO clause of the SELECT statement (see also the SELECT statement).

Example:

```
DEFINE DATA LOCAL
01 PERS-NAME    (A20)
01 EMPL-NAME    (A20)
01 AGE          (I2)
END-DEFINE
...
SELECT X.NAME , Y.NAME , X.AGE
  INTO PERS-NAME , EMPL-NAME , AGE
  FROM SQL-PERSONNEL X , SQL-EMPLOYEES Y
  WHERE X.AGE = Y.AGE
END-SELECT
...
```

Although in most cases the use of *correlation-names* is not necessary, they may help to make the statement clearer.

 Copyright © Software AG 2002

Parameters

parameter

[:] *host-variable* [**INDICATOR** [:] *host-variable*] [**LINDICATOR** [:] *host-variable*]

host-variable

A *host-variable* is a Natural program variable which is referenced in an SQL statement. It can be either an individual field or defined as part of a Natural view.

When defined as a receiving field (for example, in the INTO clause), a *host-variable* identifies a variable to which a value is assigned by the database system.

When defined as a sending field (for example, in the WHERE clause), a *host-variable* specifies a value to be passed from the program to the database system.

colon [:]

To comply with SQL standards, a *host-variable* can also be prefixed by a colon (:). When used with flexible SQL, *host-variables* must be qualified by colons.

Example:

```
SELECT NAME INTO :#NAME FROM PERSONNEL
WHERE AGE = :VALUE
```

The colon is always required if the variable name is identical to an SQL reserved word. In a context in which either a *host-variable* or a column can be referenced, the use of a name without a colon is interpreted as a reference to a column.

Copyright © Software AG 2002

Natural Formats and SQL Data Types

The Natural format of a host-variable is converted to an SQL data type according to the following table:

Natural Format/Length	SQL Data Type
A*n*	CHAR (*n*)
B2	SMALLINT
B4	INT
B*n*; *n* not equal 2 or 4	CHAR (*n*)
F4	REAL
F8	DOUBLE PRECISION
I2	SMALLINT
I4	INT
N*nn.m*	NUMERIC (*nn+m,m*)
P*nn.m*	NUMERIC (*nn+m,m*)
G*n*; for view fields only	GRAPHIC (*n*)

Natural does not check whether the converted SQL data type is compatible to the database column. Except for fields of format N, no data conversion is done.

In addition, the following extensions to standard Natural formats are available with Natural SQL:

- A one-dimensional array of format A can be used to support alphanumeric columns longer than 253 bytes. This array must be defined beginning with index 1 and can only be referenced by using an asterisk (*) as the index. The corresponding SQL data type is CHAR (*n*), where *n* is the total number of bytes in the array.
- A special host variable indicated by the keyword LINDICATOR can be used to support variable-length columns. The corresponding SQL data type is VARCHAR (*n*); see also the LINDICATOR Clause.
- The Natural formats date (D) and time (T) can be used with ENTIRE ACCESS and will be converted into the corresponding database-dependent formats (see the ENTIRE ACCESS documentation for details)

A sending field specified as one-dimensional array without a LINDICATOR field is converted into the SQL data type VARCHAR. The length is the total number of bytes in the array, not taking into account trailing blanks.

 Copyright © Software AG 2002

INDICATOR Clause

The INDICATOR clause is an optional feature to distinguish between a "null" value (that is, no value at all) and the actual values "0" or "blank".

When specified with a receiving *host-variable* (target field), the INDICATOR *host-variable* (null indicator field) serves to find out whether a column to be retrieved is "null".

Example:

```
DEFINE DATA LOCAL
1 NAME      (A20)
1 NAMEIND   (I2)
END-DEFINE
SELECT *
  INTO NAME INDICATOR NAMEIND
  ...
```

In this example, NAME represents the receiving *host-variable* and NAMEIND the null indicator field.

If a null indicator field has been specified and the column to be retrieved is null, the value of the null indicator field is negative and the target field is set to "0" or "blank" depending on its data type. Otherwise, the value of the null indicator field is greater than or equal to "0".

When specified with a sending *host-variable* (source field), the null indicator field is used to designate a null value for this field.

Example:

```
DEFINE DATA LOCAL
1 NAME      (A20)
1 NAMEIND   (I2)
UPDATE ...
SET NAME = :NAME INDICATOR :NAMEIND
WHERE ...
```

In this example, :NAME represents the sending *host-variable* and :NAMEIND the null indicator field. By entering a negative value as input for the null indicator field, a null value is assigned to a database column.

An INDICATOR *host-variable* is of format/length I2.

Copyright © Software AG 2002

LINDICATOR Clause

The LINDICATOR clause is an optional feature which is used to support columns of varying lengths, for example, VARCHAR or LONG VARCHAR type.

When specified with a receiving *host-variable* (target field), the LINDICATOR *host-variable* (length indicator field) contains the number of characters actually returned by the database into the target field. The target field is always padded with blanks.

If the VARCHAR or LONG VARCHAR column contains more characters than fit in the target field, the length indicator field is set to the length actually returned (that is, the length of the target field) and the null indicator field (if specified) is set to the total length of this column.

Example:

```
DEFINE DATA LOCAL
1 ADDRESSLIND (I2)
1 ADDRESS     (A50/1:6)
END-DEFINE
SELECT *
  INTO :ADDRESS(*) LINDICATOR :ADDRESSLIND
  ...
```

In this example, :ADDRESS(*) represents the target field which receives the first 300 bytes (if available) of the addressed VARCHAR or LONG VARCHAR column, and :ADDRESSLIND represents the length indicator field which contains the number of characters actually returned.

When specified with a sending *host-variable* (source field), the length indicator field specifies the number of characters of the source field which are to be passed to the database.

Example:

```
DEFINE DATA LOCAL
1 NAMELIND  (I2)
1 NAME      (A20)
1 AGE       (I2)
END-DEFINE
MOVE  4     TO NAMELIND
MOVE 'ABC%' TO NAME
SELECT AGE
  INTO :AGE
WHERE NAME LIKE :NAME LINDICATOR :NAMELIND
  ...
```

A LINDICATOR *host-variable* is of format/length I2 or I4. For performance reasons, it should be specified immediately before the corresponding target or source field; otherwise, the field is copied to the temporary storage at runtime.

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 *LENGTH are set to this length. In 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.

 Copyright © Software AG 2002

Let a fixed length field be defined with a LINDICATOR field specified as I2. If the VARCHAR column contains more characters than fit into this fixed length field, the length indicator field is set to the length actually returned and the null indicator field (if specified) is set to the total length of this column (retrieval). This is not possible for fixed length fields >= 32KB (length does not fit into null indicator field).

Natural View Concept

Some Natural SQL statements also support the use of Natural views.

A Natural view can be specified instead of a parameter list, where each field of the view - except group fields, redefining fields and fields prefixed with "L@" or "N@"- corresponds to one parameter (host variable).

Fields with names prefixed with "L@" or "N@" can only exist with corresponding master fields; that is, fields of the same name, where:

- L@ fields are converted into LINDICATOR fields,
- N@ fields are converted into INDICATOR fields.

L@ fields should have been specified at view definition, immediately before the master fields to which they apply.

```
DEFINE DATA LOCAL
01 PERS VIEW OF SQL-PERSONNEL
  02 PERSID     (I4)
  02 NAME       (A20)
  02 N@NAME    (I2)                    /* null indicator of NAME
  02 L@ADDRESS (I2)                    /* length indicator of ADDRESS
  02 ADDRESS    (A50/1:6)
  02 N@ADDRESS (I2)                    /* null indicator of ADDRESS
01 #PERSID      (I4)
END-DEFINE
   ...
SELECT *
  INTO VIEW PERS
  FROM SQL-PERSONNEL
  WHERE PERSID = #PERSID
   ...
END-SELECT
```

The above example is equivalent to the following one:

```
   ...
SELECT *
  INTO PERSID,
       NAME INDICATOR N@NAME,
       ADDRESS(*)INDICATOR N@ADDRESS LINDICATOR L@ADDRESS
  FROM SQL-PERSONNEL
  WHERE PERSID = #PERSID
   ...
END-SELECT
```

 Copyright © Software AG 2002

Scalar Expressions

A *scalar-expression* consists of a factor or other scalar expressions including scalar operators.

```
{                    [+]  { factor          }                  }
{                    [-]  {(scalar-expression)}                 }
{scalar-expression   scalar-operator   scalar-expression        }
```

Concerning reference priority, scalar expressions behave as follows: When a non-qualified variable name is specified in a scalar expression, the first approach is to resolve the variable name as column name of the referenced table. If no column with the specified name is available in the referenced table, Natural tries to resolve this variable as a Natural user-defined variable (host variable).

---

scalar-operator

```
{ + }
{ - }
{ * }
{ / }
{ || }
{CONCAT}
```

A *scalar-operator* can be any of the operators listed above; the operators "-" and "/" must be separated by at least one blank from preceding operators.

factor

```
{                                   }
{          atom                     }
{     column-reference              }
{     aggregate-function            }
{     special-register              }
{scalar-function (scalar-expression,...)}
{     scalar-expression  unit       }
{     case-expression               }
```

A *factor* can consist of one of the items listed in the above diagram.

atom

```
{ parameter }
{ constant  }
```

An *atom* can be either a *parameter* or a *constant*; see also the section Basic Syntactical Items.

column-reference

```
[ table-name.        ] column-name
[ correlation-name.  ]
```

A column-reference is a column name optionally qualified by either a table-name or a correlation-name (see also the section Basic Syntactical Items). Qualified names are often clearer than unqualified names and sometimes they are essential.

Note:
A table name in this context must not be qualified explicitly with an authorization identifier. Use a correlation name instead if you need a qualified table name.

If a column is referenced by a *table-name* or *correlation-name*, it must be contained in the corresponding table. If neither a *table-name* nor a *correlation-name* is specified, the respective column must be in one of the tables specified in the FROM clause.

aggregate-function

```
{ COUNT { (*)                          } }
{       { (DISTINCT column-reference)  } }
{ { AVG } { (DISTINCT column-reference) } }
{ { MAX } { ( [ALL] scalar-expression)  } }
{ { MIN }                                }
{ { SUM }                                }
```

SQL provides a number of special functions to enhance its basic retrieval power. The so-called SQL aggregate functions currently available and supported by Natural are:

- **AVG** which gives the average of the values in a column,
- **COUNT** which gives the number of values in a column,
- **MAX** which gives the highest value in a column,
- **MIN** which gives the lowest value in a column,
- **SUM** which gives the sum of the values in a column.

 Copyright © Software AG 2002

Apart from COUNT(*), each of these functions operates on the collection of scalar values in an argument (that is, a single column or a *scalar-expression*) and produces a scalar value as its result.

Example:

```
DEFINE DATA LOCAL
1 AVGAGE   (I2)
END-DEFINE
...
SELECT AVG (AGE)
  INTO AVGAGE
  FROM SQL-PERSONNEL
  ...
```

In general, the argument can optionally be preceded by the keyword DISTINCT to eliminate redundant duplicate values before the function is applied.

If DISTINCT is specified, the argument must be the name of a single column; if DISTINCT is omitted, the argument can consist of a general *scalar-expression*.

DISTINCT is not allowed with the special function COUNT(*), which is provided to count all rows without eliminating any duplicates.

special-register

```
{ USER                 }
{ CURRENT TIMEZONE     }
{ CURRENT DATE         }
{ CURRENT TIME         }
{ CURRENT TIMESTAMP    }
{ CURRENT SQLID        }
{ CURRENT PACKAGESET   }
{ CURRENT SERVER       }
```

A reference to a *special-register* returns a scalar value.

With the exception of USER, *special-registers* do not conform to standard SQL and are therefore supported by the Natural SQL Extended Set only.

scalar-function

```
{ CHAR
  COALESCE
  DATE
  DAY
  DAYS
  DECIMAL
  DIGITS
  FLOAT
  HEX
  HOUR
  INTEGER
  LENGTH
  MICROSECOND
  MINUTE
  MONTH
  NULLIF
  SECOND
  STRIP
  SUBSTR
  TIME
  TIMESTAMP
  TRANSLATE
  VALUE
  VARGRAPHIC
  YEAR }
```

A *scalar-function* is a built-in function that can be used in the construction of scalar computational expressions. The above *scalar-functions* are supported by the Natural SQL Extended Set.

units

```
{ YEAR
  YEARS
  MONTH
  MONTHS
  DAY
  DAYS
  HOUR
  HOURS
  MINUTE
  MINUTES
  SECOND
  SECONDS
  MICROSECOND
  MICROSECONDS }
```

Units do not conform to standard SQL and are therefore supported by the Natural SQL Extended Set only.

 Copyright © Software AG 2002

case-expression

```
CASE { searched-when-clause ... } [ELSE { NULL } ] END
     { simple-when-clause       }      { scalar-expression }
```

case-expressions do not conform to standard SQL and are therefore supported by the Natural SQL Extended Set only.

searched-when-clause

```
WHEN search-condition THEN { NULL }
                           { scalar-expression }
```

See details on *search-condition*.

simple-when-clause

```
scalar-expression { WHEN scalar-expression THEN { NULL } } ...
                                                { scalar-expression }
```

Search Conditions

search-condition

```
{ [NOT] { predicate          }
        { (search-condition) }
{ search-condition { AND } search-condition }
                   { OR  }
```

A *search-condition* can consist of a simple *predicate* or of multiple *search-conditions* combined with the Boolean operators AND, OR and NOT, and parentheses if required to indicate a desired order of evaluation.

Example:

```
DEFINE DATA LOCAL
01 NAME   (A20)
01 AGE    (I2)
END-DEFINE
...
SELECT *
  INTO NAME, AGE
  FROM SQL-PERSONNEL
  WHERE AGE = 32 AND NAME > 'K'
END-SELECT
...
```

 Copyright © Software AG 2002

predicate

```
{ scalar-expression comparison { scalar-expression }
                               { subquery        }

  scalar-expression [NOT] BETWEEN scalar-expression AND scalar-expression
  column-reference [NOT] LIKE { atom            } [ESCAPE atom]
                              { special-register }
  column-reference IS [NOT] NULL
                                   { subquery        }
  scalar-expression [NOT] IN { ( { atom            } ,... ) }
                                   { special-register }
                                   { ALL }
  scalar-expression comparison { ANY } subquery
                                   { SOME }
  EXISTS subquery                                           }
```

A *predicate* specifies a condition that can be "true", "false" or "unknown". In a *search-condition*, a *predicate* can consist of a simple or complex comparison operation or other kinds of conditions.

Example:

```
SELECT NAME, AGE
  INTO VIEW PERS
  FROM SQL-PERSONNEL
  WHERE AGE BETWEEN 20 AND 30
    0R AGE IN ( 32, 34, 36 )
    AND NAME LIKE '%er'
    ...
```

Note:
The percent sign (%) may conflict with Natural terminal commands. If so, you must define a terminal command control character different from "%" (see also the session parameter CF in the Natural Reference documentation).

The individual predicates are explained on the following pages (for further information on predicates, please refer to the relevant literature). According to the syntax above, they are called as follows:

- Comparison Predicate
- BETWEEN Predicate
- LIKE Predicate
- NULL Predicate
- IN Predicate
- Quantified Predicate

- EXISTS Predicate

Comparison Predicate

```
scalar-expression comparison { scalar-expression }
                             {     subquery      }
```

A comparison predicate compares two values.

See information on scalar-expression.

comparison

```
{ = }
{ < }
{ > }
{ <= }
{ >= }
{ <> }
{ ¬ = }
{ ¬ > }
{ ¬ < }
```

Comparison can be any of the following operators:

=	equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
< >	not equal to
¬ =	not equal to
¬ >	not greater than
¬ <	not less than

 Copyright © Software AG 2002

subquery

```
(select-expression)
```

A *subquery* is a *select-expression* that is nested inside another such expression.

Example:

```
DEFINE DATA LOCAL
1 #NAME    (A20)
1 #PERSNR  (I4)
END-DEFINE
...
SELECT NAME, PERSNR
  INTO #NAME, #PERSNR
  FROM SQL-PERSONNEL
  WHERE PERSNR IN
    ( SELECT PERSNR
        FROM SQL-AUTOMOBILES
        WHERE COLOR = 'black' )
        ...
END-SELECT
```

See further information on Select Expressions.

BETWEEN Predicate

```
scalar-expression [NOT] BETWEEN scalar-expression AND scalar-expression
```

A BETWEEN predicate compares a value with a range of values.

See information on scalar-expression.

LIKE Predicate

```
column-reference [NOT] LIKE { atom | special-register } [ESCAPE atom]
```

A LIKE predicate searches for strings that have a certain pattern.

For information on *column-reference* , *atom* and *special-register*, see the section Scalar Expressions.

NULL Predicate

```
column-reference IS [NOT] NULL
```

A NULL predicate tests for null values.

See information on column-reference.

IN Predicate

```
scalar-expression [NOT] IN { subquery
                           ( { atom
                             { special-register }...  )
```

An IN predicate compares a value with a collection of values.

For information on *scalar-expression*, *atom* and *special-register*, see the section Scalar Expressions.

See information on subquery.

Quantified Predicate

```
scalar-expression comparison { ALL
                             { ANY
                             { SOME } subquery
```

A quantified predicate compares a value with a collection of values.

See information on scalar-expression, on comparison, and on subquery.

EXISTS Predicate

```
EXISTS subquery
```

An EXISTS predicate tests for the existence of certain rows.

The EXISTS predicate evaluates to true only if the result of evaluating the *subquery* is not empty; that is, if there exists at least one record (row) in the FROM table of the *subquery* satisfying the search condition of the WHERE clause of this *subquery*.

Example of EXISTS:

 Copyright © Software AG 2002

```
DEFINE DATA LOCAL
1 #NAME    (A20)
END-DEFINE
...
SELECT NAME
  INTO #NAME
  FROM SQL-PERSONNEL
  WHERE EXISTS
    ( SELECT *
      FROM SQL-EMPLOYEES
      WHERE PERSNR > 1000
        AND NAME < 'L' )
        ...
END-SELECT
...
```

See information on subquery.

Select Expressions

select-expression

```
SELECT selection table-expression
```

A *select-expression* specifies a result table.

selection

```
[ ALL      ] { {scalar-expression [[AS] correlation-name]},... }
[ DISTINCT ] {                  *                                }
```

The *selection* specifies the items to be selected.

ALL/DISTINCT

Duplicate rows are not automatically eliminated from the result of a *select-expression*. To request this, specify the keyword DISTINCT.

The alternative to DISTINCT is ALL. ALL is assumed if neither is specified.

scalar-expression

Instead of, or as well as, simple column names, a selection can also include general *scalar-expressions* containing scalar operators and scalar functions which provide computed values (see also the section Scalar Expressions).

Example:

```
SELECT NAME, 65 - AGE
  FROM SQL-PERSONNEL
  ...
```

correlation-name

A *correlation-name* can be assigned to a *scalar-expression* as alias name for a result column.

The *correlation-name* need not be unique. If no *correlation-name* is specified for a result column, the corresponding *column-name* will be used (if the result column is derived from a column name; if not, the result table will have no name). The name of a result column may be used, for example, as column name in the ORDER BY clause of a SELECT statement.

 Copyright © Software AG 2002

Asterisk Notation - *

All columns of all tables specified in the FROM clause are selected.

Example:

```
SELECT *
  FROM SQL-PERSONNEL, SQL-AUTOMOBILES
  ...
```

table-expression

```
FROM table-reference,...
        [WHERE search-condition]
        [GROUP BY column-reference,...]
        [HAVING search-condition]
```

The *table-expression* specifies from where and according to what criteria rows are to be selected.

FROM Clause

```
FROM table-reference,...
```

table-reference

```
{ table-name [[AS] correlation-name]
  subquery [AS] correlation-name
           joined-table            }
```

The tables specified in the FROM clause must contain the column fields used in the selection list.

You can either specify a single table or produce an intermediate table resulting from a subquery or a "join" operation (see below).

Since various tables (that is, DDMs) can be addressed in one FROM clause and since a *table-expression* can contain several FROM clauses if *subqueries* are specified, the database ID (DBID) of the first DDM specified in the first FROM clause of the whole expression is used to identify the underlying database involved.

Optionally a *correlation-name* can be assigned to a *table-name*. For a *subquery*, a *correlation-name* must be assigned.

joined-table

```
{ table-reference [ INNER | { LEFT | RIGHT | FULL } [OUTER] ] JOIN table-reference ON join-condition }
                                  (joined-table)
```

A *joined-table* specifies an intermediate table resulting from a "join" operation.

The "join" can be an INNER, LEFT OUTER, RIGHT OUTER or FULL OUTER JOIN. If you do not specify anything, "INNER" applies.

Multiple "join" operations can be nested; that is, the tables which create the intermediate result table can themselves be intermediate result tables of a JOIN operation or a *subquery*; and the latter, in turn, can also have a *joined-table* or another *subquery* in its FROM clause.

join-condition

```
join-expression comparison join-expression [AND join-condition]...
```

AND

Multiple *join-conditions* can be combined with AND.

For a FULL OUTER JOIN, only the equal sign (=) is allowed as *comparison*. See details on comparison.

The first *join-expression* must refer to the first *table-reference*, the second *join-expression* must refer to the second *table-reference*.

join-expression

```
{ column-name
  { VALUE | COALESCE } (column-name,...) }
```

Within a *join-expression* only *column-names* and the *scalar-function* VALUE (or its synonym COALESCE) are allowed. See details on column-name.

WHERE Clause

```
[WHERE search-condition]
```

 Copyright © Software AG 2002

The WHERE clause is used a to specify the selection criteria (*search-condition*) for the rows to be selected.

Example:

```
DEFINE DATA LOCAL
01 NAME   (A20)
01 AGE    (I2)
END-DEFINE
...
SELECT *
  INTO NAME, AGE
  FROM SQL-PERSONNEL
  WHERE AGE = 32
END-SELECT
...
```

See further information on *search-condition* .

GROUP BY Clause

[**GROUP BY** *column-reference*,...]

The GROUP BY clause rearranges the table represented by the FROM clause into groups in a way that all rows within each group have the same value for the GROUP BY columns.

Each *column-reference* in the selection list must be either a GROUP BY column or specified within an *aggregate-function*. *Aggregate-functions* are applied to the individual groups (not to the entire table). The result table contains as many rows as groups.

See further details on column-reference and aggregate-function.

Example:

```
DEFINE DATA LOCAL
1 #AGE     (I2)
1 #NUMBER  (I2)
END-DEFINE
...
SELECT AGE , COUNT(*)
  INTO #AGE, #NUMBER
  FROM SQL-PERSONNEL
  GROUP BY AGE
  ...
```

If the GROUP BY clause is preceded by a WHERE clause, all rows that do not satisfy the WHERE clause are excluded before any grouping is done.

HAVING Clause

[**HAVING** *search-condition*]

If the HAVING clause is specified, the GROUP BY clause should also be specified.

Just as the WHERE clause is used to exclude rows from a result table, the HAVING clause is used to exclude groups and therefore also based on a *search-condition*. *Scalar-expressions* in a HAVING clause must be single-valued per group.

See further details on scalar-expression and *search-condition* .

Example:

```
DEFINE DATA LOCAL
1 #NAME     (A20)
1 #AVGAGE   (I2)
1 #NUMBER   (I2)
END-DEFINE
...
SELECT NAME, AVG(AGE), COUNT(*)
  INTO #NAME, #AVGAGE, #NUMBER
  FROM SQL-PERSONNEL
  GROUP BY NAME
  HAVING COUNT(*) > 1
  ...
```

 Copyright © Software AG 2002

Flexible SQL

- Text Variables

In addition to the SQL syntax described in the previous sections, flexible SQL allows you to use arbitrary SQL syntax.

"<<" and ">>" Characters

Flexible SQL is enclosed in "<<" and ">>" characters. It can include arbitrary SQL text and host variables. Within flexible SQL, host variables **must** be prefixed by a colon (:).

The flexible SQL string can cover several statement lines. Comments are possible, too (see also the statement PROCESS SQL).

Flexible SQL can be used as a replacement for any of the following syntactical SQL items:

- atom
- column-reference
- scalar-expression
- predicate

Flexible SQL can also be used between the clauses of a select expression:

```
SELECT selection
  << ... >>
  INTO ...
  FROM ...
  << ... >>
  WHERE ...
  << ... >>
  GROUP BY ...
  << ... >>
  HAVING ...
  << ... >>
  ORDER BY ...
  << ... >>
```

Note:
The SQL text used in flexible SQL is not recognized by the Natural compiler. The SQL text (with replaced host variables) is simply copied into the SQL string passed to the database system. Syntax errors in flexible SQL are detected at runtime when the database executes the corresponding statement.

Example 1

```
SELECT NAME
FROM SQL-EMPLOYEES
WHERE << MONTH (BIRTH) >>  =  << MONTH (CURRENT_DATE) >>
```

Example 2:

```
SELECT NAME
FROM SQL-EMPLOYEES
WHERE << MONTH (BIRTH) = MONTH (CURRENT_DATE) >>
```

Example 3:

```
SELECT NAME
FROM SQL-EMPLOYEES
WHERE SALARY > 50000
<< INTERSECT
   SELECT NAME
   FROM SQL-EMPLOYEES
   WHERE DEPT = 'DEPT10'
>>
```

Text Variables

<< **:T:***host-variable* [**LINDICATOR:** *host-variable*]>>

Within flexible SQL, you can also specify so-called "text variables".

:T:

A text variable is a *host-variable* prefixed by ":T:". It must be in alphanumeric format.

At runtime, a text variable within an SQL statement will be replaced by its contents that is, the text string contained in the text variable will be inserted into the SQL string.

After the replacement, trailing blanks will be removed from the inserted text string.

You have to make sure yourself that the content of a text variable results in a syntactically correct SQL string. In particular, the content of a text variable must not contain *host-variables*.

A statement containing a text variable will always be executed in dynamic SQL mode.

LINDICATOR Option

The text variable can be followed by the keyword LINDICATOR and a length indicator variable (that is, a *host-variable* prefixed by colon).

The length indicator variable has to be of format/length I2.

If no LINDICATOR variable is specified, the entire content of the text variable will be inserted into the SQL string.

If you specify a LINDICATOR variable, only the first *n* characters (*n* being the value of the LINDICATOR variable) of the text variable content will be inserted into the SQL string. If the number in the LINDICATOR variable is greater than the length of the text variable content, the entire text variable content will be inserted. If the number in the LINDICATOR variable is negative or 0, nothing will be inserted.

See general information on host-variables.

Example using Text Variable:

 Copyright © Software AG 2002

```
DEFINE DATA LOCAL
01 TEXTVAR (A200)
01 TABLES VIEW OF SYSIBM-SYSTABLES
   02 NAME
   02 CREATOR
END-DEFINE
*
MOVE 'WHERE NAME > ''SYS'' AND CREATOR = ''SYSIBM''' TO TEXTVAR
*
SELECT * INTO VIEW TABLES
  FROM SYSIBM-SYSTABLES
  << :T:TEXTVAR >>
  DISPLAY TABLES
END-SELECT
*
END
```

The generated SQL statement (as displayed with the LISTSQL system command) will look as follows:

```
SELECT NAME, CREATOR FROM SYSIBM.SYSTABLES:T: FOR FETCH ONLY
```

The executed SQL statement will look as follows:

```
SELECT TABNAME, CREATOR FROM SYSIBM.SYSTABLES
  WHERE TABNAME > 'SYS' AND CREATOR = 'SYSIBM'
```

CALLDBPROC

```
CALLDBPROC dbproc ddm-name
    [USING] [ parameter [ AD = { M } ] ]...
                               { O }
                               { A }
    [RESULT SETS result-set...]
    [GIVING sqlcode]
    [ CALLMODE = { NONE    } ]
                 { NATURAL }
```

Function

The statement CALLDBPROC is used to invoke a stored procedure of the SQL database system to which Natural is connected.

The stored procedure can be either a Natural subprogram or a program written in another programming language.

In addition to the passing of parameters between the invoking object and the stored procedure, CALLDBPROC supports "result sets"; these make it possible to return a larger amount of data from the stored procedure to the invoking object than would be possible via parameters.

The result sets are "temporary result tables" which are created by the stored procedure and which can be read and processed by the invoking object via a READ RESULT SET statement.

Note:
In general, the invoking of a stored procedure could be compared with the invoking of a Natural subprogram: when the CALLDBPROC statement is executed, control is passed to the stored procedure; after processing of the stored procedure, control is returned to the invoking object and processing continues with the statement following the CALLDBPROC statement.

dbproc

As dbproc you specify the name of the stored procedure to be invoked. The name can be specified either as an alphanumeric variable or as a constant (enclosed in apostrophes).

The name must adhere to the rules for stored procedure names of the target database system.

If the stored procedure is a Natural subprogram, the actual procedure name must not be longer than 8 characters.

 Copyright © Software AG 2002

ddm-name

The name of a DDM must be specified to provide the "address" of the database which executes the stored procedure. For more information see ddm-name.

parameter

As *parameter*, you can specify parameters which are passed from the invoking object to the stored procedure.

A *parameter* can be a host-variable (optionally with INDICATOR and LINDICATOR clauses), a constant, or the keyword NULL.

See further details on host-variable.

AD=

If the *parameter* is a *host-variable*, you can mark it as non-modifiable (AD=O), as modifiable (AD=M) or as for input only (AD=A).

- **AD=M** - By default, the passed value of a parameter can be changed in the stored procedure and the changed value passed back to the invoking object, where it overwrites the original value. This is the default. (Corresponding procedure notation in DB2 for OS/390: INOUT.)
- **AD=O** - If you mark a parameter with AD=O, the passed value can be changed in the stored procedure, but the changed value cannot be passed back to the invoking object; that is, the field in the invoking object retains its original value.
  (Corresponding procedure notation in DB2 for OS/390: IN.)
- **AD=A** - If you mark a parameter with AD=A, its value will not be passed to the stored procedure, but it will receive a value from the stored procedure.
  (Corresponding procedure notation in DB2 for OS/390: OUT.)

If the *parameter* is a *constant*, AD cannot be explicitly specified. For constants AD=O always applies.

result-set

As result-set you specify a field in which a result-set locator is to be returned.

A result-set has to be a variable of format/length I4.

The value of a result-set variable is merely a number which identifies the result set and which can be referenced in a subsequent READ RESULT SET statement.

The sequence of the result-set values correspond to the sequence of the result sets returned by the stored procedure.

The contents of the result sets can be processed by a subsequent READ RESULT SET statement.

If no result set is returned, the corresponding result-set variable will contain "0".

On mainframe computers, multiple result sets can be specified. On all other platforms, only one result set can be specified.

Copyright © Software AG 2002

GIVING sqlcode

This option may be used to obtain the SQL code of the SQL CALL statement invoking the stored procedure.

If this option is specified and the SQL code of the stored procedure is not "0", no Natural error message will be issued. In this case, the action to be taken in reaction to the SQL code value has to be coded in the invoking Natural object.

The sqlcode field has to be a variable of format/length I4.

If the GIVING sqlcode option is omitted, a Natural error message will be issued if the SQL code of the stored procedure is not "0".

CALLMODE

If the stored procedure is a Natural subprogram, CALLMODE=NATURAL has to be specified.

Note:
CALLMODE=NATURAL also has an impact on internal parameters that are passed to/from the stored procedure; see the corresponding Natural database interface documentation for details.
CALLMODE=NATURAL is only available on mainframe computers.

Example

The following example shows a Natural program that calls the stored procedure ’demo_proc’ to retrieve all names of table PERSON that belong to a given range.

Three parameter fields are passed to ’demo_proc’: the first and second parameters pass starting and ending values of the range of names to the stored procedure, and the third parameter receives a name that meets the criterion.

In this example, the names are returned in a result set that is processed using the READ RESULT SET statement.

 Copyright © Software AG 2002

```
DEFINE DATA LOCAL
1 PERSON VIEW OF DEMO-PERSON
  2 PERSON_ID
  2 LAST_NAME
1 #BEGIN (A2) INIT <'AB'>
1 #END  (A2) INIT <'DE'>
1 #RESPONSE (I4)
1 #RESULT (I4)
1 #NAME (A20)
END-DEFINE

...

CALLDBPROC 'demo_proc' DEMO-PERSON #BEGIN (AD=O) #END (AD=O) #NAME (AD=A)
    RESULT SETS #RESULT
    GIVING #RESPONSE

READ RESULT SET #RESULT INTO #NAME FROM DEMO-PERSON
    GIVING #RESPONSE
  DISPLAY #NAME
END-RESULT

...

END
```

See the corresponding Natural database interface documentation in the Natural for Mainframes documentation for further examples.

COMMIT

```
COMMIT
```

Function

The SQL COMMIT statement corresponds to the END TRANSACTION statement. It indicates the end of a logical transaction and releases all data locked during the transaction. All data modifications are committed and made permanent.

Example:

```
...
DELETE FROM SQL-PERSONNEL WHERE NAME = 'SMITH'
COMMIT
 ...
```

As all cursors are closed when a logical unit of work ends, a COMMIT statement must not be placed within a database modification loop; instead, it has to be placed outside such a loop or after the outermost loop of nested loops.

Consideration for Non-Natural-Programs

If an external program written in another standard programming language is called from a Natural program, this external program should not contain its own COMMIT statement if the Natural program issues database calls, too. The calling Natural program should issue the COMMIT statement on behalf of the external program.

 Copyright © Software AG 2002

DELETE

Syntax 1 - Searched DELETE

```
DELETE FROM table-name [correlation-name] [WHERE search-condition]
```

Syntax 2 - Positioned DELETE

```
DELETE FROM table-name WHERE CURRENT OF CURSOR [(r)]
```

Function

The SQL DELETE statement is used to delete either rows in a table without using a cursor ("searched" DELETE) or rows in a table to which a cursor is positioned ("positioned" DELETE).

The "searched" DELETE statement is a stand-alone statement not related to any SELECT statement. With a single statement you can delete zero, one, multiple or all rows of a table. The rows to be deleted are determined by a *search-condition* that is applied to the table. Optionally, the table name can be assigned a *correlation-name*.

The "positioned" DELETE statement always refers to a cursor within a database loop. Thus, the table referenced by a positioned DELETE statement must be the same as the one referenced by the corresponding SELECT statement; otherwise an error message is returned. A positioned DELETE cannot be used with a non-cursor selection. The functionality of the positioned DELETE statement corresponds to that of the "normal" Natural DELETE statement.

Note:
The number of rows that have actually been deleted with a "searched" DELETE can be ascertained by using the system variable *ROWCOUNT (see Natural Reference documentation).

FROM Clause

The FROM clause specifies the table from which the rows are to be deleted.

WHERE Clause

The WHERE clause is used to specify the selection criteria for the rows to be deleted.

If no WHERE clause is specified, the entire table is deleted.

Statement Reference - r

The "(*r*)" notation is used to reference the statement which was used to select the row to be deleted. If no statement reference is specified, the DELETE statement is related to the innermost active processing loop in which a database record was selected.

 Copyright © Software AG 2002

INSERT

```
                                  (*) VALUES (VIEW view-name)
INSERT INTO table-name { [(column-list)] { VALUES ( { VIEW view-name } ) } }
                                                   { insert-item-list }
                                       select-expression
```

Function

The SQL INSERT statement is used to add one or more new rows to a table.

INTO Clause

In the INTO clause, the table is specified into which the new rows are to be inserted.

See further information on *table-name*.

column-list

```
column-name,...
```

In the *column-list*, one or more columns can be specified, which are to be supplied with values in the row currently inserted.

If a *column-list* is specified, the sequence of the columns must match with the sequence of the values either specified in the *insert-item-list* or contained in the specified view (see below).

If the *column-list* is omitted, the values in the *insert-item-list* or in the specified view are inserted according to an implicit list of all the columns in the order they exist in the table.

VALUES Clause

With the VALUES clause, you insert a *single* row into the table. Depending on whether an asterisk (*) or a *column-list* has been specified, the VALUES clause can take one of the following forms:

VALUES Clause with Preceding Asterisk Notation

```
VALUES (VIEW view-name)
```

If asterisk notation is specified, a view **must** be specified in the VALUES clause. With the field values of this view, a new row is inserted into the specified table using the field names of the view as column names of the row.

VALUES Clause with Preceding Column List

```
VALUES ({VIEW view-name}
        {insert-item-list})
```

If a *column-list* is specified and a view is referenced in the VALUES clause, the number of items specified in the column list must correspond to the number of fields defined in the view.

If no *column-list* is specified, the fields defined in the view are inserted according to an implicit list of all the columns in the order they exist in the specified table.

insert-item-list

In the *insert-item-list*, you can specify one or more values to be assigned to the columns specified in the *column-list*. The sequence of the specified values must match the sequence of the columns.

If no *column-list* is specified, the values in the *insert-item-list* are inserted according to an implicit list of all the columns in the order they exist in the table.

The values to be specified in the *insert-item-list* can be *constants*, *parameters*, *special-registers* or NULL.

See the section Basic Syntactical Items for information on *view-name*, *constant* and *parameter*. See also the information on *special-register*.

If the value NULL has been assigned, this means that the addressed field is to receive no value (not even the value "0" or "blank").

Example - INSERT Single Row:

```
...
INSERT INTO SQL-PERSONNEL (NAME,AGE)
  VALUES ('ADKINSON',35)
 ...
```

select-expression

With a *select-expression*, you insert *multiple* rows into a table. The *select-expression* is evaluated and each row of the result table is treated as if the values in this row were specified as values in a VALUES clause of a single-row INSERT operation.

 Copyright © Software AG 2002

See further information on *select-expression*.

Example - INSERT Multiple Rows:

```
...
INSERT INTO SQL-RETIREE (NAME,AGE,SEX)
  SELECT LASTNAME, AGE, SEX
  FROM SQL-EMPLOYEES
  WHERE AGE > 60
...
```

Note:
The number of rows that have actually been inserted can be ascertained by using the system variable *ROWCOUNT (see Natural Reference documentation).

PROCESS SQL

```
PROCESS SQL ddm-name <<statement-string>>
```

Function

The PROCESS SQL statement is used to issue SQL statements to the underlying database.

ddm-name

The name of a DDM must be specified to provide the "address" of the database to which the statement string (see below) is to be addressed. For more information, see ddm-name.

statement-string

The statements which can be specified in the *statement-string* are the same statements which can be issued with the SQL statement "EXECUTE" (see also Flexible SQL).

Note:
To avoid transaction synchronization problems between the Natural environment and the underlying database, the COMMIT and ROLLBACK statements must not be used within PROCESS SQL.

The statement string can cover several statement lines without any continuation character to be specified. Comments at the end of a line as well as entire comment lines are possible.

The statement string can also include parameters.

Parameters

```
[:U|:G] : host-variable [INDICATOR : host-variable] [LINDICATOR : host-variable]
```

Unlike with the parameter described, in this context *host-variables* must be prefixed by a colon (:). In addition, they can be preceded by a further qualifier (":U" or :"G").

See further details on host-variable.

:U:host-variable

The prefix ":U" qualifies the host variable as a so-called "Using" variable. Such a variable indicates that its value is to be *passed to* the database. ":U" is the default specification.

 Copyright © Software AG 2002

:G:host-variable

The prefix ":G" qualifies the host variable as a so-called "Giving" variable. Such a variable indicates that it is to *receive* a value *from* the database.

Examples

Examples for DB2 (under OS/390):

```
PROCESS SQL DB2_DDM << CONNECT TO :LOCATION >>
```

```
PROCESS SQL DB2_DDM << SET :G:LOCATION = CURRENT SERVER >>
```

Example for Adabas D:

```
PROCESS SQL ADABAS_D_DDM << LOCK TABLE EMPLOYEES IN SHARE MODE >>
```

Example of Calling a Procedure Stored in Adabas D:

The called procedure computes the sum of two numbers.

```
...
COMPUTE #N1 = 1
COMPUTE #N2 = 2
COMPUTE #SUM = 0
...
PROCESS SQL ADABAS_D_DDM << DBPROCEDURE DEMO.SUM (:#N1, :#N2, :G:#SUM) >>
...
WRITE #N1 '+' #N2 ' =' #SUM
...
```

ENTIRE ACCESS Options

With ENTIRE ACCESS, you can also specify the following as *statement-string*:

- **SET SQLOPTION *option = value***
- **SQLCONNECT *option = value***
- **SQLDISCONNECT**

These options are only possible with ENTIRE ACCESS, and are described in the ENTIRE ACCESS documentation.

READ RESULT SET

```
READ [(limit)] RESULT-SET result-set INTO { VIEW view-name } FROM ddm-name
                                          { parameter ,... }
     [GIVING sqlcode]
END-RESULT
```

Function

The statement READ RESULT SET is used to read a result set which was created by a stored procedure that was invoked by a previous CALLDBPROC statement.

The READ RESULT SET statement can only be used in conjunction with a CALLDBPROC statement.

As result-set you specify a result-set locator variable filled by a preceding CALLDBPROC statement. Result-set has to be a variable of format/length I4.

Note:
If a syncpoint operation takes place between the CALLDBPROC statement and the READ RESULT SET statement, the result sets can no longer be accessed by the READ RESULT SET statement.

limit

You can limit the number of rows to be read. You can specify the limit either as a numeric constant (0 to 99999999) or as a variable of format N, P or I.

ddm-name

As *ddm-name* you specify the name of the DDM which is used to "address" the database executing the stored procedure. For more information, see ddm-name.

GIVING sqlcode

This option may be used to obtain the SQL code of the SQL "fetch" operation used to process the result set.

If this option is specified and the SQL code of the SQL operation is not "0", no Natural error message will be issued. In this case, the action to be taken in reaction to the SQL code value has to be coded in the invoking Natural object.

The *sqlcode* field has to be a variable of format/length I4.

If the GIVING *sqlcode* option is omitted, a Natural error message will be issued if the SQL code is not "0".

 Copyright © Software AG 2002

Example

See the example in the CALLDBPROC statement.

In addition, see the corresponding Natural database interface documentation in the Natural for Mainframes documentation.

ROLLBACK

```
ROLLBACK
```

Function

The SQL statement ROLLBACK corresponds to the Natural statement BACKOUT TRANSACTION. It undoes all database modifications made since the beginning of the last recovery unit. A recovery unit may start either after the beginning of a session or after the last SYNCPOINT, COMMIT, END TRANSACTION or BACKOUT TRANSACTION statement. This statement also releases all records held during the transaction.

If a program tries to backout updates which have already been committed by a terminal I/O, a corresponding Natural error message (NAT3711) is returned.

Example:

```
...
DELETE FROM SQL-PERSONNEL WHERE NAME = 'SMITH'
ROLLBACK
 ...
```

As all cursors are closed when a logical unit of work ends, a ROLLBACK statement must not be placed within a database modification loop; instead, it has to be placed outside such a loop or after the outermost loop of nested loops.

Consideration for Non-Natural Programs

If an external program written in another standard programming language is called from a Natural program, this external program should not contain its own ROLLBACK statement if the Natural program issues database calls, too. The calling Natural program should issue the ROLLBACK statement on behalf of the external program.

 Copyright © Software AG 2002

SELECT

According to the standard SQL functionality, Natural supports both the cursor-oriented selection that is used to retrieve an arbitrary number of rows and the non-cursor selection (singleton SELECT) that retrieves at most one single row.

With the "SELECT ... END-SELECT" construction, Natural uses the same database loop processing as with the FIND statement.

Cursor-Oriented Selection

```
SELECT selection INTO { parameter ,...                                  } table-expression
                      { VIEW {view-name [correlation-name]},...         }
       [ { UNION     } [ALL] [(] SELECT selection table-expression [)] ]...
       [ { EXCEPT    }                                                 ]
       [ { INTERSECT }                                                 ]
       [ORDER BY { integer       } [ASC] ]
                 { column-reference } [DESC] ]
       [OPTIMIZE FOR integer ROWS]
       [ WITH { CS } ]
              { RR }
              { UR }
       [WITH HOLD]
       [WITH RETURN]
       [IF-NO-RECORDS-FOUND-clause]
       statement...
{ END-SELECT                      }
{ LOOP (reporting mode only)      }
```

Like the FIND statement, a cursor-oriented selection is used to select a set of rows (records) from one or more database tables, based on a search criterion. In addition, no cursor management is required from the application program; it is automatically handled by Natural.

Non-Cursor Selection

```
SELECT SINGLE

        selection INTO { parameter ,...                                  } table-expression
                       { VIEW {view-name [correlation-name] },...        }
        [ WITH { CS } ]
               { RR }
               { UR }

        [IF-NO-RECORDS-FOUND-clause]
        statement...

{       END-SELECT                  }
{ LOOP (reporting mode only)        }
```

The SELECT SINGLE statement supports the functionality of a non-cursor selection (singleton SELECT); that is, a select expression that retrieves at most one row without using a cursor. It cannot be referenced by a positioned UPDATE or DELETE statement.

---

table-expression

The *table-expression* consists of a FROM clause and an optional WHERE clause. The GROUP BY and HAVING clauses are not permitted.

Example 1:

```
DEFINE DATA LOCAL
01 #NAME      (A20)
01 #FIRSTNAME (A15)
01 #AGE       (I2)
...
END-DEFINE
...
SELECT NAME, FIRSTNAME, AGE
  INTO #NAME, #FIRSTNAME, #AGE
  FROM SQL-PERSONNEL
    WHERE NAME IS NOT NULL
      AND AGE > 20
...
  DISPLAY #NAME #FIRSTNAME #AGE
END-SELECT
...
END
```

Example 2:

 Copyright © Software AG 2002

```
DEFINE DATA LOCAL
01 #COUNT    (I4)
...
END-DEFINE
...
SELECT SINGLE COUNT(*) INTO #COUNT FROM SQL-PERSONNEL
...
```

See further information on selection and table-expression.

Note:
In the following, the term "SELECT statement" is used as a synonym for the whole query-expression consisting of multiple select expressions concatenated with UNION operations.

INTO Clause

```
INTO { parameter ,...                                  }
     { VIEW {view-name [correlation-name]},...         }
```

The INTO clause is used to specify the target fields in the program which are to be filled with the result of the selection. The INTO clause can specify either single *parameters* or one or more views as defined in the DEFINE DATA statement.

All target field values can come either from a single table or from more than one table as a result of a join operation (see also the section Join Queries).

Note:
In standard SQL syntax, an INTO clause is only used in non-cursor select operations (singleton SELECT) and can be specified only if a single row is to be selected. In Natural, however, the INTO clause is used for both cursor-oriented and non-cursor select operations.

The *selection* can also merely consist of an asterisk (*). In a standard select expression, this is a shorthand for a list of all column names in the table(s) specified in the FROM clause. In the Natural SELECT statement, however, the same syntactical item "SELECT *" has a different semantic meaning: all the items listed in the INTO clause are also used in the selection. Their names must correspond to names of existing database columns.

Examples:

```
DEFINE DATA LOCAL
01 PERS VIEW OF SQL-PERSONNEL
  02 NAME
  02 AGE
END-DEFINE
...
SELECT *
  INTO NAME, AGE
```

```
...
SELECT *
  INTO VIEW PERS
```

These examples are equivalent to the following ones:

```
...
SELECT NAME, AGE
  INTO NAME, AGE
```

```
...
SELECT NAME, AGE
  INTO VIEW PERS
```

 Copyright © Software AG 2002

parameter

If single parameters are specified as target fields, their number and formats must correspond to the number and formats of the *columns* and/or *scalar-expressions* specified in the corresponding selection as described above (see details on Scalar Expressions).

Example:

```
DEFINE DATA LOCAL
01 #NAME   (A20)
01 #AGE    (I2)
END-DEFINE
...
SELECT NAME, AGE
  INTO #NAME, #AGE
  FROM SQL-PERSONNEL
...
```

The target fields #NAME and #AGE, which are Natural program variables, receive the contents of the table columns NAME and AGE.

VIEW Clause

VIEW {*view-name* [*correlation-name*] }**,...**

If one or more views are referenced in the INTO clause, the number of items specified in the *selection* must correspond to the number of fields defined in the view(s) (not counting group fields, redefining fields and indicator fields).

Note:
Both the Natural target fields and the table columns must be defined in a Natural DDM. Their names, however, can be different, since assignment is made according to their sequence.

Example of INTO Clause with View:

```
DEFINE DATA LOCAL
01 PERS VIEW OF SQL-PERSONNEL
  02 NAME
  02 AGE
END-DEFINE
...
SELECT FIRSTNAME, AGE
  INTO VIEW PERS
  FROM SQL-PERSONNEL
...
```

The target fields NAME and AGE, which are part of a Natural view, receive the contents of the table columns FIRSTNAME and AGE.

correlation-name

If the VIEW clause is used within a "SELECT *" construction where multiple tables are to be joined, *correlation-names* are required if the specified view contains fields that reference columns which exist in more than one of these tables. In order to know which column to select, all these columns are qualified by the specified *correlation-name* at generation of the selection list. The *correlation-name* assigned to a view must correspond to one of the *correlation-names* used to qualify the tables to be joined. See also the section Join Queries.

Example:

```
DEFINE DATA LOCAL
01 PERS VIEW OF SQL-PERSONNEL
  02 NAME
  02 FIRST-NAME
  02 AGE
END-DEFINE
...
SELECT *
  INTO VIEW PERS A
  FROM SQL-PERSONNEL A, SQL-PERSONNEL B
...
```

Query involving UNION

UNION unites the results of two or more *select-expressions*. The columns specified in the individual *select-expressions* must be UNION-compatible; that is, matching in number, type and format.

Redundant duplicate rows are always eliminated from the result of a UNION unless the UNION operator explicitly includes the ALL qualifier. With UNION, however, there is no explicit DISTINCT option as an alternative to ALL.

Example:

```
DEFINE DATA LOCAL
01 PERS VIEW OF SQL-PERSONNEL
  02 NAME
  02 AGE
  02 ADDRESS (1:6)
END-DEFINE
...
SELECT NAME, AGE, ADDRESS
  INTO VIEW PERS
  FROM SQL-PERSONNEL
  WHERE AGE > 55
UNION ALL
SELECT NAME, AGE, ADDRESS
  FROM SQL-EMPLOYEES
  WHERE PERSNR < 100
ORDER BY NAME
...
END-SELECT
...
```

In general, any number of *select expressions* can be concatenated with UNION.

 Copyright © Software AG 2002

The INTO clause must be specified with the first *select-expression* only.

Any ORDER BY clause must appear after the final *select-expression*; the ordering columns must be identified by number, not by name.

ORDER BY Clause

```
ORDER BY { { integer        } [ ASC  ] }
         { { column-reference } [ DESC ] }...
```

The ORDER BY clause arranges the result of a SELECT statement in a particular sequence.

Each ORDER BY clause must specify a column of the result table. In most ORDER BY clauses a column can be identified either by *column-reference* (that is, by an optionally qualified column name) or by column number. In a query involving UNION, a column must be identified by column number. The column number is the ordinal left-to-right position of a column within the *selection*, which means it is an *integer* value. This feature makes it possible to order a result on the basis of a computed column which does not have a name.

Example:

```
DEFINE DATA LOCAL
1 #NAME          (A20)
1 #YEARS-TO-WORK (I2)
END-DEFINE
...
SELECT NAME , 65 - AGE
  INTO #NAME, #YEARS-TO-WORK
  FROM SQL-PERSONNEL
  ORDER BY 2
  ...
```

The order specified in the ORDER BY clause can be either ascending (ASC) or descending (DESC). ASC is the default.

 Copyright © Software AG 2002

Example:

```
DEFINE DATA LOCAL
1 PERS VIEW OF SQL-PERSONNEL
1 NAME
1 AGE
1 ADDRESS   (1:6)
END-DEFINE
...
SELECT NAME, AGE, ADDRESS
  INTO VIEW PERS
  FROM SQL-PERSONNEL
  WHERE AGE = 55
  ORDER BY NAME DESC
  ...
```

See further information on integer values and column-reference in the SQL Statements overview page.

IF NO RECORDS FOUND-clause

Note:
This clause actually does not belong to Natural SQL; it represents Natural functionality which has been made available to SQL loop processing.

Structured Mode Syntax

```
IF NO [RECORDS] [FOUND]
    { ENTER      }
    { statement... }
END-NOREC
```

Reporting Mode Syntax

```
IF NO [RECORDS] [FOUND]
    {          ENTER          }
    {        statement        }
    { DO statement...DOEND    }
```

The IF NO RECORDS FOUND clause is used to initiate a processing loop if no records meet the selection criteria specified in the preceding SELECT statement.

If no records meet the specified selection criteria, the IF NO RECORDS FOUND clause causes the processing loop to be executed once with an "empty" record. If this is not desired, specify the statement ESCAPE BOTTOM within the IF NO RECORDS FOUND clause.

If one or more statements are specified with the IF NO RECORDS FOUND clause, the statements are executed immediately before the processing loop is entered. If no statements are to be executed before entering the loop, the keyword ENTER must be used.

Note:
If the result set of the SELECT statement consists of a single row of NULL values, the IF NO RECORDS FOUND clause is not executed. This could occur if the "selection" list consists solely of one of the "aggregate-functions" SUM, AVG, MIN or MAX on columns, and the set on which these "aggregate-functions" operate is empty.
When you use these "aggregate-functions" in the above-mentioned way, you should therefore check the values of the corresponding null-indicator fields instead of using an IF NO RECORDS FOUND clause.

Database Values

Unless other value assignments are made in the statements accompanying an IF NO RECORDS FOUND clause, Natural resets to empty all database fields which reference the file specified in the current loop.

 Copyright © Software AG 2002

Evaluation of System Functions

Natural system functions are evaluated once for the empty record that is created for processing as a result of the IF NO RECORDS FOUND clause.

Join Queries

A join is a query in which data is retrieved from more than one table. All the tables involved must be specified in the FROM clause.

Example:

```
DEFINE DATA LOCAL
1 #NAME     (A20)
1 #MONEY    (I4)
END-DEFINE
...
SELECT NAME, ACCOUNT
  INTO #NAME, #MONEY
  FROM  SQL-PERSONNEL P, SQL-FINANCE F
  WHERE P.PERSNR = F.PERSNR
    AND F.ACCOUNT > 10000
    ...
```

A join always forms the Cartesian product of the tables listed in the FROM clause and later eliminates from this Cartesian product table all the rows that do not satisfy the join condition specified in the WHERE clause.

Correlation-names can be used to save writing if table names are rather long. Correlation-names must be used when a column specified in the selection list exists in more than one of the tables to be joined in order to know which of the identically named columns to select.

 Copyright © Software AG 2002

UPDATE

Syntax 1 - Searched UPDATE

```
UPDATE { view-name [correlation-name] SET *
       { table-name [correlation-name] SET assignment-list }
       [WHERE search-condition]
```

Syntax 2 - Positioned UPDATE

```
UPDATE { view-name SET *
       { table-name SET assignment-list } WHERE CURRENT OF CURSOR [(r)]
```

Function

The SQL UPDATE statement is used to perform an UPDATE operation on either rows in a table without using a cursor ("searched" UPDATE) or columns in a row to which a cursor is positioned ("positioned" UPDATE).

The "searched" UPDATE statement is a stand-alone statement not related to any SELECT statement. With a single statement you can update zero, one, multiple or all rows of a table. The rows to be updated are determined by a *search-condition* that is applied to the table. Optionally, view and table names can be assigned a *correlation-name*.

The "positioned" UPDATE statement always refers to a cursor within a database loop. Thus, the table or view referenced by a positioned UPDATE statement must be the same as the one referenced by the corresponding SELECT statement; otherwise an error message is returned. A positioned UPDATE cannot be used with a non-cursor selection.

See Basic Syntactical Items for further information on *view-name*, *table-name*, *authorization-identifier* and *correlation-name*.

Note:
The number of rows that have actually been updated with a "searched" UPDATE can be ascertained by using the system variable *ROWCOUNT in the Natural Reference documentation.

SET Clause

If a view has been specified for updating, an asterisk (*) has to be specified in the SET clause, because all columns of the view must be updated.

If a table has been specified for updating, the SET clause must contain either an assignment-list or the name of view which contains the columns to be updated.

assignment-list

```
{column-name = { scalar-expression }}
               {      NULL        }},...
```

In an *assignment-list*, you can assign values to one or more columns. A value can be either a *scalar-expression* or NULL.

See further information on Scalar Expressions.

If the value NULL has been assigned, it means that the addressed field is to contain no value (not even the value "0" or "blank").

WHERE search-condition

The WHERE clause is used to specify the selection criteria for the rows to be updated.

If no WHERE clause is specified, the entire table is updated.

Statement Reference - r

The "(*r*)" notation is used to reference the statement which was used to select the row to be updated. If no statement reference is specified, the UPDATE statement is related to the innermost active processing loop in which a database record was selected.

 Copyright © Software AG 2002

Examples

Example of Searched UPDATE:

```
DEFINE DATA LOCAL
1 PERS VIEW OF SQL-PERSONNEL
2 NAME
2 AGE
...
END-DEFINE
...
ASSIGN AGE = 45
ASSIGN NAME = 'SCHMIDT'
UPDATE PERS SET * WHERE NAME = 'SCHMIDT'
...
```

Example of Searched UPDATE with *assignment-list*:

```
DEFINE DATA LOCAL
1 PERS VIEW OF SQL-PERSONNEL
2 NAME
2 AGE
...
END-DEFINE
...
UPDATE SQL-PERSONNEL SET AGE = AGE + 1 WHERE NAME = 'SCHMIDT'
...
```

Copyright © Software AG 2002

Example of Positioned UPDATE:

```
DEFINE DATA LOCAL
1 PERS VIEW OF SQL-PERSONNEL
2 NAME
2 AGE
...
END-DEFINE
...
SELECT * INTO PERS FROM SQL_PERSONNEL WHERE NAME = 'SCHMIDT'
COMPUTE AGE = AGE + 1
UPDATE PERS SET * WHERE CURRENT OF CURSOR
END-SELECT
...
```

Example of Positioned UPDATE with *assignment-list*:

```
DEFINE DATA LOCAL
1 PERS VIEW OF SQL-PERSONNEL
2 NAME
2 AGE
...
END-DEFINE
...
SELECT * INTO PERS FROM SQL-PERSONNEL WHERE NAME = 'SCHMIDT'
UPDATE SQL-PERSONNEL SET AGE = AGE + 1 WHERE CURRENT OF CURSOR
END-SELECT
...
```

 Copyright © Software AG 2002