



NATURAL

Natural

Programming Guide

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.

© June 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

Programming Guide - Overview	1
Programming Guide - Overview	1
Defining Fields	2
Defining Fields	2
DEFINE DATA Statement	2
Structure of a DEFINE DATA Statement - Level Numbers	4
Level Numbers in View Definitions	4
Level Numbers in Field Groups	4
Level Numbers in Redefinitions	5
User-Defined Variables	6
Names of User-Defined Variables	6
Format and Length of User-Defined Variables	8
User-Defined Constants	9
Numeric Constants	9
Alphanumeric Constants	9
Date and Time Constants	11
Hexadecimal Constants	12
Logical Constants	12
Floating Point Constants	12
Attribute Constants	13
Defining Named Constants	14
Initial Values	15
Default Initial Values	16
The RESET Statement	16
Redefining Fields	17
Arrays	19
Defining Arrays	19
Initial Values for Arrays	20
Assigning Initial Values to One-Dimensional Arrays	20
Assigning Initial Values to Two-Dimensional Arrays	21
A Three-Dimensional Array	25
Arrays as Part of a Larger Data Structure	26
Database Arrays	27
Using Arithmetic Expressions in Index Notation	27
Arithmetic Support for Arrays	27
Data Blocks	29
Defining Data Blocks	30
Block Hierarchies	31
Database Access	32
Database Access	32
DDMs (Data Definition Modules)	32
Displaying a DDM	33
Components of a DDM	33
Database Arrays	34
Multiple-Value Fields	34
Periodic Groups	36
Referencing Multiple-Value Fields and Periodic Groups	37
Multiple-Value Fields Within Periodic Groups	38
Referencing Multiple-Value Fields Within Periodic Groups	39
Referencing the Internal Count of a Database Array	39
DEFINE DATA Views	40
Statements for Database Access	41
READ Statement	41

Syntax	42
Limiting the Number of Records to be Read	43
The STARTING/ENDING Clauses	44
The WHERE Clause	44
FIND Statement	45
Syntax	46
Limiting the Number of Records to be Processed	46
The WHERE Clause	46
IF NO RECORDS FOUND Condition	47
HISTOGRAM Statement	49
Syntax	49
Limiting the Number of Values to be Read	49
The STARTING/ENDING Clauses	49
The WHERE Clause	49
Database Processing Loops	50
Hierarchies of Processing Loops	52
Database Update - Transaction Processing	55
Logical Transaction	55
Record Hold Logic	56
Backing Out a Transaction	57
Restarting a Transaction	57
Statements ACCEPT and REJECT	59
AT START/END OF DATA Statements	62
Output of Data	64
Output of Data	64
Layout of an Output Page - Overview	64
Statements DISPLAY and WRITE	65
DISPLAY Statement	66
WRITE Statement	67
Column Spacing - The SF Parameter and the nX Notation	68
Tab Setting - The nT Notation	70
Line Advance - The / Notation	70
Index Notation (<i>n:n</i>) for Multiple-Value Fields and Periodic Groups	72
Page Titles and Page Breaks	74
Default Page Title	74
Suppress Page Title - The NOTITLE Option	74
Define Your Own Page Title - The WRITE TITLE Statement	74
Logical Page and Physical Page	75
Page Size - The PS Parameter	77
Page Advance - The EJ Parameter	77
Page Advance - The EJECT and NEWPAGE Statements	77
EJECT/NEWPAGE WHEN LESS THAN <i>n</i> LINES LEFT	78
NEWPAGE WITH TITLE	79
Page Trailer - The WRITE TRAILER Statement	80
AT TOP OF PAGE Statement	80
AT END OF PAGE Statement	80
Column Headers	81
Default Column Headers	81
Suppress Default Column Headers - The NOHDR Option	82
Define Your Own Column Headers	82
Combining NOTITLE and NOHDR	83
Centering of Column Headers - The HC Parameter	83
Width of Column Headers - The HW Parameter	83
Filler Characters for Headers - The Parameters FC and GC	83
Underlining Character for Titles and Headers - The UC Parameter	85
Suppressing Column Headers - The Notation '/'	86

Parameters to Influence the Output of Fields	87
Leading Characters - The LC Parameter	87
Insertion Characters - The IC Parameter	87
Trailing Characters - The TC Parameter	88
Output Length - The AL and NL Parameters	88
Sign Position - The SG Parameter	89
Identical Suppress - The IS Parameter	91
Zero Printing - The ZP Parameter	92
Empty Line Suppression - The ES Parameter	92
Edit Masks - The EM Parameter	94
Edit Masks for Numeric Fields	94
Edit Masks for Alphanumeric Fields	94
Length of Fields	95
Edit Masks for Date and Time Fields	95
Examples of Edit Masks	95
Vertical Displays	98
Combining DISPLAY and WRITE	98
Tab Notation T*field	100
Positioning Notation x/y	101
DISPLAY VERT Statement	102
DISPLAY VERT without AS Clause	102
DISPLAY VERT AS CAPTIONED and HORIZ	103
DISPLAY VERT AS text	104
DISPLAY VERT AS text CAPTIONED	105
Tab Notation P*field	106
Object Types	107
Object Types	107
What Types of Programming Objects Are There?	107
Data Areas	107
Local Data Area	108
Global Data Area	109
Parameter Data Area	110
Programs, Subprograms and Subroutines	113
A Modular Application Structure	113
Multiple Levels of Invoked Objects	114
Program	115
Subroutine	118
Subprogram	121
Processing Flow when Invoking a Routine	123
Maps	124
Helproutines	124
Invoking Help	125
Specifying Helproutines	125
Programming Considerations for Helproutines	126
Passing Parameters to Helproutines	126
Equal Sign Option	127
Array Indices	127
Help as a Window	128
Multiple Use of Source Code - Copycode	129
Documenting Natural Objects - Text	129
Creating Event Driven Applications - Dialog	130
Creating Component Based Applications - Class	130
Using Non-Natural Files - Resource	130
Shared Resources	131
Private Resources	131

Further Programming Aspects	132
Further Programming Aspects	132
End of Program - The END Statement	132
End of Application - The STOP Statement	132
Conditional Processing - The IF Statement	132
Nested IF Statements	134
Loop Processing	136
Limiting Database Loops	136
Limiting Non-Database Loops - The REPEAT Statement	137
Terminating a Processing Loop - The ESCAPE Statement	139
Loops Within Loops	139
Referencing Statements within a Program	141
Control Breaks	144
AT BREAK Statement	144
Automatic Break Processing	148
BEFORE BREAK PROCESSING Statement	151
User-Initiated Break Processing - The PERFORM BREAK PROCESSING Statement	152
Data Computation	156
Format of Fields	156
COMPUTE Statement	156
Statements MOVE and COMPUTE	157
Statements ADD, SUBTRACT, MULTIPLY and DIVIDE	157
COMPRESS Statement	158
Mathematical Functions	160
System Variables and System Functions	161
System Variables	161
System Functions	163
Stack	166
Stack Processing	166
Placing Data in the Stack	167
Clearing the Stack	167
Processing of Date Information	168
Edit Masks for Date Fields and Date System Variables	168
Default Edit Mask for Date - The DTFORM Parameter	168
Date Format for Alphanumeric Representation - The DF Parameter	169
Date Format for Output - The DFOUT Parameter	172
Date Format for Stack - The DFSTACK Parameter	173
Year Sliding Window - The YSLW Parameter	174
Combinations of DFSTACK and YSLW	177
Date Format for Default Page Title - The DFTITLE Parameter	179
Reporting Mode and Structured Mode	180
Reporting Mode and Structured Mode	180
General Information	180
Setting the Programming Mode	180
Functional Differences	181
Closing a Processing Loop in Reporting Mode	182
Closing a Processing Loop in Structured Mode	183
Database Reference	184
Portable Natural Generated Programs	186
Portable Natural Generated Programs	186
Compatibility	186
Endian Mode Considerations	186
ENDIAN Parameter	186
Transferring Natural Generated Programs	187

Programming Guide - Overview

This documentation applies to all platforms on which Natural can be used. It provides basic information on various aspects of programming with Natural. You should be familiar with this information before you start to write Natural applications.

- Defining Fields Describes how you define the fields you wish to use in a program.
- Database Access Describes various aspects of using Natural to access data in a database.
- Output of Data Discusses various aspects of how you can control the format of an output report created with Natural, that is, the way in which the data are displayed.
- Object Types Within an application, you can use several types of programming objects to achieve an efficient application structure. This section discusses the various types of Natural programming objects: data areas, programs, subprograms, subroutines, help routines, maps, etc.
- Further Programming Aspects Discusses various other aspects of programming with Natural.
- Reporting Mode and Structured Mode Describes the differences between the two Natural programming modes.
- Portable Natural Generated Programs As of Natural 5, generated programs are portable across the platforms UNIX, OpenVMS and Windows.

Example Programs

This documentation contains several examples of Natural programs, as well as references to further example programs not shown in the documentation.

All these programs are also provided in source-code form in the Natural library "SYSEXP". (The programs are all written in structured mode.)

Further example programs of using Natural statements are provided in the Natural library "SYSEXR".

Please ask your Natural administrator about the availability of these libraries at your site.

The example programs use data from the files "EMPLOYEES" and "VEHICLES", which are supplied by Software AG for demonstration purposes.

Programming Modes

Natural offers two ways of programming: reporting mode and structured mode. Generally, it is recommended to use structured mode exclusively, because it provides for more clearly structured applications. Therefore all explanations and examples in this documentation refer to structured mode. Any peculiarities of reporting mode will not be taken into consideration. (For differences between the two modes, please refer to Reporting Mode and Structured Mode.)

Defining Fields

This section describes how you define the fields (database fields and user-defined) you wish to use in a program. These fields can be database fields and user-defined fields. It contains information that applies to all fields in general and to user-defined fields in particular. The particulars of database fields are described in Database Access.

- DEFINE DATA Statement
- Structure of a DEFINE DATA Statement - Level Numbers
- User-Defined Variables
- User-Defined Constants
- Initial Values (and the RESET Statement)
- Redefining Fields
- Arrays
- Data Blocks

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

DEFINE DATA Statement

The first statement in a Natural program must always be a DEFINE DATA statement. In this statement, you define all the fields - database fields as well as user-defined variables - that are to be used in the program.

All fields to be used *must be* defined in the DEFINE DATA statement.

There are two ways to define the fields:

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

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

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

Data areas are created and maintained with the data area editor, which is described in your Natural User's Guide.

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

Example 1 - Fields Defined within a DEFINE DATA Statement:

```

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

```

Example 2 - Fields Defined in a Separate Data Area:

Program:

```

DEFINE DATA LOCAL
  USING LDA39
END-DEFINE
...

```

Local Data Area "LDA39":

I	T	L	Name	F	Leng	Index/Init/EM/Name/Comment
V	1		VIEWEMP			EMPLOYEES
	2		NAME	A	20	
	2		FIRST-NAME	A	20	
	2		PERSONNEL-ID	A	8	
	1		#VARI-A	A	20	
	1		#VARI-B	N	3.2	
	1		#VARI-C	I	4	

Structure of a DEFINE DATA Statement - Level Numbers

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

- view definitions
- field groups
- redefinitions

Level numbers are 1- or 2-digit numbers in the range from 01 to 99 (the leading "0" is optional).

Generally, variable definitions are on level 1.

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

Level Numbers in View Definitions

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

Example of Level Numbers in View Definition:

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

Level Numbers in Field Groups

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

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

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

Example of Level Numbers in Group:

```

DEFINE DATA LOCAL
  1 #FIELDA (N2.2)
  1 #FIELDB (I4)
  1 #GROUPA
    2 #FIELD C (A20)
    2 #FIELD D (A10)
    2 #FIELD E (N3.2)
  1 #FIELD F (A2)
  . . .
END-DEFINE

```

In this example, the fields #FIELD C, #FIELD D and #FIELD E are defined under the common group name #GROUP A. The other three fields are not part of the group. Note that #GROUP A only serves as a group name and is not a field in its own right (and therefore does not have a format/length definition).

Level Numbers in Redefinitions

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

Example of Level Numbers in Redefinition:

```

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

```

In this example, the database field BIRTH is redefined as three user-defined variables, and the user-defined variable #FIELD A is redefined as three other user-defined variables.

User-Defined Variables

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

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

Example:

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

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

The following topics are covered below:

- Names of User-Defined Variables
- Format and Length of User-Defined Variables

Names of User-Defined Variables

The name of a user-defined variable can be 1 to 32 characters long.

Note:

You may use variable names of over 32 characters (for example, in complex applications where longer meaningful variable names enhance the readability of programs); however, only the first 32 characters are significant and must therefore be unique, the remaining characters will be ignored by Natural.

The name of a user-defined variable must not be a Natural reserved word.

Within one Natural program, you should not use the same name for a user-defined variable and a database field.

The name of a user-defined variable can consist of the following characters:

Character	Explanation
A to Z	alphabetical characters
0 to 9	numeric characters
-	hyphen
@	at sign
_	underline
/	slash
\$	dollar sign
§	paragraph sign
&	ampersand
#	hash / number sign
+	plus sign (only allowed as first character)

The first character of the name must be one of the following:

- an upper-case alphabetical character
- #
- +
- &

Note:

In this section, the names of all user-defined variables begin with a hash sign (#); this avoids any naming conflicts with database fields or Natural reserved words.

If the first character is a "#", "+", or "&", the name must consist of at least one additional character.

"+" as the first character of a name is only allowed for application-independent variables (AIVs) and variables in a global data area. Names of AIVs **must** begin with a "+".

"&" as the first character of a name is used in conjunction with dynamic source program modification (see the RUN statement in the Natural Statements documentation) and when defining processing rules (see the map editor description in your Natural User's Guide).

Format and Length of User-Defined Variables

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

A user-defined variable can have one of the following formats:

A	Alphanumeric
B	Binary
C	Attribute Control
D	Date
F	Floating Point
I	Integer
L	Logical
N	Numeric unpacked
P	Packed numeric
T	Time

Information on possible lengths of user-defined variables is provided in the Natural Reference documentation,

Examples of User-Defined Variables:

```

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

```

Note:

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

User-Defined Constants

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

- Numeric Constants
- Alphanumeric Constants
- Date and Time Constants
- Hexadecimal Constants
- Logical Constants
- Floating Point Constants
- Attribute Constants
- Defining Named Constants

Numeric Constants

A numeric constant may contain 1 to 29 numeric digits.

A numeric constant used with a COMPUTE, MOVE, or arithmetic statement may contain a decimal point and sign notation.

Examples:

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

Alphanumeric Constants

An alphanumeric constant may contain 1 to 253 alphanumeric characters.

An alphanumeric constant must be enclosed in either apostrophes (') or quotation marks (").

Examples:

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

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

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

Example:

If you want the following to be output:

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

you can use any of the following notations:

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

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

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

Examples:

```
MOVE 'XXXXXX' -  
      'YYYYYY' TO #FIELD
```

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

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

Date and Time Constants

A date constant may be used in conjunction with a format D variable. Date constants may have the following formats:

D' <i>yyyy-mm-dd</i> '	International date format
D' <i>dd.mm.yyyy</i> '	German date format
D' <i>dd/mm/yyyy</i> '	European date format
D' <i>mm/dd/yyyy</i> '	USA date format

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

Example:

```
DEFINE DATA LOCAL
  1 #DATE (D)
  END-DEFINE
  ...
  MOVE D'1997-08-11' TO #DATE
  ...
```

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

A time constant may be used in conjunction with a format T variable. A time constant has the following format:

T'*hh:ii:ss***'**

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

Example:

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

Hexadecimal Constants

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

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

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

ASCII Examples:

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

EBCDIC Examples:

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

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

ASCII Example:

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

EBCDIC Example:

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

Logical Constants

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

Example:

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

Floating Point Constants

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

Example:

```

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

```

Attribute Constants

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

The following attributes may be used:

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

Example:

```

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

```

Defining Named Constants

If you need to use the same constant value several times in a program, you can reduce the maintenance effort by defining a named constant: you define a field in the DEFINE DATA statement, assign a constant value to it, and use the field name in the program instead of the constant value. Thus, when the value has to be changed, you only have to change it once in the DEFINE DATA statement and not everywhere in the program where it occurs.

You specify the constant value in angle brackets with the keyword "CONSTANT" after the field definition in the DEFINE DATA statement. If the value is alphanumeric, it must be enclosed in apostrophes.

Example:

```
DEFINE DATA LOCAL
  1 #FIELD A (N3) CONSTANT <100>
  1 #FIELD B (A5) CONSTANT <'ABCDE'>
END-DEFINE
...
```

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

Initial Values

You can assign an initial value to a user-defined variable. You specify the initial value in angle brackets with the keyword "INIT" after the variable definition in the DEFINE DATA statement. If the initial value is alphanumeric, it must be enclosed in apostrophes.

Example:

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

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

Example of System Variable as Initial Value:

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

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

With the option **FULL LENGTH**<character(s)> the entire field is filled with the specified *character(s)*.

With the option **LENGTHn** <character(s)> the first n positions of the field are filled with the specified *character(s)*.

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
...
```

Default Initial Values

If you specify no initial value for a field, the field will be initialised with a default initial value (null 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)

The RESET Statement

The RESET statement is used to set the value of a field to a null value, or to a specific initial value.

- RESET (without INITIAL) sets the value of each specified field to a null value.
- RESET INITIAL sets each specified field to the initial value as defined for the field in the DEFINE DATA statement.

Example:

```

DEFINE DATA LOCAL
  1 #FIELD A (N3) INIT <100>
  1 #FIELD B (A20) INIT <'ABC'>
  1 #FIELD C (I4) INIT <5>
END-DEFINE
...
...
RESET #FIELD A /* resets field value to null
...
RESET INITIAL #FIELD A #FIELD B #FIELD C /* resets field values to initial values
...

```

Redefining Fields

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

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

Important:

Dynamic variables are not allowed.

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

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

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

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

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

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

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

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

```

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

```

The following program illustrates the use of a redefinition:

```

** Example Program 'DDATAX01'
DEFINE DATA LOCAL
01 VIEWEMP VIEW OF EMPLOYEES
  02 NAME
  02 FIRST-NAME
  02 SALARY (1:1)
01 #PAY (N9)
01 REDEFINE #PAY
  02 FILLER 3X
  02 #USD (N3)
  02 #000 (N3)
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  MOVE SALARY (1) TO #PAY
  DISPLAY NAME FIRST-NAME #PAY #USD #000
END-READ
END

```

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

Page	1					99-08-08	17:48:59
	NAME	FIRST-NAME	#PAY	#USD	#000		
	-----	-----	-----	-----	-----		
	JONES	VIRGINIA	46000	46	0		
	JONES	MARSHA	50000	50	0		
	JONES	ROBERT	31000	31	0		

Arrays

Natural supports the processing of *arrays*. Arrays are multi-dimensional tables, that is, two or more logically related elements identified under a single name. Arrays can consist of single data elements of multiple dimensions or hierarchical data structures which contain repetitive structures or individual elements. In Natural, an array can be one-, two- or three-dimensional. It can be an independent variable, part of a larger data structure or part of a database view.

The following topics are covered below:

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

Defining Arrays

To define an array variable, after the format and length you specify a slash followed by a so-called *index notation*, that is, the number of occurrences of the array.

Important:

Dynamic variables are not allowed.

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

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

To define a two-dimensional array, you specify an index notation for both dimensions:

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

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

Initial Values for Arrays

To assign initial values to one or more occurrences of an array, you use an INIT specification, similar to that for "ordinary" variables.

Assigning Initial Values to One-Dimensional Arrays

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

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

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

"A" is assigned to the second occurrence.

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

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

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

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

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

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

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

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

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

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

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

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

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

Alternatively, you could specify:

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

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

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

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

Assigning Initial Values to Two-Dimensional Arrays

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

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

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

Vertical: First Dimension (1:3), Horizontal: Second Dimension (1:4):

(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)

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

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

- Assigning the Same Value
- Assigning Different Values

Assigning the Same Value

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

		A	

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

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

		A	
		A	
		A	

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

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

A	A	A	A
A	A	A	A

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

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

A	A		
A	A		

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

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

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

Alternatively, you could specify:

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

Assigning Different Values

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

	A		
	B		
	C		

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

	A	A	
	B	B	
	C	C	

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

A	A	A	A
B	B	B	B
C	C	C	C

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

A	A	A	A
C	C	C	C

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

A	A	A	A
B	B	B	B

```
1 #ARRAY (A1/1:3,1:4) INIT (V,1) <'A','B','C'>
(V,3) <'D','E','F'>
```

A		D	
B		E	
C		F	

```
1 #ARRAY (A1/1:3,1:4) INIT (3,V) <'A','B','C','D'>
```

A	B	C	D

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

A	B	C	D
A	B	C	D
A	B	C	D

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

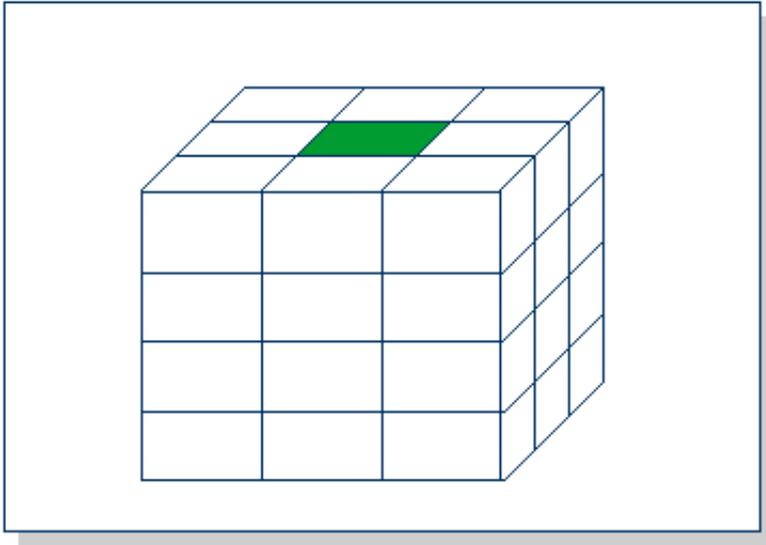
	B		
A	B		
	B	C	D

```
1 #ARRAY (A1/1:3,1:4) INIT (2,1) <'A'> (V,2) <'B','C','D'>
(3,3) <'E'> (3,4) <'F'>
```

	B		
A	C		
	D	E	F

A Three-Dimensional Array

A three-dimensional array could be visualized as follows:



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

```

DEFINE DATA LOCAL
  1 #ARRAY2
  2 #ROW (1:4)
  3 #COLUMN (1:3)
  4 #PLANE (1:3)
  5 #FIELD2 (P3) INIT (1,2,2) <100>
END-DEFINE
...

```

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

I	T	L	Name	F	Leng	Index/Init/EM/Name/Comment
			1 #ARRAY2			
			2 #ROW			(1:4)
			3 #COLUMN			(1:3)
			4 #PLANE			(1:3)
I			5 #FIELD2	P	3	

Arrays as Part of a Larger Data Structure

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

Example:

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

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

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

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

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

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

Database Arrays

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

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

```

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

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

I	T	L	Name	F	Leng	Index/Init/EM/Name/Comment
V	1		EMPLOYEES-VIEW			EMPLOYEES
	2		NAME	A	20	
M	2		ADDRESS-LINE	A	20	(1:10) /* MU-FIELD

Using Arithmetic Expressions in Index Notation

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

Examples:

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

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

Only the arithmetic operators "+" and "-" may be used in index expressions.

Arithmetic Support for Arrays

Arithmetic support for arrays include operations at array level, at row/column level, and at individual element level. Only simple arithmetic expressions are permitted with array variables, with only one or two operands and an optional third variable as the receiving field. Only the arithmetic operators "+" and "-" are allowed for expressions defining index ranges.

Examples of Array Arithmetics:

The following examples assume the following field definitions:

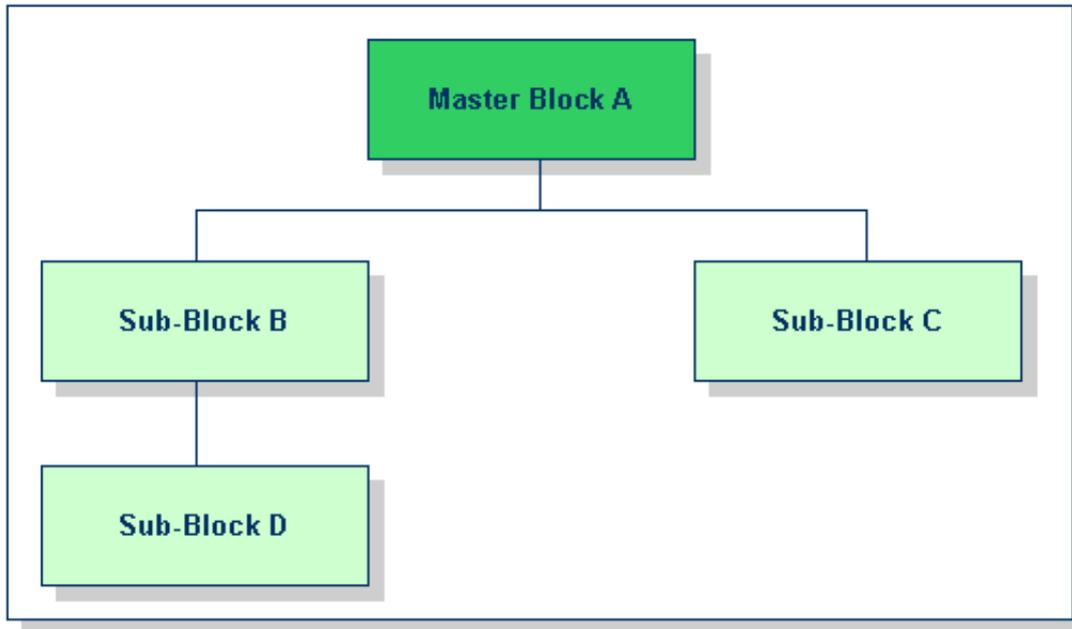
```
DEFINE DATA LOCAL
  01 #A (N5/1:10,1:10)
  01 #B (N5/1:10,1:10)
  01 #C (N5)
END-DEFINE
. . .
```

1. **ADD #A(*,*) TO #B(*,*)**
The result operand, array #B, contains the addition, element by element, of the array #A and the original value of array #B.
2. **ADD 4 TO #A(*,2)**
The second column of the array #A is replaced by its original value plus 4.
3. **ADD 2 TO #A(2,*)**
The second row of the array #A is replaced by its original value plus 2.
4. **ADD #A(2,*) TO #B(4,*)**
The value of the second row of array #A is added to the fourth row of array #B.
5. **ADD #A(2,*) TO #B(*,2)**
This is an illegal operation and will result in a syntax error. Rows may only be added to rows and columns to columns.
6. **ADD #A(2,*) TO #C**
All values in the second row of the array #A are added to the scalar value #C.
7. **ADD #A(2,5:7) TO #C**
The fifth, sixth, and seventh column values of the second row of array #A are added to the scalar value #C.

Data Blocks

To save data storage space, you can create a global data area with data blocks. Data blocks can overlay each other during program execution, thereby saving storage space.

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



The following topics are covered below:

- Defining Data Blocks
- Block Hierarchies

Defining Data Blocks

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

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

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

Example:

Global Data Area G-BLOCK:

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

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

Program 1:

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

Program 2:

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

Program 3:

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

Program 4:

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

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

Block Hierarchies

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

Program 1:

```

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

```

Program 2:

```

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

```

Program 3:

```

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

```

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

Database Access

This section describes various aspects of accessing data in a database with Natural. It covers the following topics:

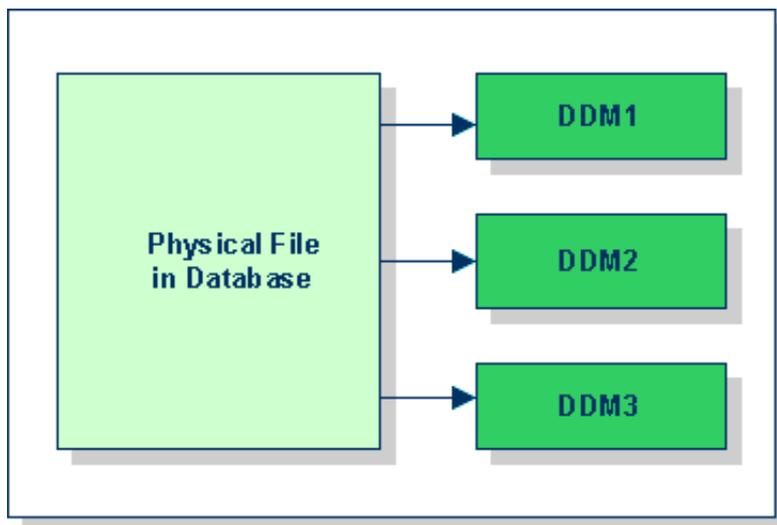
- DDMs (Data Definition Modules)
 - Database Arrays
 - DEFINE DATA Views
 - Statements for Database Access
 - READ Statement
 - FIND Statement
 - HISTOGRAM Statement
 - Database Processing Loops
 - Database Update - Transaction Processing
 - Statements ACCEPT and REJECT
 - AT START/END OF DATA Statements
-

DDMs (Data Definition Modules)

For Natural to be able to access a database file, a logical definition of the physical database file is required. Such a logical file definition is called a DDM (data definition module).

The DDM contains information about the individual fields of the file - information which is relevant for the use of these fields in a Natural program. A DDM constitutes a logical view of a physical database file.

For each physical file of a database, one or more DDMs can be defined.



DDMs are defined by the Natural administrator with Predict (or, if Predict is not available, with the corresponding Natural function, as described in your Natural User's Guide for Mainframes documentation).

For each database field, a DDM contains the database-internal field name as well as the "external" field name, that is, the name of the field as used in a Natural program. Moreover, the formats and lengths of the fields are defined in the DDM, as well as various specifications that are used when the fields are output with a DISPLAY or WRITE statement (column headings, edit masks, etc.).

The following topics are covered below:

- Displaying a DDM
- Components of a DDM

Displaying a DDM

If you do not know the name of the DDM you want, you can use the system command **LIST DDM** to get a list of all existing DDMs that are available. From the list, you can then select a DDM for display.

To display a DDM whose name you know, you use the system command **LIST DDM *dsm-name***.

For example:

LIST DDM EMPLOYEES

A list of all fields defined in the DDM will then be displayed, along with information about each field:

Components of a DDM

For each field, a DDM contains the following information:

Column	Explanation
T	<p>The <i>type</i> of the field:</p> <p><i>blank</i> Elementary field. This type of field can have only one value within a record.</p> <p>M Multiple-value field. This type of field can have more than one value within a record.</p> <p>P Periodic group. A periodic group is a group of fields that can have more than one occurrence within a record.</p> <p>G Group. A group is a number of fields defined under one common group name. This makes it possible to reference several fields collectively by using the group name instead of the names of all the individual fields.</p> <p>* Comment line.</p>
L	<p>The <i>level</i> number assigned to the field.</p> <p>Levels are used to indicate the structure and grouping of the field definitions. This is relevant with view definitions, redefinitions and field groups.</p>
DB	The two-character database- <i>internal field name</i> .
Name	<p>The 3- to 32-character <i>external field name</i>. This is the field name used in a Natural program to reference the field.</p> <p>HD= indicates a default column header to appear above the field when the field is output via a DISPLAY statement. If no header is specified, the field name is used as column header.</p> <p>EM= indicates a default edit mask to be used when the field is output via a DISPLAY statement.</p>
F	The <i>format</i> of the field (A=alphanumeric, N=numeric unpacked, P=packed numeric, etc.).
Len	<p>The <i>length</i> of the field.</p> <p>For numeric fields, length is specified as "<i>nn.m</i>", where "<i>nn</i>" is the number of digits before the decimal point and "<i>m</i>" is the number of digits after the decimal point.</p>

Column	Explanation
S	<p>The type of <i>suppression</i> assigned to the field:</p> <p>N indicates <i>null-value suppression</i>, which means that null values for the field will not be returned when the field is used to construct a basic search criterion (WITH clause of a FIND statement), in a HISTOGRAM statement, or in a READ LOGICAL statement.</p> <p>F indicates that the field is defined with the <i>fixed storage</i> option (that is, the field is not compressed).</p> <p>A blank indicates <i>normal compression</i>, which means that trailing blanks in alphanumeric fields and leading zeros in numeric fields are suppressed.</p>
D	<p>The <i>descriptor</i> type of the field; for example:</p> <p>D elementary descriptor, N non-descriptor, P phonetic descriptor. U subdescriptor, S superdescriptor,</p> <p>A blank in this column indicates that the field is not a descriptor.</p> <p>A descriptor can be used as the basis of a database search. A field which has a "D" or "S" in this column can be used in the BY clause of the READ statement. Once a record has been read from the database using the READ statement, a DISPLAY statement can reference any field which has either a "D" or a blank in the "D" column.</p>
Remarks	This column can contain <i>comments</i> about the field.

Above the list of fields, the following is displayed: the number of the file from the DDM is derived (DDM FNR), the number of the database where that file is stored (DDM DBID), and the "Default Sequence" field, that is, the name of the field used to control logical sequential reading of the file if no such field is specified in the READ LOGICAL statement of a program.

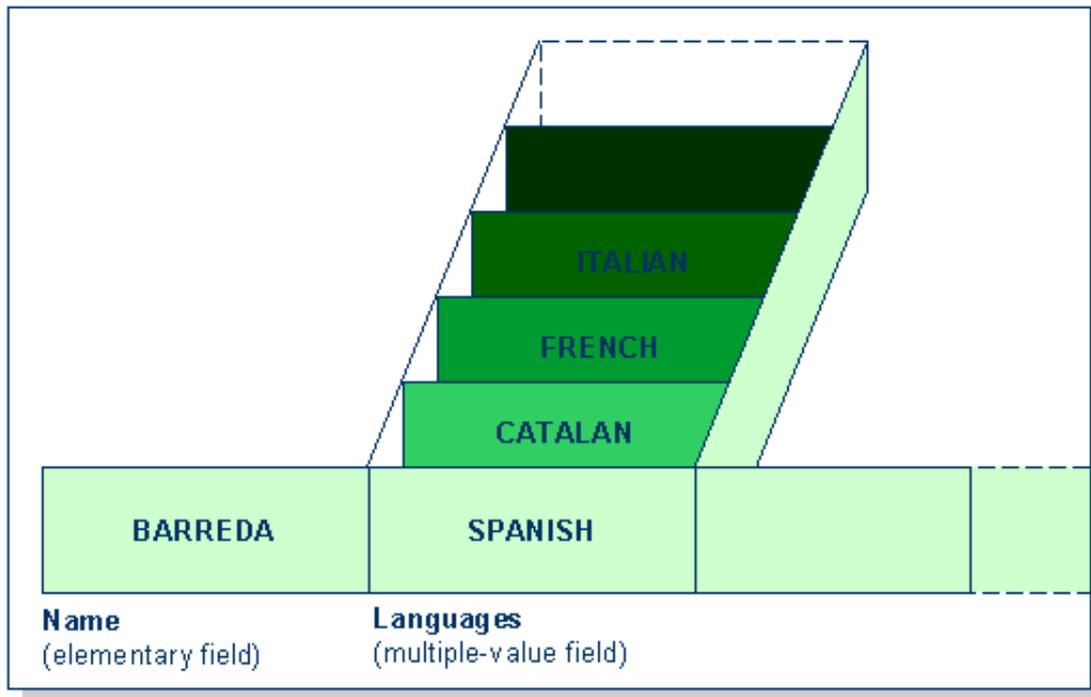
Database Arrays

Adabas supports array structures within the database in the form of *multiple-value fields* and *periodic groups*.

- Multiple-Value Fields
- Periodic Groups
- Referencing Multiple-Value Fields and Periodic Groups
- Multiple-Value Fields Within Periodic Groups
- Referencing Multiple-Value Fields Within Periodic Groups
- Referencing the Internal Count of a Database Array

Multiple-Value Fields

A multiple-value field is a field which can have more than one value (up to 191) within a given record.

Example:

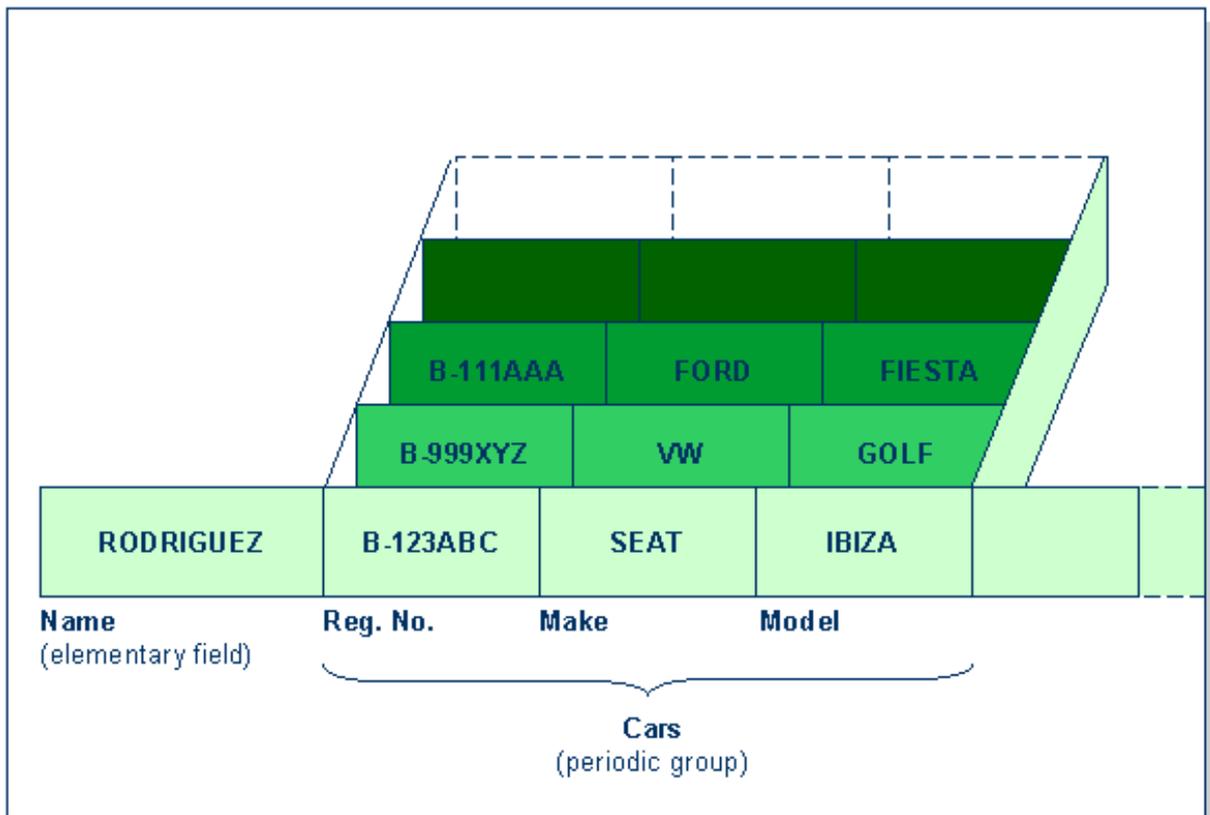
Assuming that the above is a record in an employees file, the first field (Name) is an elementary field, which can contain only one value, namely the name of the person; whereas the second field (Languages), which contains the languages spoken by the person, is a multiple-value field, as a person can speak more than one language.

Periodic Groups

A periodic group is a group of fields (which may be elementary fields and/or multiple-value fields) that may have more than one occurrence (up to 191) within a given record.

The different values of an multiple-value field are usually called *occurrences*; that is, the number of occurrences is the number of values which the field contains, and a specific occurrence means a specific value. Similarly, in the case of periodic groups, occurrences refer to a group of values.

Example:



Assuming that the above is a record in a vehicles file, the first field (Name) is an elementary field which contains the name of a person; Cars is a periodic group which contains the automobiles owned by that person. The periodic group consists of three fields which contain the registration number, make and model of each automobile. Each occurrence of Cars contains the values for one automobile.

Referencing Multiple-Value Fields and Periodic Groups

To reference one or more occurrences of a multiple-value field or a periodic group, you specify an *index notation* after the field name.

Examples:

The following examples use the multiple-value field LANGUAGES and the periodic group CARS from the previous examples.

The various values of the multiple-value field LANGUAGES can be referenced as follows.

LANGUAGES (1)	References the first value ("SPANISH").
LANGUAGES (X)	The value of the variable X determines the value to be referenced.
LANGUAGES (1:3)	References the first three values ("SPANISH", "CATALAN" and "FRENCH").
LANGUAGES (6:10)	References the sixth to tenth values.
LANGUAGES (X:Y)	The values of the variables X and Y determine the values to be referenced.

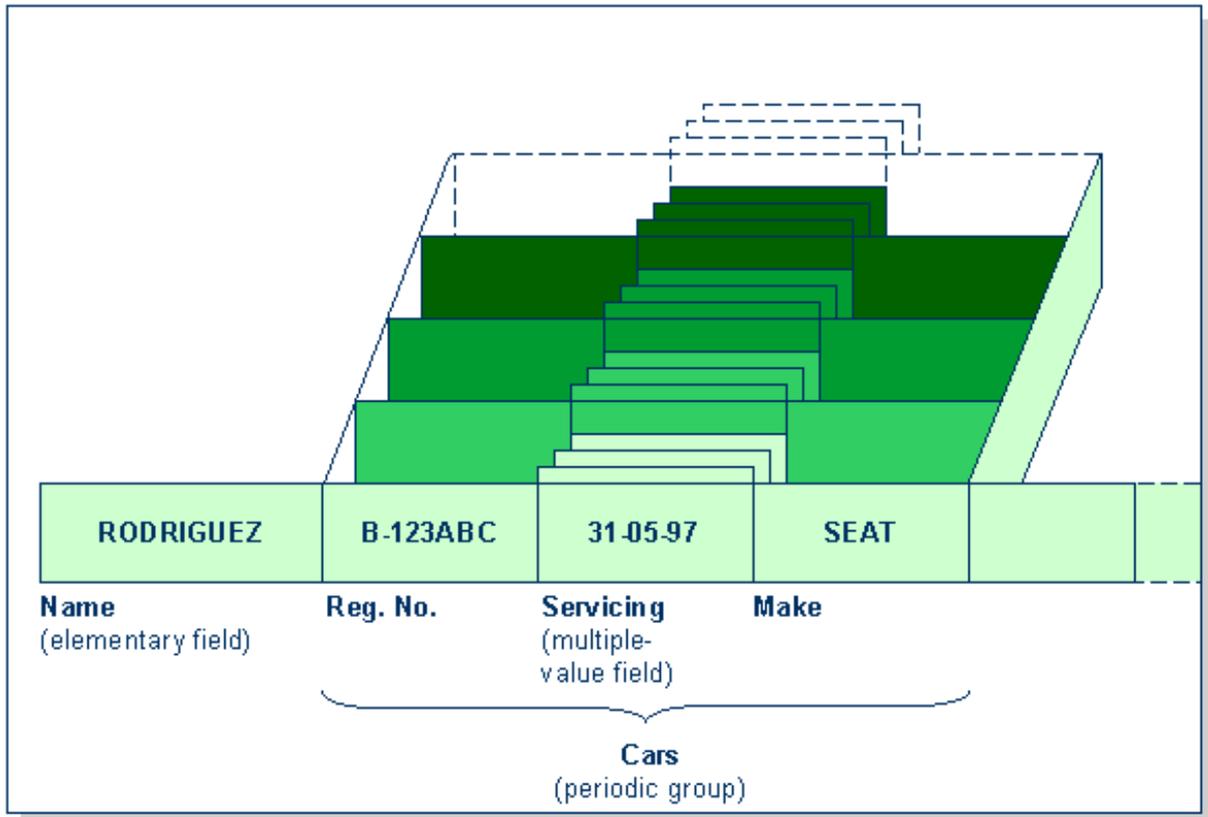
The various occurrences of the periodic group CARS can be referenced in the same manner:

CARS (1)	References the first occurrence ("B-123ABC/SEAT/IBIZA").
CARS (X)	The value of the variable X determines the occurrence to be referenced.
CARS (1:2)	References the first two occurrences ("B-123ABC/ SEAT/IBIZA" and "B-999XYZ/VW/GOLF").
CARS (4:7)	References the fourth to seventh occurrences.
CARS (X:Y)	The values of the variables X and Y determine the occurrences to be referenced.

Multiple-Value Fields Within Periodic Groups

An Adabas array can have up to two dimensions: a multiple-value field within a periodic group.

Example:



Assuming that the above is a record in a vehicles file, the first field (Name) is an elementary field which contains the name of a person; Cars is a periodic group which contains the automobiles owned by that person. The periodic group consists of three fields which contain the registration number, servicing dates and make of each automobile. Within the periodic group Cars, the field Servicing is a multiple-value field, containing the different servicing dates for each automobile.

Referencing Multiple-Value Fields Within Periodic Groups

To reference one or more occurrences of a multiple-value field within a periodic group, you specify a "two-dimensional" index notation after the field name.

Examples:

The following examples use the multiple-value field **SERVICING** within the periodic group **CARS** from the example above. The various values of the multiple-value field can be referenced as follows:

SERVICING (1,1)	References the first value of SERVICING in the first occurrence of CARS ("31-05-97")
SERVICING (1:5,1)	References the first value of SERVICING in the first five occurrences of CARS .
SERVICING (1:5,1:10)	References the first ten values of SERVICING in the first five occurrences of CARS .

Referencing the Internal Count of a Database Array

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

The count is returned in format/length N3. See the Natural Reference documentation for further details.

Examples:

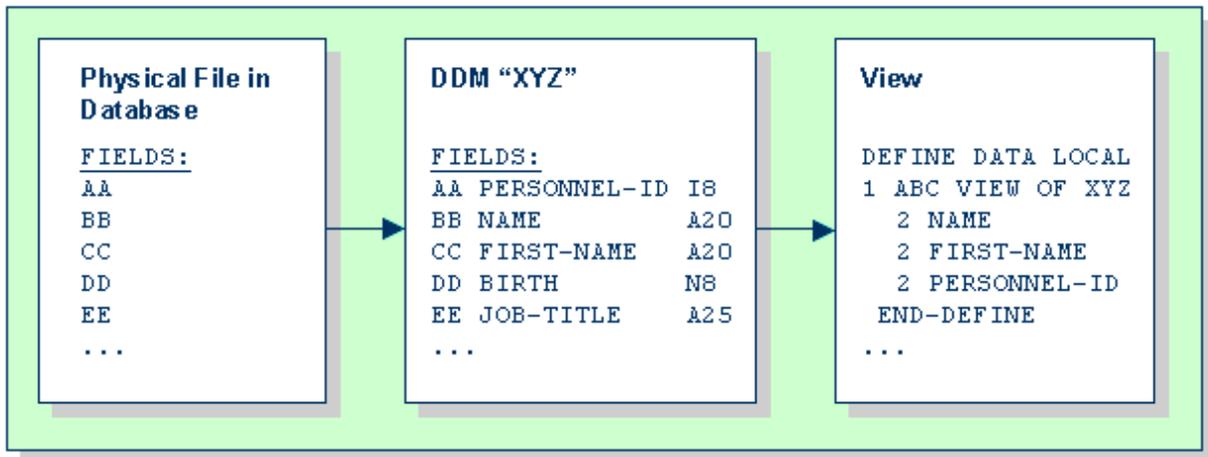
C*LANGUAGES	Returns the number of values of the multiple-value field LANGUAGES .
C*CARS	Returns the number of occurrences of the periodic group CARS .
C*SERVICING(1)	Returns the number of values of the multiple-value field SERVICING in the first occurrence of a periodic group (assuming that SERVICING is a multiple-value field within a periodic group.)

DEFINE DATA Views

To be able to use database fields in a Natural program, you must specify the fields in a *view*.

In the view, you specify the name of the DDM from which the fields are taken, and the names of the database fields themselves (that is, their long names, not their database-internal short names).

You define such a database view either within the DEFINE DATA statement of the program, or in a local or global data area outside the program with the DEFINE DATA statement referencing that data area (as described in the section Defining Fields).



At level 1, you specify the view name as follows:

1 *view-name* **VIEW OF** *ddm-name*

where *view-name* is the name you choose for the view, and *ddm-name* is the name of the DDM from which the fields specified in the view are taken. Below that, at level 2, you specify the names of the database fields from the DDM.

In the illustration above, the name of the view is "ABC", and it comprises the fields NAME, FIRST-NAME and PERSONNEL-ID from the DDM "XYZ".

The format and length of a database field need not be specified in the view, as these are already defined in the underlying DDM.

The view may comprise an entire DDM or only a subset of it. The order of the fields in the view need not be the same as in the underlying DDM.

As shown later in this section, the view name is used in database access statements to determine which database is to be accessed.

Statements for Database Access

To read data from a database, the following statements are available:

- **READ**
This statement is used to select a range of records from a database in a specified sequence.
- **FIND**
This statement is used to select from a database those records which meet a specified search criterion.
- **HISTOGRAM**
This statement is used to read only the values of one database field, or determine the number of records which meet a specified search criterion.

READ Statement

The READ statement is used to read records from a database. The records can be retrieved from the database:

- in the order in which they are physically stored in the database (READ IN PHYSICAL SEQUENCE), or
- in the order of Adabas Internal Sequence Numbers (READ BY ISN), or
- in the order of the values of a descriptor field (READ IN LOGICAL SEQUENCE).

In this documentation, only READ IN LOGICAL SEQUENCE is discussed, as it is the most frequently used form of the READ statement; for information on the other two options, please refer to the description of the READ statement in the Natural Statements documentation.

The following topics are covered below:

- Syntax
- Limiting the Number of Records to be Read
- The STARTING/ENDING Clauses
- The WHERE Clause

Syntax

The basic syntax of the READ statement is:

READ *view* **IN LOGICAL SEQUENCE BY** *descriptor*

or shorter:

READ *view* **LOGICAL BY** *descriptor*

view is the name of a view defined in the DEFINE DATA statement (as explained earlier in this section).

descriptor is the name of a database field defined in that view. The values of this field determine the order in which the records are read from the database.

If you specify a descriptor, you need not specify the keyword "LOGICAL":

READ *view* **BY** *descriptor*

If you do not specify a descriptor, the records will be read in the order of values of the field defined as default descriptor (under "Default Sequence") in the DDM. However, if you specify no descriptor, you must specify the keyword "LOGICAL":

READ *view* **LOGICAL**

Example:

```
** Example Program 'READX01'  
  DEFINE DATA LOCAL  
  1 MYVIEW VIEW OF EMPLOYEES  
    2 PERSONNEL-ID  
    2 NAME  
    2 JOB-TITLE  
  END-DEFINE  
  READ (6) MYVIEW BY NAME  
    DISPLAY NAME PERSONNEL-ID JOB-TITLE  
  END-READ  
  END
```

With the READ statement in the above example, records from the EMPLOYEES file are read in alphabetical order of their last names.

The above program will produce the following output, displaying the information of each employee in alphabetical order of the employees' last names:

Page	1		99-08-19 13:16:04
	NAME	PERSONNEL ID	CURRENT POSITION
	-----	-----	-----
	ABELLAN	60008339	MAQUINISTA
	ACHIESON	30000231	DATA BASE ADMINISTRATOR
	ADAM	50005800	CHEF DE SERVICE
	ADKINSON	20008800	PROGRAMMER
	ADKINSON	20009800	DBA
	ADKINSON	2001100	

If you wanted to read the records to create a report with the employees listed in sequential order by date of birth, the appropriate READ statement would be:

```
READ MYVIEW BY BIRTH
```

You can only specify a field which is defined as a "descriptor" in the underlying DDM (it can also be a subdescriptor, superdescriptor or hyperdescriptor).

Limiting the Number of Records to be Read

As shown in the previous example program, you can limit the number of records to be read by specifying a number in parentheses after the keyword READ:

```
READ (6) MYVIEW BY NAME
```

In that example, the READ statement would read no more than 6 records.

Without the limit notation, the above READ statement would read *all* records from the EMPLOYEES file in the order of last names from A to Z.

The STARTING/ENDING Clauses

The READ statement also allows you to qualify the selection of records based on the **value** of a descriptor field. With an EQUAL TO/STARTING from option in the BY or WITH clause, you can specify the value at which reading should begin. By adding a THRU/ENDING AT option, you can also specify the value in the logical sequence at which reading should end.

For example, if you wanted a list of those employees in the order of job titles starting with "TRAINEE" and continuing on to "Z", you would use one of the following statements:

```
READ MYVIEW WITH JOB-TITLE = 'TRAINEE'
  READ MYVIEW WITH JOB-TITLE STARTING from 'TRAINEE'
  READ MYVIEW BY JOB-TITLE = 'TRAINEE'
  READ MYVIEW BY JOB-TITLE STARTING from 'TRAINEE'
```

Note that the value to the right of the equal sign (=) or STARTING from option must be enclosed in apostrophes. If the value is numeric, this *text notation* is not required.

If a BY option is used, a WITH option cannot be used and vice versa.

The sequence of records to be read can be even more closely specified by adding an end limit with a THRU or ENDING AT clause.

To read just the records with the job title "TRAINEE", you would specify:

```
READ MYVIEW BY JOB-TITLE STARTING from 'TRAINEE' THRU 'TRAINEE'
  READ MYVIEW WITH JOB-TITLE EQUAL TO 'TRAINEE'
  ENDING AT 'TRAINEE'
```

To read just the records with job titles that begin with "A" or "B", you would specify:

```
READ MYVIEW BY JOB-TITLE = 'A' THRU 'C'
  READ MYVIEW WITH JOB-TITLE STARTING from 'A' ENDING AT 'C'
```

The values are read up to and including the value specified after THRU/ENDING AT. In the two examples above, all records with job titles that begin with "A" or "B" are read; if there were a job title "C", this would also be read, but not the next higher value "CA".

The WHERE Clause

The WHERE clause may be used to further qualify which records are to be read.

For instance, if you wanted only those employees with job titles starting from "TRAINEE" who are paid in US currency, you would specify:

```
READ MYVIEW WITH JOB-TITLE = 'TRAINEE'
  WHERE CURR-CODE = 'USD'
```

The WHERE clause can also be used with the BY clause as follows:

```
READ MYVIEW BY NAME
  WHERE SALARY = 20000
```

The WHERE clause differs from the WITH/BY clause in two respects:

- The field specified in the WHERE clause need not be a descriptor.
- The expression following the WHERE option is a logical condition. The following logical operators are possible in a WHERE clause:

EQUAL	EQ	=
NOT EQUAL TO	NE	≠
LESS THAN	LT	<
LESS THAN OR EQUAL TO	LE	<=
GREATER THAN	GT	>
GREATER THAN OR EQUAL TO	GE	>=

The following program illustrates the use of the STARTING from, ENDING AT and WHERE clauses:

```

** Example Program 'READX02'
DEFINE DATA LOCAL
1 MYEMP VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 INCOME (1:2)
  3 CURR-CODE
  3 SALARY
  3 BONUS (1:1)
END-DEFINE
*
READ (3) MYVIEW WITH JOB-TITLE = 'TRAINEE' THRU 'TRAINEE'
                WHERE CURR-CODE (*) = 'USD'
  DISPLAY NOTITLE NAME / JOB-TITLE 5X INCOME (1:2)
  SKIP 1
END-READ
END
    
```

It produces the following output:

NAME CURRENT POSITION	INCOME		
	CURRENCY CODE	ANNUAL SALARY	BONUS
-----	-----	-----	-----
SENKO	USD	23000	0
TRAINEE	USD	21800	0
BANGART	USD	25000	0
TRAINEE	USD	23000	0
LINCOLN	USD	24000	0
TRAINEE	USD	22000	0

Further Example of READ Statement:

See program READX03 in library SYSEXPG.

FIND Statement

The FIND statement is used to select from a database those records which meet a specified search criterion.

The following topics are covered below:

- Syntax
- Limiting the Number of Records to be Processed
- The WHERE Clause
- IF NO RECORDS FOUND Condition

Syntax

The basic syntax of the FIND statement is:

FIND RECORDS IN *view* **WITH** *field* = *value*

or shorter:

FIND *view* **WITH** *field* = *value*

view is the name of a view defined in the DEFINE DATA statement (as explained earlier in this section).

field is the name of a database field defined in that view. You can only specify a *field* which is defined as a "descriptor" in the underlying DDM (it can also be a subdescriptor, superdescriptor, hyperdescriptor or phonetic descriptor).

Limiting the Number of Records to be Processed

In the same way as with the READ statement, you can limit the number of records to be processed by specifying a number in parentheses after the keyword FIND:

```
FIND (6) RECORDS IN MYVIEW WITH NAME = 'CLEGG'
```

In the above example, only the first 6 records that meet the search criterion would be processed.

Without the limit notation, all records that meet the search criterion would be processed.

Note:

If the FIND statement contains a WHERE clause (see below), records which are rejected as a result of the WHERE clause are **not** counted against the limit.

The WHERE Clause

With the WHERE clause of the FIND statement, you can specify an additional selection criterion which is evaluated *after* a record (selected with the WITH clause) has been read and *before* any processing is performed on the record.

Example of WHERE Clause:

```
** Example Program 'FINDX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 CITY
END-DEFINE
*
FIND MYVIEW WITH CITY = 'PARIS'
```

```

WHERE JOB-TITLE = 'INGENIEUR COMMERCIAL'
DISPLAY NOTITLE CITY JOB-TITLE PERSONNEL-ID NAME
END-FIND
END
    
```

Note that in this example only those records which meet the criteria of the WITH clause *and* the WHERE clause are processed in the DISPLAY statement.

CITY	CURRENT POSITION	PERSONNEL ID	NAME
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 Condition

If no records are found that meet the search criteria specified in the WITH and WHERE clauses, the statements within the FIND processing loop are not executed (for the previous example, this would mean that the DISPLAY statement would not be executed and consequently no employee data would be displayed).

However, the FIND statement also provides an IF NO RECORDS FOUND clause, which allows you to specify processing you wish to be performed in the case that no records meet the search criteria.

Example of IF NO RECORDS FOUND Clause:

```

** Example Program 'FINDX02'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
2 NAME
2 FIRST-NAME
END-DEFINE
*
FIND MYVIEW WITH NAME = 'BLACKMORE'
IF NO RECORDS FOUND
WRITE 'NO PERSON FOUND.'
END-NOREC
DISPLAY NAME FIRST-NAME
END-FIND
END
    
```

The above program selects all records in which the field NAME contains the value "BLACKMORE". For each selected record, the name and first name are displayed. If no record with NAME = 'BLACKMORE' is found on the file, the WRITE statement within the IF NO RECORDS FOUND clause is executed:

Page	1	97-08-19	11:44:00
	NAME	FIRST-NAME	

NO PERSON FOUND.			

Further Examples of FIND Statement:

See programs FINDX07, FINDX08, FINDX09, FINDX10 and FINDX11 in library SYSEXP.

HISTOGRAM Statement

The HISTOGRAM statement is used to either read only the values of one database field, or determine the number of records which meet a specified search criterion.

The HISTOGRAM statement does not provide access to any database fields other than the one specified in the HISTOGRAM statement.

The following topics are covered below:

- Syntax
- Limiting the Number of Values to be Read
- The STARTING/ENDING Clauses
- The WHERE Clause

Syntax

The basic syntax of the HISTOGRAM statement is:

HISTOGRAM VALUE IN *view* **FOR** *field*

or shorter:

HISTOGRAM *view field*

view is the name of a view defined in the DEFINE DATA statement (as explained earlier in this section). *field* is the name of the database field defined in that view.

Limiting the Number of Values to be Read

In the same way as with the READ statement, you can limit the number of values to be read by specifying a number in parentheses after the keyword HISTOGRAM:

```
HISTOGRAM (6) MYVIEW FOR NAME
```

In the above example, only the first 6 values of the field NAME would be read.

Without the limit notation, all values would be read.

The STARTING/ENDING Clauses

Like the READ statement, the HISTOGRAM statement also provides a STARTING from clause and an ENDING AT (or THRU) clause to narrow down the range of values to be read by specifying a starting value and ending value.

Examples:

```
HISTOGRAM MYVIEW FOR NAME STARTING from 'BOUCHARD'
HISTOGRAM MYVIEW FOR NAME STARTING from 'BOUCHARD' ENDING AT 'LANIER'
HISTOGRAM MYVIEW FOR NAME from 'BLOOM' THRU 'ROESER'
```

The WHERE Clause

The HISTOGRAM statement also provides a WHERE clause which may be used to specify an additional selection criterion that is evaluated *after* a value has been read and *before* any processing is performed on the value. The field specified in the WHERE clause must be the same as in the main clause of the HISTOGRAM statement.

Example of HISTOGRAM Statement:

```

** Example Program 'HISTOX01'
  DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
  END-DEFINE
  *
  LIMIT 8
  HISTOGRAM MYVIEW CITY STARTING FROM 'M'
  DISPLAY NOTITLE CITY 'NUMBER OF/PERSONS' *NUMBER *COUNTER
  END-HISTOGRAM
  END

```

CITY	NUMBER OF PERSONS	CNT
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

In the above program, the system variables *NUMBER and *COUNTER are also evaluated by the HISTOGRAM statement, and output with the DISPLAY statement. *NUMBER contains the number of database records that contain the last value read; *COUNTER contains the total number of values which have been read.

Database Processing Loops

Natural automatically creates the necessary processing loops which are required to process data that have been selected from a database as a result of a FIND, READ or HISTOGRAM statement.

Example:

```

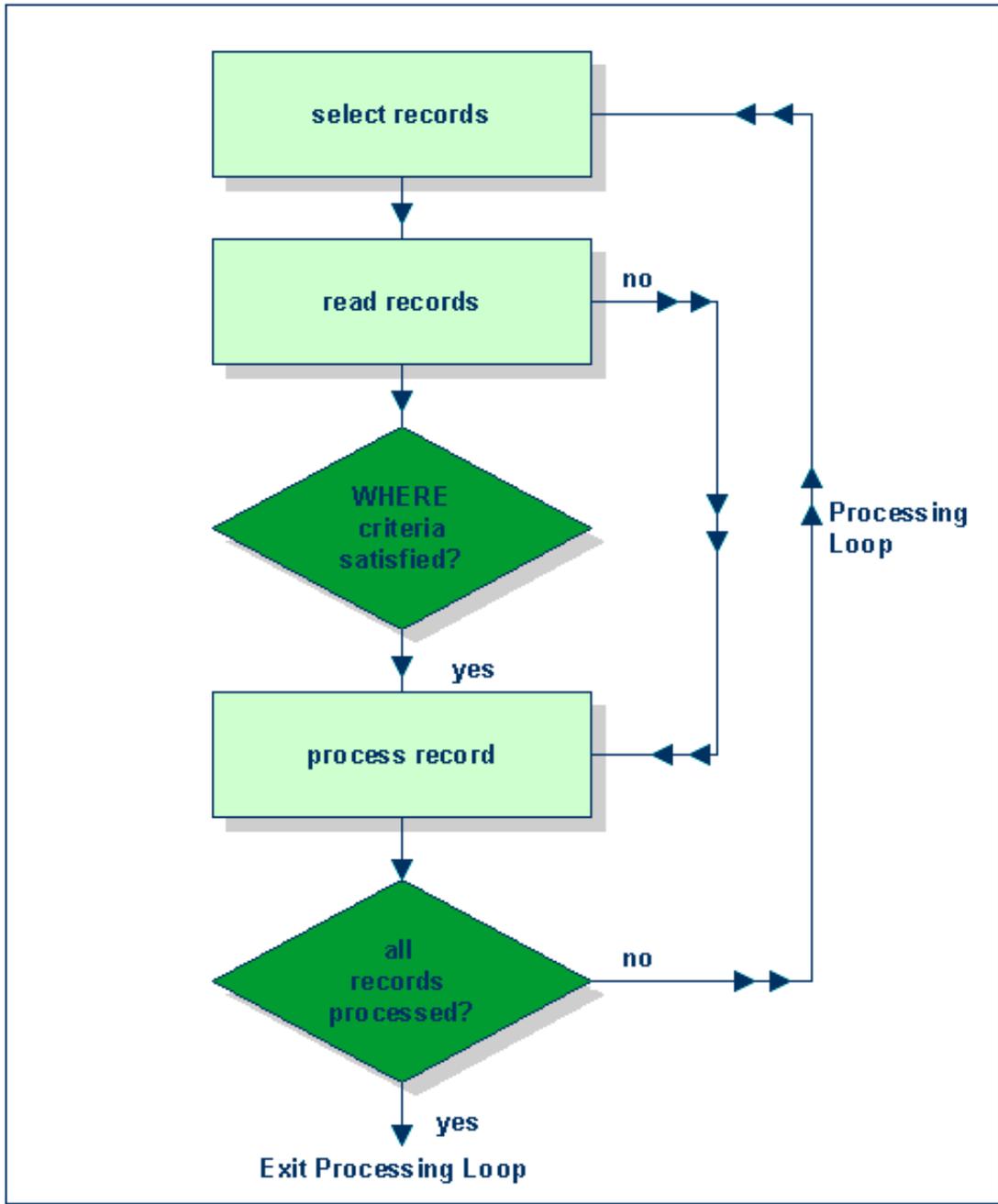
** Example Program 'FINDX03'
  DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  END-DEFINE
  *
  FIND MYVIEW WITH NAME = 'ADKINSON'
  DISPLAY NAME FIRST-NAME CITY
  END-FIND
  END

```

The above FIND loop selects all records from the EMPLOYEES file in which the field NAME contains the value "ADKINSON" and processes the selected records. In this example, the processing consists of displaying certain fields from each record selected.

If the FIND statement contained a WHERE clause in addition to the WITH clause, only those records that were selected as a result of the WITH clause *and* met the WHERE criteria would be processed.

The following diagram illustrates the flow logic of a database processing loop:



Hierarchies of Processing Loops

The use of multiple FIND and/or READ statements creates a hierarchy of processing loops, as shown in the following example:

Example of Processing Loop Hierarchy:

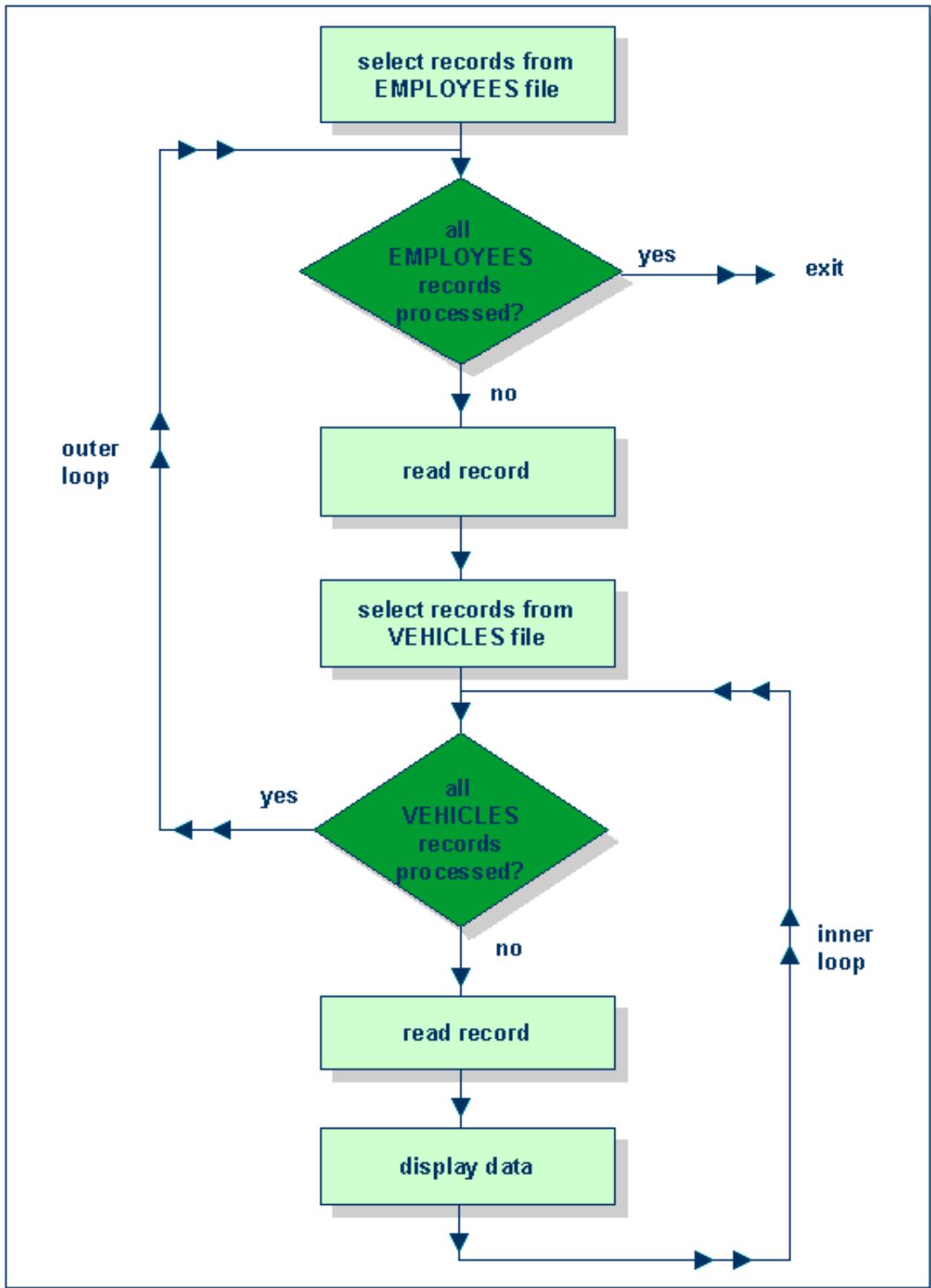
```
** Example Program 'FINDX04'  
DEFINE DATA LOCAL  
1 PERSONVIEW VIEW OF EMPLOYEES  
  2 PERSONNEL-ID  
  2 NAME  
1 AUTOVIEW VIEW OF VEHICLES  
  2 PERSONNEL-ID  
  2 MAKE  
  2 MODEL  
END-DEFINE  
*  
EMP. FIND PERSONVIEW WITH NAME = 'ADKINSON'  
VEH.  FIND AUTOVIEW WITH PERSONNEL-ID = PERSONNEL-ID (EMP.)  
      DISPLAY NAME MAKE MODEL  
      END-FIND  
      END-FIND  
END
```

The above program selects from the EMPLOYEES file all people with the name "ADKINSON". Each record (person) selected is then processed as follows:

1. The second FIND statement is executed to select the automobiles from the VEHICLES file, using as selection criterion the PERSONNEL-IDs from the records selected from the EMPLOYEES file with the first FIND statement.
2. The NAME of each person selected is displayed; this information is obtained from the EMPLOYEES file. The MAKE and MODEL of each automobile owned by that person is also displayed; this information is obtained from the VEHICLES file.

The second FIND statement creates an inner processing loop within the outer processing loop of the first FIND statement, as shown in the following diagram.

The diagram illustrates the flow logic of the hierarchy of processing loops in the previous example program:



Example of Nested FIND Loops Accessing the Same File:

It is also possible to construct a processing loop hierarchy in which the same file is used at both levels of the hierarchy:

```

** Example Program 'FINDX05'
DEFINE DATA LOCAL
1 PERSONVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
1 #NAME (A40)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED
  'PEOPLE IN SAME CITY AS:' #NAME / 'CITY:' CITY SKIP 1
FIND PERSONVIEW WITH NAME = 'JONES'
      WHERE FIRST-NAME = 'LAUREL'
  compress NAME FIRST-NAME INTO #NAME
  FIND PERSONVIEW WITH CITY = CITY
  DISPLAY NAME FIRST-NAME CITY
END-FIND
END-FIND
END

```

The above program first selects all people with name "JONES" and first name "LAUREL" from the EMPLOYEES file. Then all who live in the same city are selected from the EMPLOYEES file and a list of these people is created. All fields values displayed by the DISPLAY statement are taken from the second FIND statement.

PEOPLE IN SAME CITY AS: JONES LAUREL		
CITY: BALTIMORE		
NAME	FIRST-NAME	CITY

JENSEN	MARTHA	BALTIMORE
LAWLER	EDDIE	BALTIMORE
FORREST	CLARA	BALTIMORE
ALEXANDER	GIL	BALTIMORE
NEEDHAM	SUNNY	BALTIMORE
ZINN	CARLOS	BALTIMORE
JONES	LAUREL	BALTIMORE

Further Examples of Nested READ and FIND Statements:

See programs READX04 and LIMITX01 in library SYSEXP.

Database Update - Transaction Processing

- Logical Transaction
- Record Hold Logic
- Backing Out a Transaction
- Restarting a Transaction

Logical Transaction

Natural performs database updating operations based on *transactions*, which means that all database update requests are processed in logical transaction units. A logical transaction is the smallest unit of work (as defined by you) which must be performed in its entirety to ensure that the information contained in the database is logically consistent.

A logical transaction may consist of one or more update statements (DELETE, STORE, UPDATE) involving one or more database files. A logical transaction may also span multiple Natural programs.

A logical transaction begins when a record is put on "hold"; Natural does this automatically when the record is read for updating, for example, if a FIND loop contains an UPDATE or DELETE statement.

The end of a logical transaction is determined by an END TRANSACTION statement in the program. This statement ensures that all updates within the transaction have been successfully applied, and releases all records that were put on "hold" during the transaction.

Example:

```
DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
END-DEFINE
FIND MYVIEW WITH NAME = 'SMITH'
  DELETE
  END TRANSACTION
END-FIND
END
```

Each record selected would be put on "hold", deleted, and then - when the END TRANSACTION statement is executed - released from "hold".

Note:

The OPRB parameter, as set by the Natural administrator, determines whether or not Natural will generate an END TRANSACTION statement at the end of each Natural program. Ask your Natural administrator for details.

Example of STORE Statement:

See program STOREX01 in library SYSEXPG.

Record Hold Logic

If Natural is used with Adabas, any record which is to be updated will be placed in "hold" status until an END TRANSACTION or BACKOUT TRANSACTION statement is issued or the transaction time limit is exceeded.

When a record is placed in "hold" status for one user, the record is not available for update by another user. Another user who wishes to update the same record will be placed in "wait" status until the record is released from "hold" when the first user ends or backs out his/her transaction.

To prevent users from being placed in wait status, the session parameter WH (Wait Hold) can be used (see the Natural Reference documentation).

When you use update logic in a program, you should consider the following:

- The maximum time that a record can be in hold status is determined by the Adabas transaction time limit (Adabas parameter TT). If this time limit is exceeded, you will receive an error message and all database modifications done since the last END TRANSACTION will be made undone.
- The number of records on hold and the transaction time limit are affected by the size of a transaction, that is, by the placement of the END TRANSACTION statement in the program. Restart facilities should be considered when deciding where to issue an END TRANSACTION. For example, if a majority of records being processed are *not* to be updated, the GET statement is an efficient way of controlling the "holding" of records. This avoids issuing multiple END TRANSACTION statements and reduces the number of ISNs on hold. When you process large files, you should bear in mind that the GET statement requires an additional Adabas call. An example of a GET statement is shown below.

Example of GET Statement:

```

DEFINE DATA LOCAL
  1 EMPLOY-VIEW VIEW OF EMPLOYEES
    2 NAME
    2 SALARY (1)
END-DEFINE
RD. READ EMPLOY-VIEW BY NAME
  IF SALARY (1) > 30000
GE.  GET EMPLOY-VIEW *ISN (RD.)
      compute SALARY (1) = SALARY (1) * 1.15
      UPDATE (GE.)
      END TRANSACTION
  END-IF
END-READ
END

```

On mainframe computers, the placing of records in "hold" status is also controlled by the profile parameter RI, as set by the Natural administrator.

Backing Out a Transaction

During an active logical transaction, that is, before the END TRANSACTION statement is issued, you can cancel the transaction by using a BACKOUT TRANSACTION statement. The execution of this statement removes all updates that have been applied (including all records that have been added or deleted) and releases all records held by the transaction.

Restarting a Transaction

With the END TRANSACTION statement, you can also store transaction-related information. If processing of the transaction terminates abnormally, you can read this information with a GET TRANSACTION DATA statement to ascertain where to resume processing when you restart the transaction.

Example of Using Transaction Data to Restart a Transaction:

The following program updates the EMPLOYEES and VEHICLES files. After a restart operation, the user is informed of the last EMPLOYEES record successfully processed. The user can resume processing from that EMPLOYEES record. It would also be possible to set up the restart transaction message to include the last VEHICLES record successfully updated before the restart operation.

```

** Example Program 'GETTRX01'
DEFINE DATA LOCAL
01 PERSON VIEW OF EMPLOYEES
    02 PERSONNEL-ID      (A8)
    02 NAME              (A20)
    02 FIRST-NAME       (A20)
    02 MIDDLE-I         (A1)
    02 CITY              (A20)
01 AUTO VIEW OF VEHICLES
    02 PERSONNEL-ID      (A8)
    02 MAKE              (A20)
    02 MODEL             (A20)
01 ET-DATA
    02 #APPL-ID          (A8) INIT <' '>
    02 #USER-ID          (A8)
    02 #PROGRAM          (A8)
    02 #DATE             (A10)
    02 #TIME             (A8)
    02 #PERSONNEL-NUMBER (A8)
END-DEFINE
*
GET TRANSACTION DATA #APPL-ID #USER-ID #PROGRAM
                    #DATE      #TIME      #PERSONNEL-NUMBER
*
IF #APPL-ID NOT = 'NORMAL' /* IF LAST EXECUTION ENDED ABNORMALLY
    AND #APPL-ID NOT = ' '
    INPUT (AD=OIL)
        // 20T '*** LAST SUCCESSFUL TRANSACTION ***' (I)
        / 20T '*****'
    /// 25T      'APPLICATION:' #APPL-ID
        / 32T      'USER:' #USER-ID
        / 29T      'PROGRAM:' #PROGRAM
        / 24T      'COMPLETED ON:' #DATE 'AT' #TIME
        / 20T 'PERSONNEL NUMBER:' #PERSONNEL-NUMBER
END-IF
REPEAT
    INPUT (AD=MIL) // 20T 'ENTER PERSONNEL NUMBER:' #PERSONNEL-NUMBER
    IF #PERSONNEL-NUMBER = 99999999
        ESCAPE bottom
END-IF

```

```

FIND1. FIND PERSON WITH PERSONNEL-ID = #PERSONNEL-NUMBER
IF NO RECORDS FOUND
  REINPUT 'SPECIFIED NUMBER DOES NOT EXIST; ENTER ANOTHER ONE.'
END-NOREC
FIND2. FIND AUTO WITH PERSONNEL-ID = #PERSONNEL-NUMBER
IF NO RECORDS FOUND
  WRITE 'PERSON DOES NOT OWN ANY CARS'
END-NOREC
IF *COUNTER (FIND1.) = 1 /* FIRST PASS THROUGH THE LOOP
  INPUT (AD=M)
    / 20T 'EMPLOYEES/AUTOMOBILE DETAILS' (I)
    / 20T '-----'
  /// 20T 'NUMBER:' PERSONNEL-ID (AD=O)
    / 22T 'NAME:' NAME ' ' FIRST-NAME ' ' MIDDLE-I
    / 22T 'CITY:' CITY
    / 22T 'MAKE:' MAKE
    / 21T 'MODEL:' MODEL
  UPDATE (FIND1.) /* UPDATE THE EMPLOYEES FILE
ELSE /* SUBSEQUENT PASSES THROUGH THE LOOP
  INPUT NO ERASE (AD=M) ////////// 20T MAKE / 20T MODEL
END-IF
UPDATE (FIND2.) /* UPDATE THE VEHICLES FILE
MOVE *APPLIC-ID TO #APPL-ID
MOVE *INIT-USER TO #USER-ID
MOVE *PROGRAM TO #PROGRAM
MOVE *DAT4E TO #DATE
MOVE *TIME TO #TIME
END TRANSACTION #APPL-ID #USER-ID #PROGRAM
                #DATE #TIME #PERSONNEL-NUMBER
END-FIND /* FOR VEHICLES (FIND2.)
END-FIND /* FOR EMPLOYEES (FIND1.)
END-REPEAT /* FOR REPEAT
STOP /* Simulate abnormal transaction end
END TRANSACTION 'NORMAL '
END

```

Statements ACCEPT and REJECT

The statements ACCEPT and REJECT are used to select records based on user-specified logical criteria.

The statements ACCEPT and REJECT can be used in conjunction with the database access statements READ, FIND and HISTOGRAM.

Example of ACCEPT Statement:

```

** Example Program 'ACCEPX01'
  DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEES
    2 NAME
    2 JOB-TITLE
    2 CURR-CODE (1:1)
    2 SALARY (1:1)
  END-DEFINE
  READ (20) MYVIEW BY NAME WHERE CURR-CODE (1) = 'USD'
    ACCEPT IF SALARY (1) >= 40000
    DISPLAY NAME JOB-TITLE SALARY (1)
  END-READ
  END

```

Page	1	97-08-13	17:26:33
NAME	CURRENT POSITION	ANNUAL SALARY	
ADKINSON	DBA	46700	
ADKINSON	MANAGER	47000	
ADKINSON	MANAGER	47000	
AFANASSIEV	DBA	42800	
ALEXANDER	DIRECTOR	48000	
ANDERSON	MANAGER	50000	
ATHERTON	ANALYST	43000	
ATHERTON	MANAGER	40000	

ACCEPT/REJECT statements allow you to specify logical conditions in addition to those that were specified in WITH and WHERE clauses of the READ statement. The logical condition criteria in the IF clause of an ACCEPT/REJECT statement are evaluated *after* the record has been selected and read.

Logical condition operators include the following (see the Natural Reference documentation for more detailed information):

EQUAL	EQ	:=
NOT EQUAL TO	NE	≠
LESS THAN	LT	<
LESS EQUAL	LE	<=
GREATER THAN	GT	>
GREATER EQUAL	GE	>=

Logical condition criteria in ACCEPT/REJECT statements may also be connected with the Boolean operators AND, OR, and NOT. Moreover, parentheses may be used to indicate logical grouping.

Example of ACCEPT Statement with AND Operator:

The following program illustrates the use of the Boolean operator AND in an ACCEPT statement.

```

** Example Program 'ACCEPX02'
  DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEES
    2 NAME
    2 JOB-TITLE
    2 CURR-CODE (1:1)
    2 SALARY    (1:1)
  END-DEFINE
  READ (20) MYVIEW BY NAME WHERE CURR-CODE (1) = 'USD'
    ACCEPT IF SALARY (1) >= 40000
      AND SALARY (1) <= 45000
    DISPLAY NAME JOB-TITLE SALARY (1)
  END-READ
  END

```

Example of REJECT Statement with OR Operator:

The following program, which uses the Boolean operator OR in a REJECT statement, produces the same output as the ACCEPT statement above, as the logical operators are reversed.

```

** Example Program 'ACCEPX03'
  DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEES
    2 NAME
    2 JOB-TITLE
    2 CURR-CODE (1:1)
    2 SALARY    (1:1)
  END-DEFINE
  READ (20) MYVIEW BY NAME WHERE CURR-CODE (1) = 'USD'
    REJECT IF SALARY (1) < 40000
      OR SALARY (1) > 45000
    DISPLAY NAME JOB-TITLE SALARY (1)
  END-READ
  END

```

Page	1		97-08-18 12:21:09
	NAME	CURRENT POSITION	ANNUAL SALARY
	-----	-----	-----
	AFANASSIEV	DBA	42800
	ATHERTON	ANALYST	43000
	ATHERTON	MANAGER	40000

Further Examples of ACCEPT and REJECT Statements:

See programs ACCEPX04, ACCEPX05 and ACCEPX06 in library SYSEXP.

AT START/END OF DATA Statements

AT START OF DATA Statement

The AT START OF DATA statement is used to specify any processing that is to be performed after the first of a set of records has been read in a database processing loop.

The AT START OF DATA statement must be placed within the processing loop.

If the AT START OF DATA processing produces any output, this will be output *before the first field value*. By default, this output is displayed left-justified on the page.

AT END OF DATA Statement

The AT END OF DATA statement is used to specify processing that is to be performed after all records for a database processing loop have been processed.

The AT END OF DATA statement must be placed within the processing loop.

If the AT END OF DATA processing produces any output, this will be output *after the last field value*. By default, this output is displayed left-justified on the page.

Example of AT START OF DATA and AT END OF DATA Statements

The following example program illustrates the use of the statements AT START OF DATA and AT END OF DATA. The system variable *TIME has been incorporated into the AT START OF DATA statement to display the time of day. The system function OLD has been incorporated into the AT END OF DATA statement to display the name of the last person selected.

```

** Example Program 'ATSTAX01'
  DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEES
    2 CITY
    2 NAME
    2 JOB-TITLE
    2 INCOME (1:1)
    3 CURR-CODE
    3 SALARY
    3 BONUS (1:1)
  END-DEFINE
  WRITE TITLE 'XYZ EMPLOYEE ANNUAL SALARY AND BONUS REPORT' /
  READ (3) MYVIEW BY CITY STARTING FROM 'E'
  DISPLAY GIVE SYSTEM FUNCTIONS
    NAME (AL=15) JOB-TITLE (AL=15) INCOME (1)
  AT START OF DATA
    WRITE 'RUN TIME:' *TIME /
  END-START
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD (NAME) /
  END-ENDDATA
  END-READ
  AT END OF PAGE
    WRITE / 'AVERAGE SALARY:' AVER (SALARY(1))
  END-ENDPAGE
  END

```

The program produces the following output:

XYZ EMPLOYEE ANNUAL SALARY AND BONUS REPORT				
NAME	CURRENT POSITION	INCOME		
		CURRENCY CODE	ANNUAL SALARY	BONUS

RUN TIME: 11:18:58.2				
DUYVERMAN	PROGRAMMER	USD	34000	0
PRATT	SALES PERSON	USD	38000	9000
MARKUSH	TRAINEE	USD	22000	0
LAST PERSON SELECTED: MARKUSH				
AVERAGE SALARY:		31333		

Further Examples of AT START OF DATA and AT END OF DATA Statements:

See programs ATENDX01, ATSTAX02 and WRITEX09 in library SYSEXP.

Output of Data

This section discusses various aspects of how you can control the format of an output report created with Natural, that is, the way in which the data are displayed.

It covers the following topics:

- Layout of an Output Page - Overview
 - Statements DISPLAY and WRITE
 - Index Notation (n:n) for Multiple-Value Fields and Periodic Groups
 - Page Titles and Page Breaks
 - Column Headers
 - Parameters to Influence the Output of Fields
 - Edit Masks - The EM Parameter
 - Vertical Displays
-

Layout of an Output Page - Overview

The following program illustrates the general layout of an output page:

```

** Example Program 'OUTPUX01'
  DEFINE DATA LOCAL
  1 EMP-VIEW VIEW OF EMPLOYEES
    2 NAME
    2 FIRST-NAME
    2 BIRTH
  END-DEFINE
  *
  WRITE TITLE '***** Page Title *****'
  WRITE TRAILER '***** Page Trailer *****'
  AT TOP OF PAGE
    WRITE '==== Top of Page ====='
  END-TOPPAGE
  AT END OF PAGE
    WRITE '==== End of Page ====='
  END-ENDPAGE
  READ (10) EMP-VIEW BY NAME
  DISPLAY NAME FIRST-NAME BIRTH (EM=YYY-MM-DD)
  AT START OF DATA
    WRITE '>>>> Start of Data >>>>'
  END-START
  AT END OF DATA
    WRITE '<<<< End of Data <<<<'
  END-ENDDATA
  END-READ
  END

```

```

***** Page Title *****
===== Top of Page =====
          NAME                FIRST-NAME                DATE
                              OF
                              BIRTH
-----
>>>> Start of Data >>>>
ABELLAN                KEPA                1961-04-08
ACHIESON                ROBERT                1963-12-24
ADAM                    SIMONE                1952-01-30
ADKINSON                JEFF                1951-06-15
ADKINSON                PHYLLIS                1956-09-17
ADKINSON                HAZEL                1954-03-19
ADKINSON                DAVID                1946-10-12
ADKINSON                CHARLIE                1950-03-02
ADKINSON                MARTHA                1970-01-01
ADKINSON                TIMMIE                1970-03-03
<<<<< End of Data <<<<<
***** Page Trailer *****
===== End of Page =====

```

The following statements have an impact on the layout of the report:

Statement	Function
WRITE TITLE	With this statement, you can specify a page title, that is, text to be output at the top of a page.
WRITE TRAILER	With this statement, you can specify a page trailer, that is, text to be output at the bottom of a page.
AT TOP OF PAGE	With this statement, you can specify any processing that is to be performed whenever a new page of the report is started. Any output from this processing will be output below the page title.
AT END OF PAGE	With this statement, you can specify any processing that is to be performed whenever an end-of-page condition occurs. Any output from this processing will be output below any page trailer (as specified with the WRITE TRAILER statement).
AT START OF DATA	With this statement, you specify processing that is to be performed after the first record has been read in a database processing loop. Any output from this processing will be output before the first field value.
AT END OF DATA	With this statement, you specify processing that is to be performed after all records for a processing loop have been processed. Any output from this processing will be output immediately after the last field value .
DISPLAY/WRITE	With these statements, you control the format in which the field values that have been read are to be output.

The statements AT START OF DATA and AT END OF DATA are described in the section Database Access. The other statements listed above are described below.

Statements DISPLAY and WRITE

With the statements DISPLAY and WRITE, you output data and control the format in which information is output.

- DISPLAY Statement
- WRITE Statement
- Column Spacing - The SF Parameter and the nX Notation
- Tab Setting - The nT Notation
- Line Advance - The / Notation

DISPLAY Statement

The DISPLAY statement produces output in column format; that is, the values for one field are output in a column underneath one another. If multiple fields are output, that is, if multiple columns are produced, these columns are output next to one another horizontally.

The order in which fields are displayed is determined by the sequence in which you specify the field names in the DISPLAY statement.

The DISPLAY statement in the following program displays for each person first the personnel number, then the name and then the job title:

```

** Example Program 'DISPLX01'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 PERSONNEL-ID
    2 NAME
    2 BIRTH
    2 JOB-TITLE
  END-DEFINE
  READ (3) VIEWEMP BY BIRTH
    DISPLAY PERSONNEL-ID NAME JOB-TITLE
  END-READ
  END
    
```

Page	1		99-01-22	11:31:01
PERSONNEL ID	NAME	CURRENT POSITION		

30020013	GARRET	TYPIST		
30016112	TAILOR	WAREHOUSEMAN		
20017600	PIETSCH	SECRETARY		

To change the order of the columns that appear in the output report, simply reorder the field names in the DISPLAY statement. For example, if you prefer to list employee names first, then job titles followed by personnel numbers, the appropriate DISPLAY statement would be:

```

** Example Program 'DISPLX02'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 PERSONNEL-ID
    2 NAME
    2 BIRTH
    2 JOB-TITLE
  END-DEFINE
  READ (3) VIEWEMP BY BIRTH
    DISPLAY NAME JOB-TITLE PERSONNEL-ID
  END-READ
  END
    
```

Page	1		99-01-22 11:32:06
	NAME	CURRENT POSITION	PERSONNEL ID
	-----	-----	-----
	GARRET	TYPIST	30020013
	TAILOR	WAREHOUSEMAN	30016112
	PIETSCH	SECRETARY	20017600

A header is output above each column. Various ways to influence this header are described later in this section.

WRITE Statement

The WRITE statement is used to produce output in free format (that is, not in columns). In contrast to the DISPLAY statement, the following applies to the WRITE statement:

- If necessary, it automatically creates a line advance; that is, a field or text element that does not fit onto the current output line, is automatically output in the next line.
- It does not produce any headers.
- The values of a multiple-value field are output next to one another horizontally, and not underneath one another.

The two example programs on the following page illustrate the basic differences between the DISPLAY statement and the WRITE statement.

You can also use the two statements in combination with one another, as described later in this section.

Example of DISPLAY Statement:

```

** Example Program 'DISPLX03'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 NAME
    2 FIRST-NAME
    2 SALARY (1:3)
  END-DEFINE
  READ (2) VIEWEMP BY NAME STARTING FROM 'JONES'
    DISPLAY NAME FIRST-NAME SALARY (1:3)
  END-READ
  END

```

Page	1		97-08-14 11:44:00
	NAME	FIRST-NAME	ANNUAL SALARY
	-----	-----	-----
	JONES	VIRGINIA	46000 42300 39300
	JONES	MARSHA	50000 46000 42700

Example of WRITE Statement:

```

** Example Program 'WRITEX01'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 NAME
    2 FIRST-NAME
    2 SALARY (1:3)
  END-DEFINE
  READ (2) VIEWEMP BY NAME STARTING FROM 'JONES'
    WRITE NAME FIRST-NAME SALARY (1:3)
  END-READ
  END

```

Page	1			97-08-14	11:45:00
JONES		VIRGINIA	46000	42300	
39300					
JONES		MARSHA	50000	46000	
42700					

Column Spacing - The SF Parameter and the nX Notation

By default, the columns output with a DISPLAY statement are separated from one another by *one* space.

With the session parameter SF, you can specify the default number of spaces to be inserted between columns output with a DISPLAY statement. You can set the number of spaces to any value from 1 to 30.

The parameter can be specified with a FORMAT statement to apply to the whole report, or with a DISPLAY statement at statement level, but not at field level.

With the *nX* notation, you can specify the number of spaces (*n*) to be inserted between two columns.

An *nX* notation overrides the specification made with the SF parameter.

```

** Example Program 'DISPLX04'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 PERSONNEL-ID
    2 NAME
    2 BIRTH
    2 JOB-TITLE
  END-DEFINE
  FORMAT SF=3
  READ (3) VIEWEMP BY BIRTH
    DISPLAY PERSONNEL-ID NAME 5X JOB-TITLE
  END-READ
  END

```

The above example program produces the following output, where the first two columns are separated by 3 spaces due to the SF parameter in the FORMAT statement, while the second and third columns are separated by 5 spaces due to the notation "5X" in the DISPLAY statement:

Page	1		99-01-22	11:33:40
PERSONNEL ID	NAME		CURRENT POSITION	
-----	-----		-----	
30020013	GARRET		TYPIST	
30016112	TAILOR		WAREHOUSEMAN	
20017600	PIETSCH		SECRETARY	

The *nX* notation is also available with the WRITE statement to insert spaces between individual output elements:

```
WRITE PERSONNEL-ID 5X NAME 3X JOB-TITLE
```

With the above statement, 5 spaces will be inserted between the fields PERSONNEL-ID and NAME, and 3 spaces between NAME and JOB-TITLE.

Tab Setting - The nT Notation

With the *nT* notation, which is available with the DISPLAY and the WRITE statement, you can specify the position where an output element is to be output.

```
** Example Program 'DISPLX05'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 NAME
    2 FIRST-NAME
  END-DEFINE
  READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
    DISPLAY 5T NAME 30T FIRST-NAME
  END-READ
  END
```

The above program produces the following output, where the field NAME is output starting in the 5th position (counted from the left margin of the page), and the field FIRST-NAME starting in the 30th position:

Page	1	97-08-21	11:46:01
	NAME	FIRST-NAME	
	-----	-----	
	JONES	VIRGINIA	
	JONES	MARSHA	
	JONES	ROBERT	

Line Advance - The / Notation

With a slash "/" in a DISPLAY or WRITE statement, you cause a line advance.

- In a DISPLAY statement, a slash causes a line advance *between fields* and *within text*.
- In a WRITE statement, a slash causes a line advance only when placed *between fields*; within text, it is treated like an ordinary text character.

When placed between fields, the slash must have a blank on either side.

For multiple line advances, you specify multiple slashes.

Example of Line Advance in DISPLAY Statement:

```
** Example Program 'DISPLX06'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 NAME
    2 FIRST-NAME
    2 DEPARTMENT
  END-DEFINE
  READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
    DISPLAY NAME / FIRST-NAME 'DEPART-/MENT' DEPARTMENT
  END-READ
  END
```

The above DISPLAY statement produces a line advance after each value of the field NAME and within the text "DEPART-MENT":

Page	1	97-08-14	11:45:12
	NAME	DEPART-	
	FIRST-NAME	MENT	
	-----	-----	
	JONES	SALE	
	VIRGINIA		
	JONES	MGMT	
	MARSHA		
	JONES	TECH	
	ROBERT		

Example of Line Advance in WRITE Statement:

```

** Example Program 'WRITEX02'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 NAME
    2 FIRST-NAME
    2 DEPARTMENT
  END-DEFINE
  READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
    WRITE NAME / FIRST-NAME 'DEPART-/MENT' DEPARTMENT //
  END-READ
  END

```

The above WRITE statement produces a line advance after each value of the field NAME, and a double line advance after each value of the field DEPARTMENT, but none within the text "DEPART-/MENT":

Page	1	97-08-14	11:45:12
	JONES		
	VIRGINIA	DEPART-/MENT	SALE
	JONES		
	MARSHA	DEPART-/MENT	MGMT
	JONES		
	ROBERT	DEPART-/MENT	TECH

Further Examples of DISPLAY and WRITE Statements:

See programs DISPLX13, WRITEX08, DISPLX14, WRITEX09 and DISPLX21 in library SYSEXP.

Index Notation (*n:n*) for Multiple-Value Fields and Periodic Groups

With the index notation (*n:n*) you can specify how many values of a multiple-value field or how many occurrences of a periodic group are to be output.

For example, the field INCOME in the DDM EMPLOYEES is a periodic group which keeps a record of the annual incomes of an employee for each year he/she has been with the company. These annual incomes are maintained in chronological order. The income of the most recent year is in occurrence "1". If you wanted to have the annual incomes of an employee for the last three years displayed - that is, occurrences "1" to "3" - you would specify the notation "(1:3)" after the field name in a DISPLAY or WRITE statement (as shown in the following example program).

Example of Index Notation in DISPLAY Statement:

```

** Example Program 'DISPLX07'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 PERSONNEL-ID
    2 NAME
    2 BIRTH
    2 INCOME (1:3)
      3 CURR-CODE
      3 SALARY
      3 BONUS (1:1)
  END-DEFINE
  READ (3) VIEWEMP BY BIRTH
    DISPLAY PERSONNEL-ID NAME INCOME (1:3)
  SKIP 1
  END-READ
  END

```

Note that a DISPLAY statement outputs multiple values of a multiple-value field underneath one another:

Page		1		99-01-22 11:36:58	
PERSONNEL ID	NAME	CURRENCY CODE	ANNUAL SALARY	BONUS	
30020013	GARRET	UKL	4200	0	
		UKL	4150	0	
			0	0	
30016112	TAILOR	UKL	7450	0	
		UKL	7350	0	
		UKL	6700	0	
20017600	PIETSCH	USD	22000	0	
		USD	20200	0	
		USD	18700	0	

As a WRITE statement displays multiple values horizontally instead of vertically, this may cause a line overflow and a - possibly undesired - line advance.

If you use only a single field within a periodic group (for example, SALARY) instead of the entire periodic group, and if you also insert a line advance "/" (as shown in the following example between NAME and JOB-TITLE), the report format becomes manageable:

Example of Index Notation in WRITE Statement:

```

** Example Program 'WRITEX03'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 PERSONNEL-ID
    2 NAME
    2 BIRTH
    2 JOB-TITLE
    2 SALARY (1:3)
  END-DEFINE
  READ (3) VIEWEMP BY BIRTH
    WRITE PERSONNEL-ID NAME / JOB-TITLE SALARY (1:3)
  SKIP 1
  END-READ
  END

```

Page	1			99-01-22	11:37:18
30020013	GARRET				
TYPIST		4200	4150	0	
30016112	TAILOR				
WAREHOUSEMAN		7450	7350	6700	
20017600	PIETSCH				
SECRETARY		22000	20200	18700	

Page Titles and Page Breaks

This section describes various ways of controlling page breaks in a report and the output of page titles at the top of each report page.

- Default Page Title
- Suppress Page Title - The NOTITLE Option
- Define Your Own Page Title - The WRITE TITLE Statement
- Logical Page and Physical Page
- Page Size - The PS Parameter
- Page Advance - The EJ Parameter
- Page Advance - The EJECT and NEWPAGE Statements
- Page Trailer - The WRITE TRAILER Statement
- AT TOP OF PAGE Statement
- AT END OF PAGE Statement

Default Page Title

For each page output via a DISPLAY or WRITE statement, Natural automatically generates a single default title line. This title line contains the page number, the date and the time of day.

```
WRITE 'HELLO'
END
```

The above program produces the following output with default page title:

Page	1	97-08-14	18:27:35
HELLO			

Suppress Page Title - The NOTITLE Option

If you wish your report to be output without page titles, you add the keyword "NOTITLE" to the DISPLAY or WRITE statement.

```
WRITE NOTITLE 'HELLO'
END
```

The above program produces the following output without page title:

HELLO

Define Your Own Page Title - The WRITE TITLE Statement

If you wish a page title of your own to be output instead of the Natural default page title, you use the statement WRITE TITLE. With this statement, you specify the text for your title (in apostrophes).

```
WRITE TITLE 'THIS IS MY PAGE TITLE'
WRITE 'HELLO'
END
```

HELLO	THIS IS MY PAGE TITLE
-------	-----------------------

With the SKIP option of the WRITE TITLE statement, you can specify the number of empty lines to be output immediately below the title line. After the keyword SKIP, you specify the number of empty lines to be inserted.

```
WRITE TITLE 'THIS IS MY PAGE TITLE' SKIP 2
WRITE 'HELLO'
END
```

```

                THIS IS MY PAGE TITLE

HELLO
```

SKIP is not only available as part of the WRITE TITLE statement, but also as a stand-alone statement.

By default, the page title is centered on the page and not underlined. However, the WRITE TITLE statement provides the options LEFT JUSTIFIED and UNDERLINED to display the title left-justified and/or underlined.

```
WRITE TITLE LEFT JUSTIFIED UNDERLINED 'THIS IS MY PAGE TITLE' SKIP 2
WRITE 'HELLO'
END
```

```
THIS IS MY PAGE TITLE
-----
HELLO
```

By default, titles are underlined with a hyphen (-). However, with the UC parameter you can specify another character to be used as underlining character (as described later in this section).

The WRITE TITLE statement is executed whenever a new page is initiated for the report.

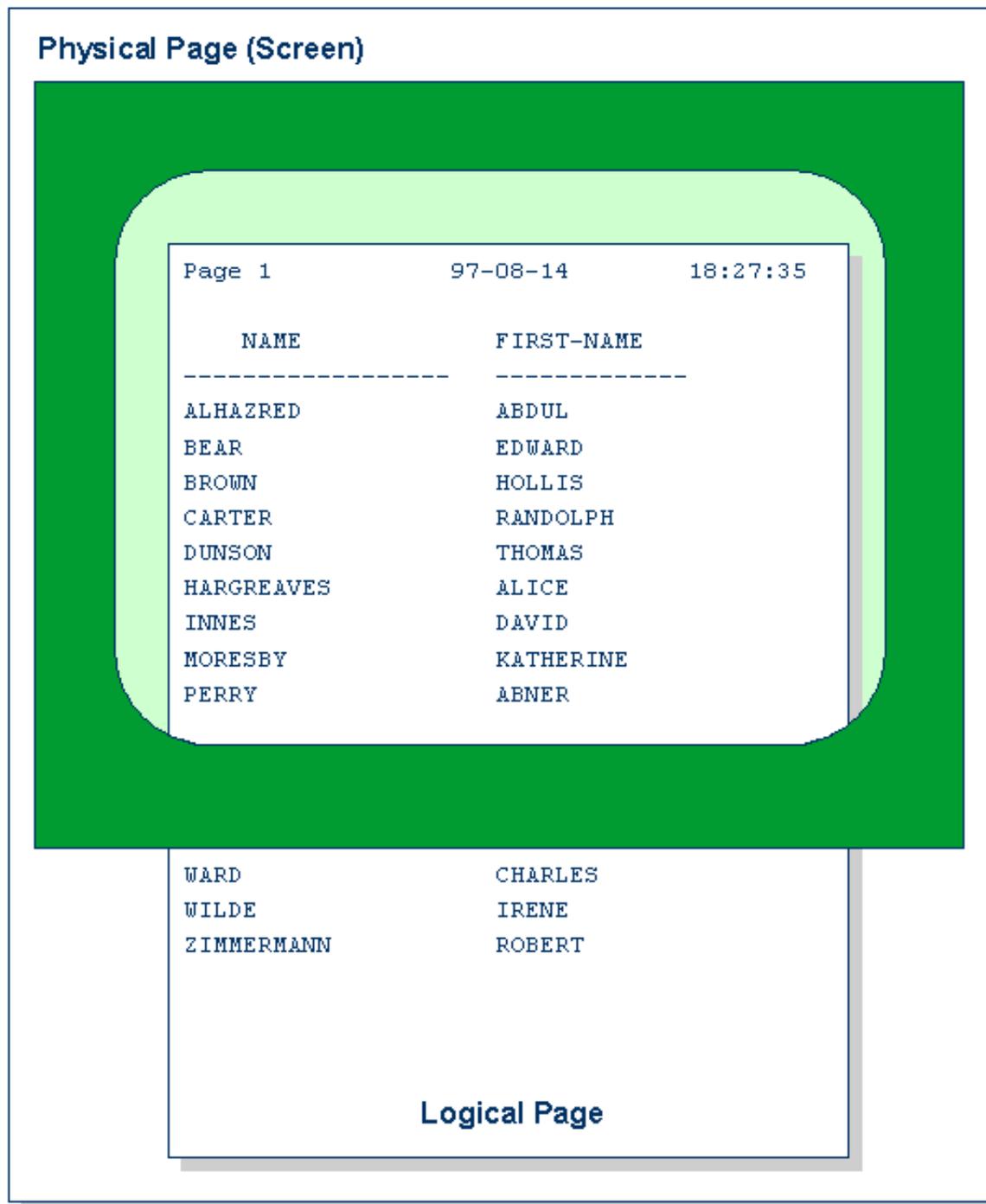
Logical Page and Physical Page

A *logical page* is the output produced by a Natural program.

A *physical page* is your terminal screen on which the output is displayed; or it may be the piece of paper on which the output is printed.

The size of the logical page is determined by the number of lines output by the Natural program.

If more lines are output than fit onto one screen, the logical page will exceed the physical screen, and the remaining lines will be displayed on the next screen.



If information you wish to appear at the bottom of the screen (for example, output created by a WRITE TRAILER or AT END OF PAGE statement) is output on the next screen instead, reduce the logical page size accordingly (with the session parameter PS, which is described below).

Page Size - The PS Parameter

With the parameter PS, you determine the maximum number of lines per (logical) page for a report.

When the number of lines specified with the PS parameter is reached, a page advance occurs (unless page advance is controlled with a NEWPAGE or EJECT statement; see below).

The PS parameter can be set either at session level with the system command GLOBALS, or within a program with the following statements:

- at report level:
FORMAT PS=*nn*
- at statement level:
DISPLAY (PS=*nn*)
WRITE (PS=*nn*)
WRITE TITLE (PS=*nn*)
WRITE TRAILER (PS=*nn*)
INPUT (PS=*nn*)

Page Advance - The EJ Parameter

With the session parameter EJ, you determine whether page ejects are to be performed or not. By default, EJ=ON applies, which means that page ejects will be performed as specified. If you specify EJ=OFF, page break information will be ignored. This may be useful to save paper during test runs where page ejects are not needed.

The EJ parameter can be set at session level with the system command GLOBALS; for example:

GLOBALS EJ=OFF

Page Advance - The EJECT and NEWPAGE Statements

The EJECT statement overrides the EJ parameter setting. The EJECT statement causes a page advance *without* a title or header line being generated on the next page. A new physical page is started *without* any top-of-page or end-of-page processing being performed (for example, no WRITE TRAILER or AT END OF PAGE , WRITE TITLE, AT TOP OF PAGE or *PAGE-NUMBER processing).

The NEWPAGE statement causes a page advance *with* associated end-of-page and top-of-page processing. A trailer line will be displayed, if specified. A title line, either default or user-specified, will be displayed on the new page (unless the NOTITLE option has been specified in a DISPLAY or WRITE statement).

If the NEWPAGE statement is not used, page advance is automatically controlled by the setting of the PS parameter (see above).

EJECT/NEWPAGE WHEN LESS THAN n LINES LEFT

Both the NEWPAGE statement and the EJECT statement provide a WHEN LESS THAN n LINES LEFT option. With this option, you specify a number of lines n . The NEWPAGE/EJECT statement will then be executed if - at the time the statement is processed - less than n lines are available on the current page.

Example:

```
FORMAT PS=55
...
NEWPAGE WHEN LESS THAN 7 LINES LEFT
...
```

In this example, the page size is set to 55 lines.

If only 6 or less lines are left on the current page at the time when the NEWPAGE statement is processed, the NEWPAGE statement is executed and a page advance occurs.

If 7 or more lines are left, the NEWPAGE statement is not executed and no page advance occurs; the page advance then occurs depending on the PS parameter, that is, after 55 lines.

NEWPAGE WITH TITLE

The NEWPAGE statement also provides a WITH TITLE option. If this option is not used, a default title will appear at the top of the new page or a WRITE TITLE statement or NOTITLE clause will be executed. The WITH TITLE option of the NEWPAGE statement allows you to override these with a title of your own choice. The syntax of the WITH TITLE option is the same as for the WRITE TITLE statement.

Example:

```
NEWPAGE WITH TITLE LEFT JUSTIFIED 'PEOPLE LIVING IN BOSTON:'
```

The following program illustrates the use of the PS parameter and the NEWPAGE statement. Moreover, the system variable *PAGE-NUMBER is used to display the current page number.

```
** Example Program 'NEWPAX01'
  DEFINE DATA LOCAL
  1 VIEWEMP OF EMPLOYEES
    2 NAME
    2 CITY
    2 DEPT
  END-DEFINE
  FORMAT PS=20
  READ (5) VIEWEMP BY CITY STARTING FROM 'M'
    DISPLAY NAME 'DEPT' DEPT 'LOCATION' CITY
    AT BREAK OF CITY
      NEWPAGE WITH TITLE LEFT JUSTIFIED
        'EMPLOYEES BY CITY - PAGE:' *PAGE-NUMBER
    END-BREAK
  END-READ
  END
```

Note the position of the page breaks and the title line printed on the new page:

Page	1	97-08-19	18:27:35
	NAME	DEPT	LOCATION
	-----	-----	-----
	FICKEN	TECH10	MADISON
	KELLOGG	TECH10	MADISON
	ALEXANDER	SALE20	MADISON

EMPLOYEES BY CITY - PAGE:	2		
	NAME	DEPT	LOCATION
	-----	-----	-----
	DE JUAN	SALE03	MADRID
	DE LA MADRID	PROD01	MADRID

Page Trailer - The WRITE TRAILER Statement

The WRITE TRAILER statement is used to output text (in apostrophes) at the bottom of a page.

```
WRITE TRAILER 'THIS IS THE END OF THE PAGE'
```

The statement is executed when an end-of-page condition is detected, or as a result of a SKIP or NEWPAGE statement.

As the end-of-page condition is checked only *after* an entire DISPLAY or WRITE statement has been processed, it may occur that the logical page size (that is, the number of lines output by a DISPLAY or WRITE statement) causes the physical size of the output page to be exceeded before the WRITE TRAILER statement is executed. To ensure that a page trailer actually appears at the bottom of a physical page, you should set the logical page size (with the PS session parameter) to a value less than the physical page size.

By default, the page trailer is displayed centered on the page and not underlined. However, the WRITE TRAILER statement provides the options LEFT JUSTIFIED and UNDERLINED to display the trailer left-justified and/or underlined:

```
WRITE TRAILER LEFT JUSTIFIED UNDERLINED 'THIS IS THE END OF THE PAGE'
```

AT TOP OF PAGE Statement

The AT TOP OF PAGE statement is used to specify any processing that is to be performed whenever a new page of the report is started.

If the AT TOP OF PAGE processing produces any output, this will be output below the page title (with a skipped line in between). By default, this output is displayed left-justified on the page.

AT END OF PAGE Statement

The AT END OF PAGE statement is used to specify any processing that is to be performed whenever an end-of-page condition occurs.

If the AT END OF PAGE processing produces any output, this will be output after any page trailer (as specified with the WRITE TRAILER statement). By default, this output is displayed left-justified on the page.

The same considerations described above for page trailers regarding physical and logical page sizes and the number of lines output by a DISPLAY or WRITE statement also apply to AT END OF PAGE output.

Further Examples of WRITE TITLE, WRITE TRAILER, AT TOP OF PAGE, AT END OF PAGE and SKIP Statements:

See programs WTITLX01, DISPLX21, ATENPX01, ATTOPX01, SKIPX01 and SKIPX02 in library SYSEXPG.

Further Example of NOTITLE Option:

See program DISPLX20 in library SYSEXPG.

Further Example of NEWPAGE and EJECT Statements:

See program NEWPAX02 in library SYSEXPG.

Column Headers

This section describes various ways of controlling the display of column headers produced by a DISPLAY statement.

- Default Column Headers
- Suppress Default Column Headers - The NOHDR Option
- Define Your Own Column Headers
- Combining NOTITLE and NOHDR
- Centering of Column Headers - The HC Parameter
- Width of Column Headers - The HW Parameter
- Filler Characters for Headers - The Parameters FC and GC
- Underlining Character for Titles and Headers - The UC Parameter
- Suppressing Column Headers - The Notation '/'

Default Column Headers

By default, each database field output with a DISPLAY statement is displayed with a default column header (which is defined for the field in the DDM).

```

** Example Program 'DISPLX01'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 PERSONNEL-ID
    2 NAME
    2 BIRTH
    2 JOB-TITLE
  END-DEFINE
  READ (3) VIEWEMP BY BIRTH
    DISPLAY PERSONNEL-ID NAME JOB-TITLE
  END-READ
  END

```

The above example program uses default headers and produces the following output:

Page	1	99-01-22	11:31:01
PERSONNEL ID	NAME	CURRENT POSITION	

30020013	GARRET	TYPIST	
30016112	TAILOR	WAREHOUSEMAN	
20017600	PIETSCH	SECRETARY	

Suppress Default Column Headers - The NOHDR Option

If you wish your report to be output without column headers, add the keyword "NOHDR" to the DISPLAY statement.

```
DISPLAY NOHDR PERSONNEL-ID NAME JOB-TITLE
```

Define Your Own Column Headers

If you wish column headers of your own to be output instead of the default headers, you specify *'text'* (in apostrophes) immediately before a field, *text* being the header to be used for the field.

```
** Example Program 'DISPLX08'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 PERSONNEL-ID
    2 NAME
    2 BIRTH
    2 JOB-TITLE
  END-DEFINE
  READ (3) VIEWEMP BY BIRTH
    DISPLAY PERSONNEL-ID
           'EMPLOYEE' NAME
           'POSITION' JOB-TITLE
  END-READ
  END
```

The above program contains the header "EMPLOYEE" for the field NAME, and the header "POSITION" for the field JOB-TITLE; for the field PERSONNEL-ID, the default header is used. The program produces the following output:

Page	1		99-01-22 11:39:53
PERSONNEL ID	EMPLOYEE	POSITION	

30020013	GARRET	TYPIST	
30016112	TAILOR	WAREHOUSEMAN	
20017600	PIETSCH	SECRETARY	

Combining NOTITLE and NOHDR

To create a report that has neither page title nor column headers, you specify the NOTITLE and NOHDR options together in the following order:

```
DISPLAY NOTITLE NOHDR PERSONNEL-ID NAME JOB-TITLE
```

Centering of Column Headers - The HC Parameter

By default, column headers are centered above the columns. With the HC parameter, you can influence the placement of column headers:

- If you specify **HC=L**, headers will be left-justified.
- If you specify **HC=R**, headers will be right-justified.
- If you specify **HC=C**, headers will be centered.

The HC parameter can be used in a FORMAT statement to apply to the whole report, or it can be used in a DISPLAY statement at both statement level and field level.

```
DISPLAY (HC=L) PERSONNEL-ID NAME JOB-TITLE
```

Width of Column Headers - The HW Parameter

With the HW parameter, you determine the width of a column output with a DISPLAY statement.

- If you specify **HW=ON**, the width of a DISPLAY column is determined by either the length of the header text or the length of the field, whichever is longer. This also applies by default.
- If you specify **HW=OFF**, the width of a DISPLAY column is determined only by the length of the field. However, **HW=OFF** only applies to DISPLAY statements which do *not* create headers; that is, either a first DISPLAY statement with NOHDR option or a subsequent DISPLAY statement (see also the Natural Reference documentation).

The HW parameter can be used in a FORMAT statement to apply to the entire report, or it can be used in a DISPLAY statement at both statement level and field level.

Filler Characters for Headers - The Parameters FC and GC

With the FC parameter, you specify the *filler character* which will appear on either side of a *header* produced by a DISPLAY statement across the full column width if the column width is determined by the field length and not by the header (see HW parameter above); otherwise FC will be ignored.

When a group of fields or a periodic group is output via a DISPLAY statement, a *group header* is displayed across all field columns that belong to that group above the headers for the individual fields within the group. With the GC parameter, you can specify the *filler character* which will appear on either side of such a group header.

While the FC parameter applies to the headers of individual fields, the GC parameter applies to the headers for groups of fields.

The parameters FC and GC can be specified in a FORMAT statement to apply to the whole report, or they can be specified in a DISPLAY statement at both statement level and field level.

```
** Example Program 'FORMAX01'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 NAME
    2 INCOME (1:1)
    3 CURR-CODE
```

```

3 SALARY
3 BONUS (1:1)
END-DEFINE
FORMAT FC=* GC=$
READ (3) VIEWEMP BY NAME
    DISPLAY NAME (FC==) INCOME (1)
END-READ
END
    
```

The above program produces the following output:

Page	1		97-08-19	17:37:27
=====NAME===== \$\$\$\$\$\$\$\$\$\$INCOME\$\$\$\$\$\$\$\$\$\$\$\$				
		CURRENCY	**ANNUAL**	**BONUS**
		CODE	SALARY	

ABELLAN	PTA	1450000		0
ACHIESON	UKL	10500		0
ADAM	FRA	159980	23000	

Underlining Character for Titles and Headers - The UC Parameter

By default, titles and headers are underlined with a hyphen (-).

With the UC parameter, you can specify another character to be used as underlining character.

The UC parameter can be specified in a FORMAT statement to apply to the whole report, or it can be specified in a DISPLAY statement at both statement level and field level.

```

** Example Program 'FORMAX02'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 PERSONNEL-ID
    2 NAME
    2 BIRTH
    2 JOB-TITLE
  END-DEFINE
  FORMAT UC==
  WRITE TITLE LEFT JUSTIFIED UNDERLINED 'EMPLOYEES REPORT' SKIP 1
  READ (3) VIEWEMP BY BIRTH
    DISPLAY PERSONNEL-ID (UC=*) NAME JOB-TITLE
  END-READ
  END

```

In the above program, the UC parameter is specified at program level and at field level: the underlining character specified with the FORMAT statement (=) applies for the whole report - except for the field PERSONNEL-ID, for which a different underlining character (*) is specified. The program produces the following output:

EMPLOYEES REPORT		
=====		
PERSONNEL ID	NAME	CURRENT POSITION
*****	=====	=====
30020013	GARRET	TYPIST
30016112	TAILOR	WAREHOUSEMAN
20017600	PIETSCH	SECRETARY

Suppressing Column Headers - The Notation '/'

With the notation apostrophe-slash-apostrophe ('/'), you can suppress default column headers for individual fields displayed with a DISPLAY statement. While the NOHDR option suppresses the headers of all columns, the notation '/' can be used to suppress the header for an individual column.

The notation is specified in the DISPLAY statement immediately before the name of the field for which the column header is to be suppressed.

Compare the following two examples:

```
DISPLAY NAME PERSONNEL-ID JOB-TITLE
```

In this case, the default column headers of all three fields will be displayed:

Page	1		97-04-19 17:37:27
	NAME	PERSONNEL ID	CURRENT POSITION
	-----	-----	-----
	ABELLAN	60008339	MAQUINISTA
	ACHIESON	30000231	DATA BASE ADMINISTRATOR
	ADAM	50005800	CHEF DE SERVICE
	ADKINSON	20008800	PROGRAMMER
	ADKINSON	20009800	DBA
	ADKINSON	20011000	SALES PERSON

```
DISPLAY '/' NAME PERSONNEL-ID JOB-TITLE
```

In this case, the notation '/' causes the column header for the field NAME to be suppressed:

Page	1		97-04-19 17:38:45
		PERSONNEL ID	CURRENT POSITION
		-----	-----
	ABELLAN	60008339	MAQUINISTA
	ACHIESON	30000231	DATA BASE ADMINISTRATOR
	ADAM	50005800	CHEF DE SERVICE
	ADKINSON	20008800	PROGRAMMER
	ADKINSON	20009800	DBA
	ADKINSON	20011000	SALES PERSON

Further Examples of Column Headers:

See programs DISPLX15 and DISPLX16 in library SYSEXPG.

Parameters to Influence the Output of Fields

Natural provides several parameters you can use to control the format in which fields are output:

- With the parameters LC, IC and TC, you can specify characters that are to be displayed before or after a field or before a field value.
- With the parameters AL and NL, you can increase or reduce the output length of fields.
- With the parameter SG, you can determine whether negative values are to be displayed with or without a minus sign.
- With the parameter IS, you can suppress the display of subsequent identical field values.
- With the parameter ZP, you can determine whether field values of "0" are to be displayed or not.
- With the parameter ES, you can suppress the display of empty lines generated by a DISPLAY or WRITE statement.

This is discussed in the following topics:

- Leading Characters - The LC Parameter
- Insertion Characters - The IC Parameter
- Trailing Characters - The TC Parameter
- Output Length - The AL and NL Parameters
- Sign Position - The SG Parameter
- Identical Suppress - The IS Parameter
- Zero Printing - The ZP Parameter
- Empty Line Suppression - The ES Parameter

Leading Characters - The LC Parameter

With the LC parameter, you can specify leading characters that are to be displayed immediately *before a field* that is output with a DISPLAY statement. The width of the output column is enlarged accordingly. You can specify 1 to 10 characters.

By default, values are displayed left-justified in alphanumeric fields and right-justified in numeric fields. (These defaults can be changed with the AD parameter; see the Natural Reference documentation). When a leading character is specified for an alphanumeric field, the character is therefore displayed immediately before the field value; for a numeric field, a number of spaces may occur between the leading character and the field value.

The LC parameter can be used with the following statements: FORMAT and DISPLAY. It can be set at statement level and at field level.

Insertion Characters - The IC Parameter

With the IC parameter, you specify the characters to be inserted in the column immediately *preceding the value of a field* that is output with a DISPLAY statement. You can specify 1 to 10 characters.

For a numeric field, the insertion characters will be placed immediately before the first significant digit that is output, with no intervening spaces between the specified character and the field value. For alphanumeric fields, the effect of the IC parameter is the same as that of the LC parameter.

The parameters LC and IC cannot both be applied to one field.

The IC parameter can be used with the following statements: FORMAT and DISPLAY. It can be set at statement level and at field level.

Trailing Characters - The TC Parameter

With the TC parameter, you can specify trailing characters that are to be displayed immediately *to the right of a field* that is output with a DISPLAY statement. The width of the output column is enlarged accordingly. You can specify 1 to 10 characters.

The TC parameter can be used with the following statements: FORMAT and DISPLAY. It can be set at statement level and at field level.

Output Length - The AL and NL Parameters

With the AL parameter, you can specify the *output length* for an alphanumeric field; with the NL parameter, you can specify the *output length* for a numeric field. This determines the length of a field as it will be output, which may be shorter or longer than the actual length of the field (as defined in the DDM for a database field, or in the DEFINE DATA statement for a user-defined variable).

Both parameters can be used with the following statements: FORMAT, DISPLAY, WRITE, and INPUT. They can be set at statement level and at field level.

Note:

If an edit mask is specified, it overrides an NL or AL specification. Edit masks are described later in this section.

Sign Position - The SG Parameter

With the SG parameter, you can determine whether or not a sign position is to be allocated for numeric fields.

- By default, SG=ON applies, which means that a sign position is allocated for numeric fields.
- If you specify SG=OFF, negative values in numeric fields will be output without a minus sign (-).

The SG parameter can be used with the following statements: FORMAT, DISPLAY, WRITE, and INPUT. It can be set at both statement level and field level.

Note:

If an edit mask is specified, it overrides an SG specification. Edit masks are described later in this section.

Example Program without Parameters:

```

** Example Program 'FORMAX03'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 NAME
    2 FIRST-NAME
    2 SALARY (1:1)
    2 BONUS (1:1,1:1)
  END-DEFINE
  READ (5) VIEWEMP BY NAME STARTING FROM 'JONES'
    DISPLAY NAME FIRST-NAME
      SALARY (1:1) BONUS (1:1,1:1)
  END-READ
  END

```

The above program contains no parameter settings and produces the following output:

Page	1			97-08-15 17:25:19
	NAME	FIRST-NAME	ANNUAL SALARY	BONUS

	JONES	VIRGINIA	46000	9000
	JONES	MARSHA	50000	0
	JONES	ROBERT	31000	0
	JONES	LILLY	24000	0
	JONES	EDWARD	37600	0

Example Program with Parameters AL, NL, LC, IC and TC:

```

** Example Program 'FORMAX04'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:1)
  2 BONUS (1:1,1:1)
END-DEFINE
FORMAT AL=10 NL=6
READ (5) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME (LC=*) FIRST-NAME (TC=*)
          SALARY (1:1)(IC=$) BONUS (1:1,1:1)(LC=>)
END-READ
END

```

The above program produces the following output. Compare the layout of this output with that of the previous program to see the effect of the individual parameters:

Page	1			97-08-19	17:26:12
NAME	FIRST-NAME	ANNUAL	BONUS		
		SALARY			

*JONES	VIRGINIA	* \$46000	> 9000		
*JONES	MARSHA	* \$50000	> 0		
*JONES	ROBERT	* \$31000	> 0		
*JONES	LILLY	* \$24000	> 0		
*JONES	EDWARD	* \$37600	> 0		

As you can see in the above example, any output length you specify with the AL or NL parameter does not include any characters specified with the LC, IC and TC parameters: the width of the NAME column, for example, is 11 characters - 10 for the field value (AL=10) plus 1 leading character.

The width of the SALARY and BONUS columns is 8 characters - 6 for the field value (NL=6), plus 1 leading/inserted character, plus 1 sign position (because SG=ON applies).

Identical Suppress - The IS Parameter

With the IS parameter, you can suppress the display of identical information in successive lines created by a WRITE or DISPLAY statement.

- By default, IS=OFF applies, which means that identical field values will be displayed.
- If IS=ON is specified, a value which is identical to the previous value of that field will not be displayed.

The IS parameter can be specified with a FORMAT statement to apply to the whole report, or it can be specified in a DISPLAY or WRITE statement at both statement level and field level.

The effect of the parameter IS=ON can be suspended for one record by using the statement SUSPEND IDENTICAL SUPPRESS; see the Natural Statements documentation for details.

Compare the output of the following two example programs to see the effect of the IS parameter. In the second one, the display of identical values in the NAME field is suppressed.

Example Program without IS Parameter:

```
** Example Program 'FORMAX05'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 NAME
    2 FIRST-NAME
  END-DEFINE
  READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
    DISPLAY NAME FIRST-NAME
  END-READ
  END
```

Page	1	97-08-18 17:25:19
	NAME	FIRST-NAME
	-----	-----
	JONES	VIRGINIA
	JONES	MARSHA
	JONES	ROBERT

Example Program with IS Parameter:

```
** Example Program 'FORMAX06'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 NAME
    2 FIRST-NAME
  END-DEFINE
  FORMAT IS=ON
  READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
    DISPLAY NAME FIRST-NAME
  END-READ
  END
```

Page	1	97-08-18	17:26:02
	NAME	FIRST-NAME	
	-----	-----	
	JONES	VIRGINIA	
		MARSHA	
		ROBERT	

Zero Printing - The ZP Parameter

With the ZP parameter, you determine how a field value of zero is to be displayed.

- By default, ZP=ON applies, which means that one "0" (for numeric fields) or all zeros (for time fields) will be displayed for each field value that is zero.
- If you specify ZP=OFF, the display of each field value which is zero will be suppressed.

The ZP parameter can be specified with a FORMAT statement to apply to the whole report, or it can be specified in a DISPLAY or WRITE statement at both statement level and field level.

Empty Line Suppression - The ES Parameter

With the ES parameter, you can suppress the output of empty lines created by a DISPLAY or WRITE statement.

- By default, ES=OFF applies, which means that lines containing all blank values will be displayed.
- If ES=ON is specified, a line resulting from a DISPLAY or WRITE statement which contains all blank values will not be displayed. This is particularly useful when displaying multiple-value fields or fields which are part of a periodic group if a large number of empty lines are likely to be produced.

The ES parameter can be specified with a FORMAT statement to apply to the whole report, or it can be specified in a DISPLAY or WRITE statement at statement level.

Note:

To achieve empty suppression for numeric values, in addition to ES=ON the parameter ZP=OFF must also be set for the fields concerned in order to have null values turned into blanks and thus not output either.

Compare the output of the following two example programs to see the effect of the parameters ZP and ES.

Example Program without Parameters ZP and ES:

```
** Example Program 'FORMAX07'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 NAME
    2 FIRST-NAME
    2 BONUS (1:2,1:1)
  END-DEFINE
  READ (4) VIEWEMP BY NAME STARTING FROM 'JONES'
    DISPLAY NAME FIRST-NAME BONUS (1:2,1:1)
  END-READ
  END
```

Page	1		97-08-18 17:26:19
	NAME	FIRST-NAME	BONUS
	-----	-----	-----
	JONES	VIRGINIA	9000
			6750
	JONES	MARSHA	0
			0
	JONES	ROBERT	0
			0
	JONES	LILLY	0
			0

Example Program with Parameters ZP and ES:

```

** Example Program 'FORMAX08'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BONUS (1:2,1:1)
END-DEFINE
FORMAT ES=ON
READ (4) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME FIRST-NAME BONUS (1:2,1:1)(ZP=OFF)
END-READ
END

```

Page	1		97-08-18 17:27:12
	NAME	FIRST-NAME	BONUS
	-----	-----	-----
	JONES	VIRGINIA	9000
			6750
	JONES	MARSHA	
	JONES	ROBERT	
	JONES	LILLY	

Further Examples of Parameters LC, IC, TC, AL, NL, IS, ZP and ES, and SUSPEND IDENTICAL SUPPRESS Statement:

See programs DISPLX17, DISPLX18, DISPLX19 SUSPEX01, SUSPEX02 and COMPRX03 in library SYSEXPG.

Edit Masks - The EM Parameter

With the EM parameter you can specify an *edit mask* for an alphanumeric or numeric field, that is, determine character by character the format in which the field values are to be output.

Example:

```
DISPLAY NAME (EM=X^X^X^X^X^X^X^X^X^X)
```

In this example, each "X" represents one character of an alphanumeric field value to be displayed, and each "^" represents a blank. If displayed via the above DISPLAY statement, the name "JOHNSON" would appear as follows:

```
J O H N S O N
```

You can specify the EM parameter at report level (in a FORMAT statement), at statement level (in a DISPLAY, WRITE, INPUT, MOVE EDITED or PRINT statement) or at field level (in a DISPLAY, WRITE or INPUT statement).

An edit mask specified with the EM parameter will override a default edit mask specified for a field in the DDM. If EM=OFF is specified, no edit mask at all will be used. An edit mask specified at statement level will override an edit mask specified at report level. An edit mask specified at field level will override an edit mask specified at statement level.

The following topics are covered below:

- Edit Masks for Numeric Fields
- Edit Masks for Alphanumeric Fields
- Length of Fields
- Edit Masks for Date and Time Fields
- Examples of Edit Masks

Edit Masks for Numeric Fields

Edit masks for numeric fields (formats N, I, P, F) must include a "9" for each output position you want filled with a number (even if it is zero). A "Z" is used to indicate that the output position will be filled only if the available number is not zero. A decimal point is indicated with a period "." To the right of the decimal point, a "Z" must not be specified. Leading, trailing, and insertion characters - for example, sign indicators - can be added.

Edit Masks for Alphanumeric Fields

Edit masks for alphanumeric fields must include an "X" for each alphanumeric character that is to be output. With a few exceptions, you may add leading, trailing and insertion characters (with or without enclosing them in apostrophes).

The character "^" is used to insert blanks in edit mask for both numeric and alphanumeric fields.

Length of Fields

It is important to be aware of the length of the field to which you assign an edit mask. If the edit mask is longer than the field, this will yield unexpected results. If the edit mask is shorter than the field, the field output will be truncated to just those positions specified in the edit mask.

Examples:

Assuming an alphanumeric field that is 12 characters long and the field value to be output is "JOHNSON", the following edit masks will yield the following results:

```
EM=X.X.X.X.X      Output:   J.O.H.N.S
EM=*****XXXXXX** Output:   *****JOHNSO**
```

Edit Masks for Date and Time Fields

Edit masks for date fields can include the characters "D" (day), "M" (month) and "Y" (year) in various combinations. Edit masks for time fields can include the characters "H" (hour), "I" (minute), "S" (second) and "T" (tenth of a second) in various combinations.

In conjunction with edit masks for date and time fields, see also the date and time system variables.

Examples of Edit Masks

Some examples of edit masks, along with possible output they produce, are provided below. In addition, the abbreviated notation for each edit mask is given. You can use either the abbreviated or the long notation.

Edit Mask	Abbreviation	Output A	Output B
EM=999.99	EM=9(3).9(2)	367.32	005.40
EM=ZZZZZ9	EM=Z(5)9(1)	0	579
EM=X^XXXXXX	EM=X(1)^X(5)	B LUE	A 19379
EM=XXX...XX	EM=X(3)...X(2)	BLU...E	AAB...01
EM=MM.DD.YY	*	01.05.87	12.22.86
EM=HH.II.SS.T	**	08.54.12.7	14.32.54.3

* Use a date system variable.

** Use a time system variable.

For further information about edit masks, see the session parameter EM in the Natural Reference documentation.

Example Program without EM Parameters:

```

** Example Program 'EDITMX01'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 SALARY (1:3)
  2 CITY
END-DEFINE
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY 'N A M E' NAME /
    'OCCUPATION' JOB-TITLE
    'SALARY' SALARY (1:3)
    'LOCATION' CITY

  SKIP 1
END-READ
END
    
```

The above program produces the following output which shows the default edit masks available:

Page	1	97-08-19	17:26:19
	N A M E	SALARY	LOCATION
	OCCUPATION		

	JONES	46000	TULSA
	MANAGER	42300	
		39300	
	JONES	50000	MOBILE
	DIRECTOR	46000	
		42700	
	JONES	31000	MILWAUKEE
	PROGRAMMER	29400	
		27600>	

Example Program with EM Parameters:

```

** Example Program 'EDITMX02'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 JOB-TITLE
  2 SALARY (1:3)
END-DEFINE
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY 'N A M E' NAME (EM=X^X^X^X^X^X^X^X^X^X^X^X^X^X^X^X) /
    FIRST-NAME (EM=...X(10)...)
    'OCCUPATION' JOB-TITLE (EM=' ____ 'X(12))
    'SALARY' SALARY (1:3) (EM=' USD 'ZZZ,999)

  SKIP 1
END-READ
END
    
```

The above program produces the following output. Compare the output with that of the previous program to see how the EM specifications affect the way the fields are displayed.

Page	1		97-08-19	17:26:29
	N A M E FIRST-NAME	OCCUPATION	SALARY	

J O N E S		___ MANAGER	USD	46,000
..VIRGINIA	...		USD	42,300
			USD	39,300
J O N E S		___ DIRECTOR	USD	50,000
..MARSHA	...		USD	46,000
			USD	42,700
J O N E S		___ PROGRAMMER	USD	31,000
..ROBERT	...		USD	29,400
			USD	27,600

Further Examples of Edit Masks:

See programs EDITMX03, EDITMX04 and EDITMX05 in library SYSEXP.

Vertical Displays

There are two ways of creating vertical displays:

- You can use a combination of the statements DISPLAY and WRITE.
- You can use the VERT option of the DISPLAY statement.

The following topics are covered below:

- Combining DISPLAY and WRITE
- The Tab Notation T*-field
- The Positioning Notation x/y
- The DISPLAY VERT Statement
- The Tab Notation P*-field

Combining DISPLAY and WRITE

As described earlier in this section, the DISPLAY statement normally presents the data in columns with default headers, while the WRITE statement presents data horizontally without headers.

You can combine the features of the two statements to produce vertical displays of field values.

The DISPLAY statement produces the values of different fields for the same record across the page with a column for each field. The field values for each record are displayed below the values for the previous record.

By using a WRITE statement after a DISPLAY statement, you can insert text and/or field values specified in the WRITE statement between records displayed via the DISPLAY statement.

The following program illustrates the combination of DISPLAY and WRITE:

```

** Example Program 'WRITEX04'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 NAME
    2 JOB-TITLE
    2 CITY
    2 DEPT
  END-DEFINE
  READ (3) VIEWEMP BY CITY STARTING FROM 'SAN FRANCISCO'
    DISPLAY NAME JOB-TITLE
    WRITE 20T 'DEPT:' DEPT
    SKIP 1
  END-READ
  END

```

It produces the following output:

Page	1	97-08-19	17:52:19
	NAME	CURRENT	POSITION

KOLENCE	MANAGER		
	DEPT: TECH05		
GOSDEN	ANALYST		
	DEPT: TECH10		
WALLACE	SALES PERSON		
	DEPT: SALE20		

Tab Notation T*field

In the previous example, the position of the field DEPT is determined by the tab notation *nT* (in this case "20T", which means that the display begins in column 20 on the screen).

Field values specified in a WRITE statement can be lined up automatically with field values specified in the first DISPLAY statement of the program by using the tab notation *T*field* (where *field* is the name of the field to which the field is to be aligned).

In the following program, the output produced by the WRITE statement is aligned to the field JOB-TITLE by using the notation "T*JOB-TITLE":

```

** Example Program 'WRITEX05'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 NAME
    2 JOB-TITLE
    2 DEPT
    2 CITY
  END-DEFINE
  READ (3) VIEWEMP BY CITY STARTING FROM 'SAN FRANCISCO'
  DISPLAY NAME JOB-TITLE
  WRITE T*JOB-TITLE 'DEPT:' DEPT
  SKIP 1
  END-READ
  END

```

Page	1	97-08-19 17:52:19
	NAME	CURRENT POSITION

	KOLENCE	MANAGER DEPT: TECH05
	GOSDEN	ANALYST DEPT: TECH10
	WALLACE	SALES PERSON DEPT: SALE20

Positioning Notation x/y

When you use the DISPLAY and WRITE statements in sequence and multiple lines are to be produced by the WRITE statement, you can use the notation x/y (number-slash-number) to determine in which row/column something is to be displayed. The positioning notation causes the next element in the DISPLAY or WRITE statement to be placed x lines below the last output, beginning in column y of the output.

The following program illustrates the use of this notation:

```

** Example Program 'WRITEX06'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 MIDDLE-I
  2 ADDRESS-LINE (1:1)
  2 CITY
  2 ZIP
END-DEFINE
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  DISPLAY 'NAME AND ADDRESS' NAME
  WRITE 1/5 FIRST-NAME          1/30 MIDDLE-I
        2/5 ADDRESS-LINE (1:1)
        3/5 CITY                3/30 ZIP /
END-READ
END

```

Page	1	97-08-19	17:55:47
NAME AND ADDRESS			

RUBIN			
SYLVIA		L	
2003 SARAZEN PLACE			
NEW YORK			10036
WALLACE			
MARY		P	
12248 LAUREL GLADE C			
NEW YORK			10036
KELLOGG			
HENRIETTA		S	
1001 JEFF RYAN DR.			
NEWARK			19711

DISPLAY VERT Statement

The standard display mode in Natural is horizontal. With the VERT clause option of the DISPLAY statement, you can override the standard display and produce a vertical field display. The HORIZ clause option, which can be used in the same DISPLAY statement, re-activates the standard horizontal display mode.

Column headings in vertical mode are controlled with various forms of the AS clause:

- Without AS clause, no column headings will be output.
- AS CAPTIONED causes default headings to be displayed.
- AS *text* causes the specified *text* to be displayed as column heading. Note that a slash (/) within the *text* element in a DISPLAY statement causes a line advance.
- AS *text* CAPTIONED causes the specified *text* to be displayed as column heading, and the default column headings to be displayed immediately before the field value in each line that is output.

The following example programs illustrate the use of the DISPLAY VERT statement.

DISPLAY VERT without AS Clause

The following program has no AS clause, which means that no column headings are output.

```
** Example Program 'DISPLX09'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 NAME
    2 FIRST-NAME
    2 CITY
  END-DEFINE
  READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
    DISPLAY VERT NAME FIRST-NAME / CITY
    SKIP 2
  END-READ
  END
```

Note that all field values are displayed vertically underneath one another:

Page	1	97-08-19	17:55:47
RUBIN			
SYLVIA			
NEW YORK			
WALLACE			
MARY			
NEW YORK			
KELLOGG			
HENRIETTA			
NEWARK			

DISPLAY VERT AS CAPTIONED and HORIZ

The following program contains a VERT and a HORIZ clause, which causes some column values to be output vertically and others horizontally; moreover AS CAPTIONED causes the default column headers to be displayed.

```

** Example Program 'DISPLX10'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 NAME
    2 FIRST-NAME
    2 CITY
    2 JOB-TITLE
    2 SALARY (1:1)
  END-DEFINE
  READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
    DISPLAY VERT AS CAPTIONED NAME FIRST-NAME
      HORIZ JOB-TITLE SALARY (1:1)
    SKIP 1
  END-READ
  END

```

NAME FIRST-NAME	CURRENT POSITION	ANNUAL SALARY
RUBIN SYLVIA	SECRETARY	17000
WALLACE MARY	ANALYST	38000
KELLOGG HENRIETTA	DIRECTOR	52000

Page 1

97-08-19 17:55:47

DISPLAY VERT AS *text*

The following program contains an AS *text* clause, which displays the specified *text* as column header.

```

** Example Program 'DISPLX11'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 NAME
    2 FIRST-NAME
    2 CITY
    2 JOB-TITLE
    2 SALARY (1:1)
  END-DEFINE
  READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
    DISPLAY VERT AS 'EMPLOYEES' NAME FIRST-NAME
      HORIZ JOB-TITLE SALARY (1:1)
    SKIP 1
  END-READ
  END
    
```

Page	1		97-08-19 7:55:47
	EMPLOYEES	CURRENT POSITION	ANNUAL SALARY
	-----	-----	-----
	RUBIN SYLVIA	SECRETARY	17000
	WALLACE MARY	ANALYST	38000
	KELLOGG HENRIETTA	DIRECTOR	52000

DISPLAY VERT AS *text* CAPTIONED

The following program contains an AS *text* CAPTIONED clause.

```

** Example Program 'DISPLX12'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 NAME
    2 FIRST-NAME
    2 CITY
    2 JOB-TITLE
    2 SALARY (1:1)
  END-DEFINE
  READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
    DISPLAY VERT AS 'EMPLOYEES' CAPTIONED NAME FIRST-NAME
      HORIZ JOB-TITLE SALARY (1:1)
    SKIP 1
  END-READ
  END

```

This clause causes the default column headers (NAME and FIRST-NAME) to be placed before the field values:

Page	1		97-04-19	17:55:47
	EMPLOYEES		CURRENT POSITION	ANNUAL SALARY

NAME RUBIN		SECRETARY		17000
FIRST-NAME SYLVIA				
NAME WALLACE		ANALYST		38000
FIRST-NAME MARY				
NAME KELLOGG		DIRECTOR		52000
FIRST-NAME HENRIETTA				

Tab Notation P*field

If you use a combination of DISPLAY VERT statement and subsequent WRITE statement, you can use the tab notation P*field in the WRITE statement to align the position of a field to the column *and* line position of a particular field specified in the DISPLAY VERT statement.

In the following program, the fields SALARY and BONUS are displayed in the same column, SALARY in every first line, BONUS in every second line.

The text "***SALARY PLUS BONUS***" is aligned to SALARY, which means that it is displayed in the same column as SALARY and in the first line, whereas the text "(IN US DOLLARS)" is aligned to BONUS and therefore displayed in the same column as BONUS and in the second line.

```

** Example Program 'WRITEX07'
  DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
    2 CITY
    2 NAME
    2 JOB-TITLE
    2 SALARY (1:1)
    2 BONUS (1:1,1:1)
  END-DEFINE
  READ (3) VIEWEMP BY CITY STARTING FROM 'LOS ANGELES'
    DISPLAY NAME JOB-TITLE VERT AS 'INCOME' SALARY (1) BONUS (1,1)
    WRITE P*SALARY '***SALARY PLUS BONUS***'
      P*BONUS '(IN US DOLLARS)'
    SKIP 1
  END-READ
  END

```

Page	1	97-08-19 18:14:11	
	NAME	CURRENT POSITION	INCOME
	-----	-----	-----
	POORE JR	SECRETARY	25000 0 ***SALARY PLUS BONUS*** (IN US DOLLARS)
	PREPARATA	MANAGER	46000 9000 ***SALARY PLUS BONUS*** (IN US DOLLARS)
	MARKUSH	TRAINEE	22000 0 ***SALARY PLUS BONUS*** (IN US DOLLARS)

Further Example of DISPLAY VERT with WRITE Statement:

See program WRITEX10 in library SYSEXPG.

Object Types

This section covers the following topics:

- What Types of Programming Objects Are There?
 - Data Areas
 - Programs, Subprograms and Subroutines
 - Maps
 - Helproutines
 - Multiple Use of Source Code - Copycode
 - Documenting Natural Objects - Text
 - Creating Event Driven Applications - Dialog
 - Creating Component Based Applications - Class
 - Using Non-Natural Files - Resource
-

What Types of Programming Objects Are There?

Within a Natural application, several types of programming objects can be used to achieve an efficient application structure.

There are the following types of Natural programming objects:

- Local Data Area
- Global Data Area
- Parameter Data Area
- Program
- Subprogram
- Subroutine
- Helproutine
- Map
- Copycode
- Text
- Dialog
- Class

To create and maintain all these objects, you use the Natural editors:

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

The editors are described in your Natural User's Guide.

Data Areas

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

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

Natural supports three types of data areas:

- **Local Data Area**
In a local data area, you define the data elements that are to be used by a single Natural module in an application.
- **Global Data Area**
In a global data area, you define the data elements that are to be used by more than one Natural program, routine, etc. in an application.
- **Parameter Data Area**
In a parameter data area, you define the fields that are passed as parameters to a subprogram, external subroutine or help routine.

Local Data Area

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

- You can define the data within the program.
- You can define the data in a local data area outside the program.

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

Example 1 - Fields Defined within a DEFINE DATA Statement:

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

Example 2 - Fields Defined in a Separate Data Area:

Program:

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

Local Data Area "LDA39":

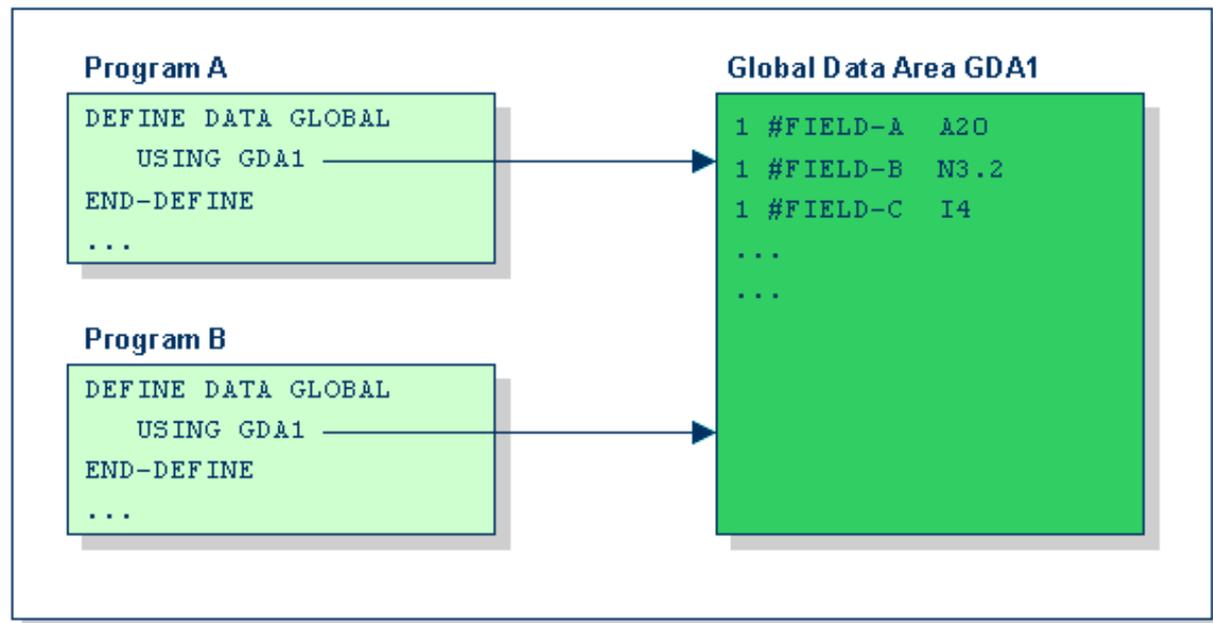
I	T	L	Name	F	Leng	Index/Init/EM/Name/Comment
V	1		VIEWEMP			EMPLOYEES
	2		NAME	A	20	
	2		FIRST-NAME	A	20	
	2		PERSONNEL-ID	A	8	
	1		#VARI-A	A	20	
	1		#VARI-B	N	3.2	
	1		#VARI-C	I	4	

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

Global Data Area

In a global data area, you define the data elements that are to be used by more than one program, routine, etc. in an application.

Variables defined in a global data area may be referenced by several objects in an application.



The global data area and the objects which reference it must be in the same library (or a steplib).

Global data areas must be defined with the data area editor, and a program using that data area must reference it in the DEFINE DATA statement. Any number of main programs, external subroutines and help routines can share the same global data area.

Each object can reference only one global data area; that is, a DEFINE DATA statement must not contain more than one GLOBAL clause.

Note:

When you build an application where multiple objects share a global data area, remember that modifications to a global data area affect all programs or routines that reference that data area. Therefore these objects must be STOWed again after the global data area has been modified.

When are Global Data Areas Initialized?

A global data area is initialized when it is used for the first time. It remains active in the current Natural session (that is, the variables in the global data area retain their contents) until:

- the next LOGON, or
- another global data area is used on the same level (levels are described later in this section), or
- a RELEASE VARIABLES statement is executed. In this case, the variables in the global data area are reset when either the execution of the level 1 program is finished, or the program invokes another program via a FETCH or RUN statement.

Note:

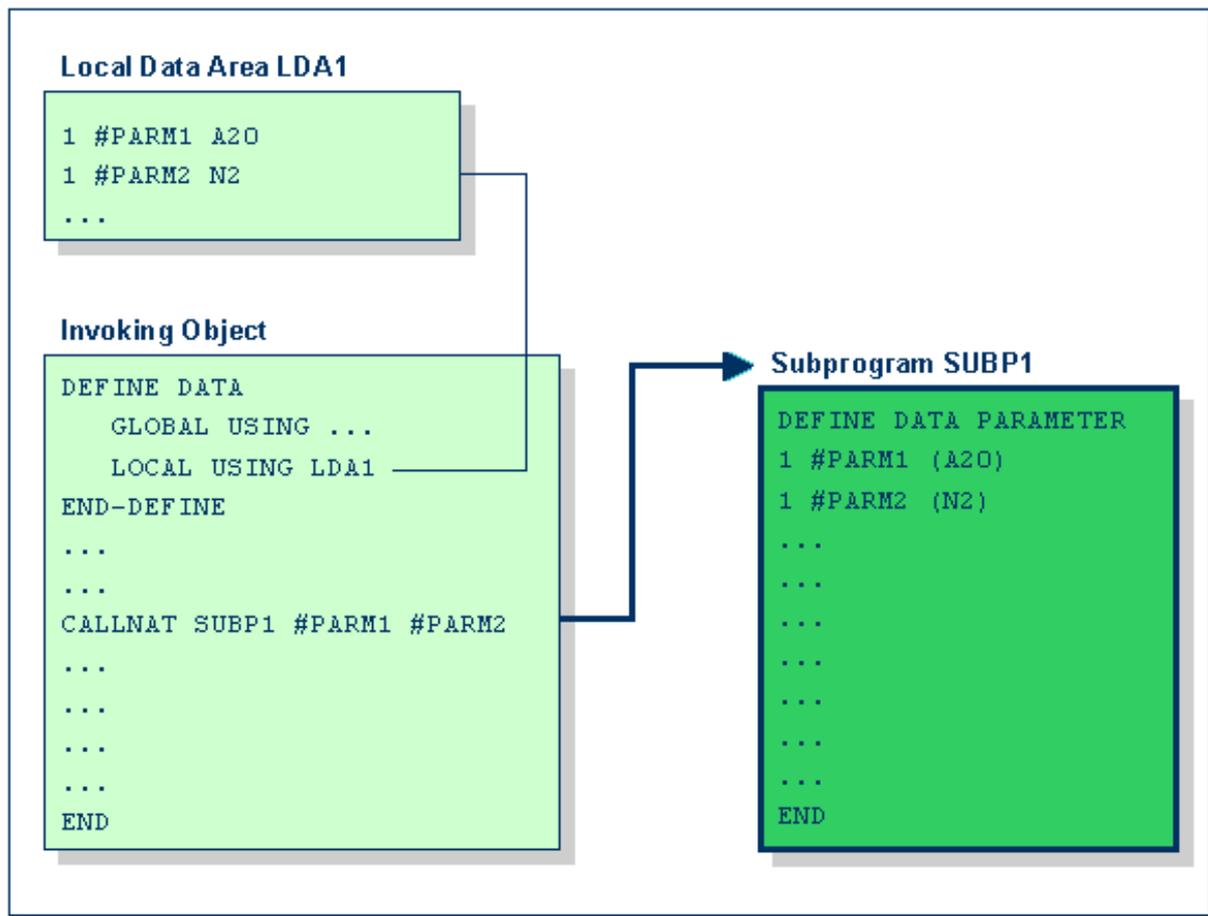
If a GDA named "COMMON" exists in a library, the program named ACOMMON is invoked automatically when you LOGON to that library.

Parameter Data Area

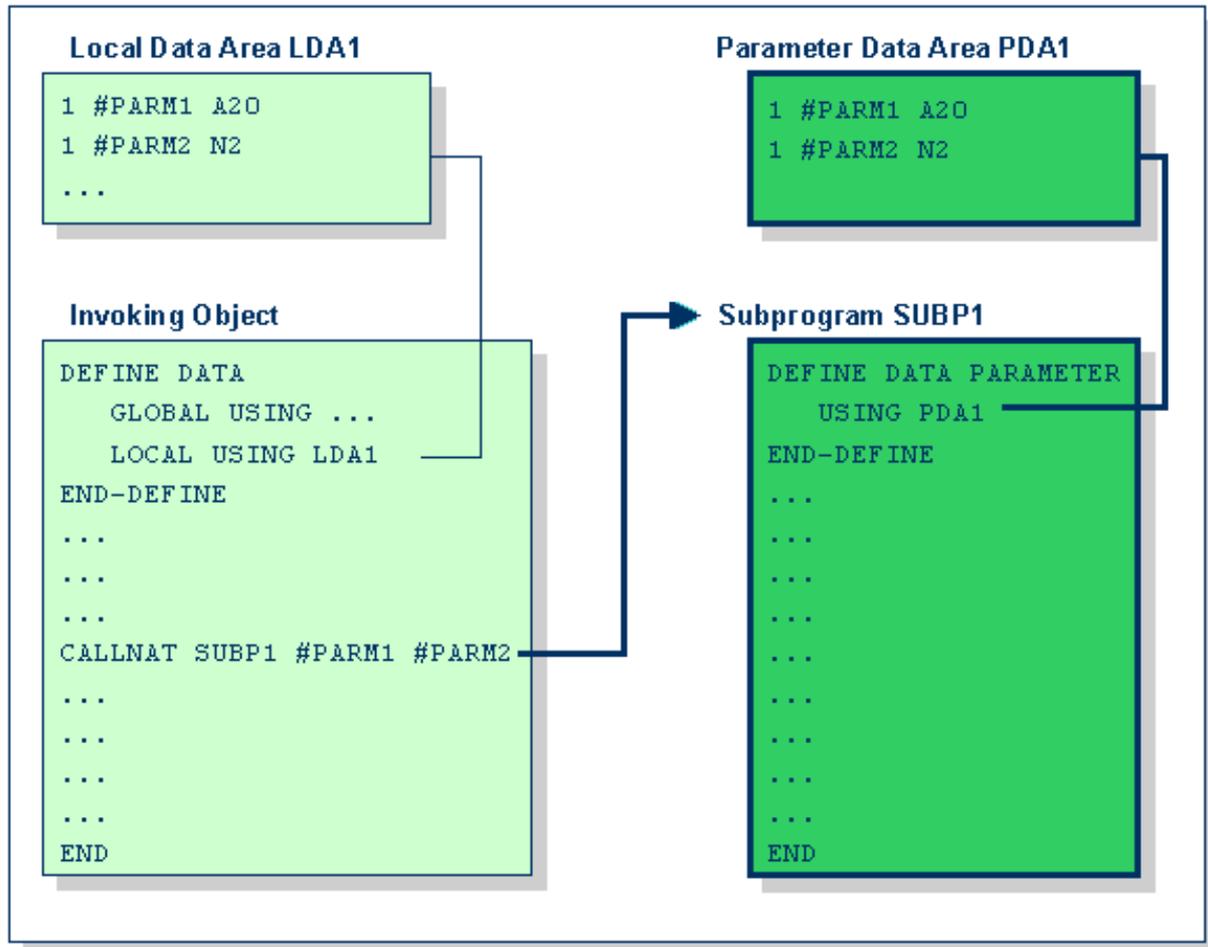
Parameter data areas are used by subprograms and external subroutines.

A subprogram is invoked with a CALLNAT statement. With the CALLNAT statement, parameters can be passed from the invoking object to the subprogram. These parameters must be defined with a DEFINE DATA PARAMETER statement in the subprogram: they can be defined in the PARAMETER clause of the DEFINE DATA statement itself; or they can be defined in a separate parameter data area, with the DEFINE DATA PARAMETER statement referencing that parameter data area.

Parameter Defined Within DEFINE DATA PARAMETER Statement:



Parameter Defined in Parameter Data Area:



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

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

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

Programs, Subprograms and Subroutines

The following topics are covered below:

- A Modular Application Structure
- Multiple Levels of Invoked Objects
- Program
- Subroutine
- Subprogram
- Processing Flow when Invoking a Routine

A Modular Application Structure

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

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

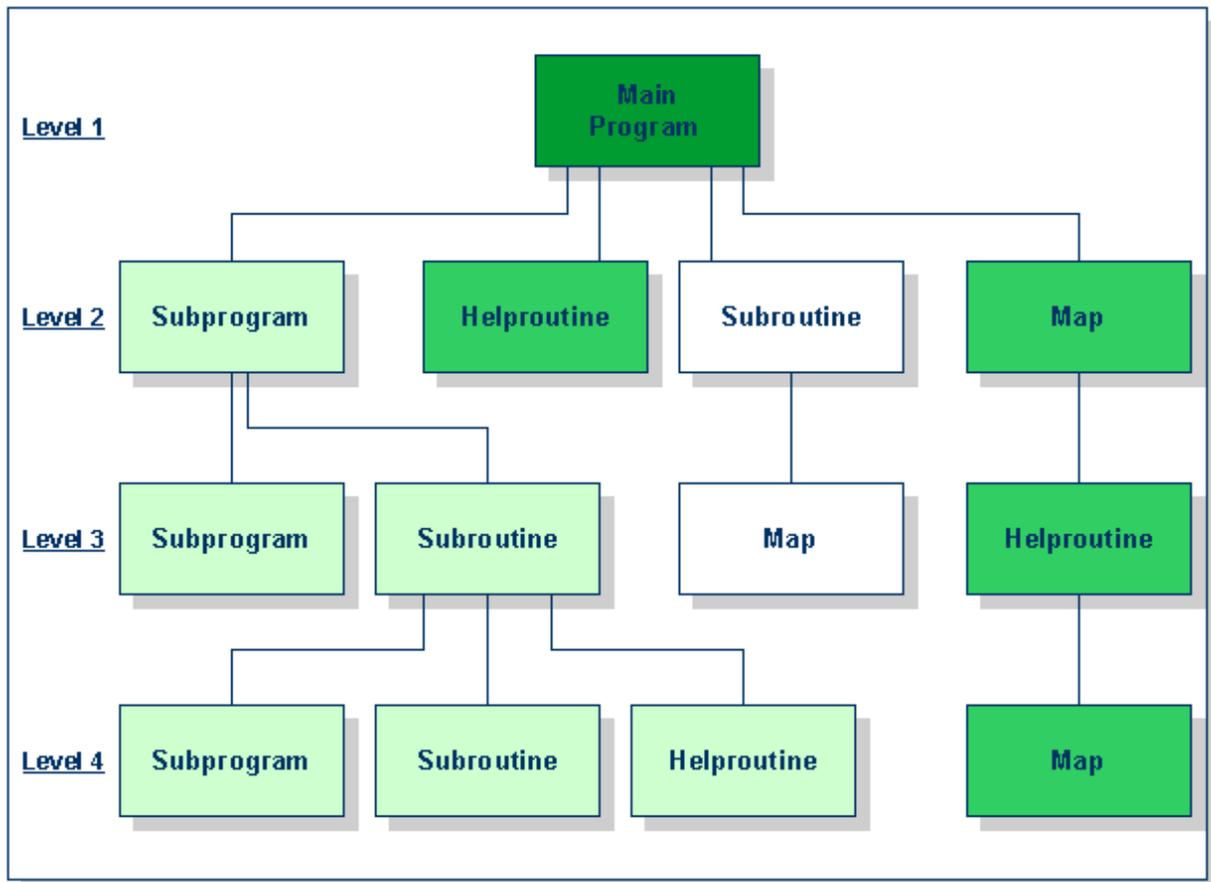
Multiple Levels of Invoked Objects

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

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

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

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



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

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

- program
- subroutine
- subprogram

Help routines and maps, although they are also invoked from other objects, are strictly speaking not routines as such, and are therefore discussed in later sections of this section.

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

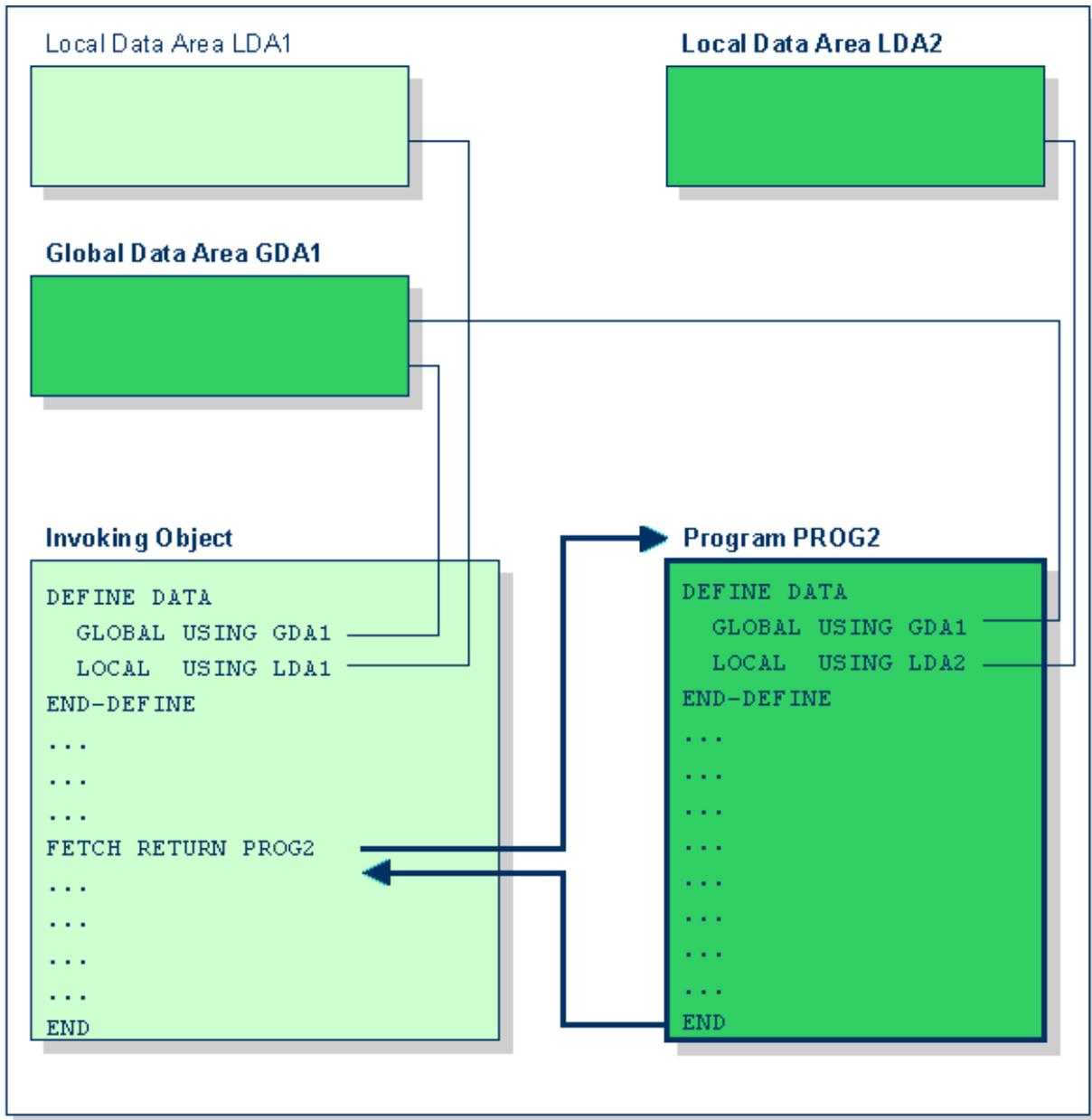
Program

A program can be executed - and thus tested - by itself. To compile and execute a source program, you use the system command RUN. To execute a program that already exists in compiled form, you use the system command EXECUTE.

A program can also be invoked from another object with a FETCH or FETCH RETURN statement. The invoking object can be a program, subprogram, subroutine or help routine.

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

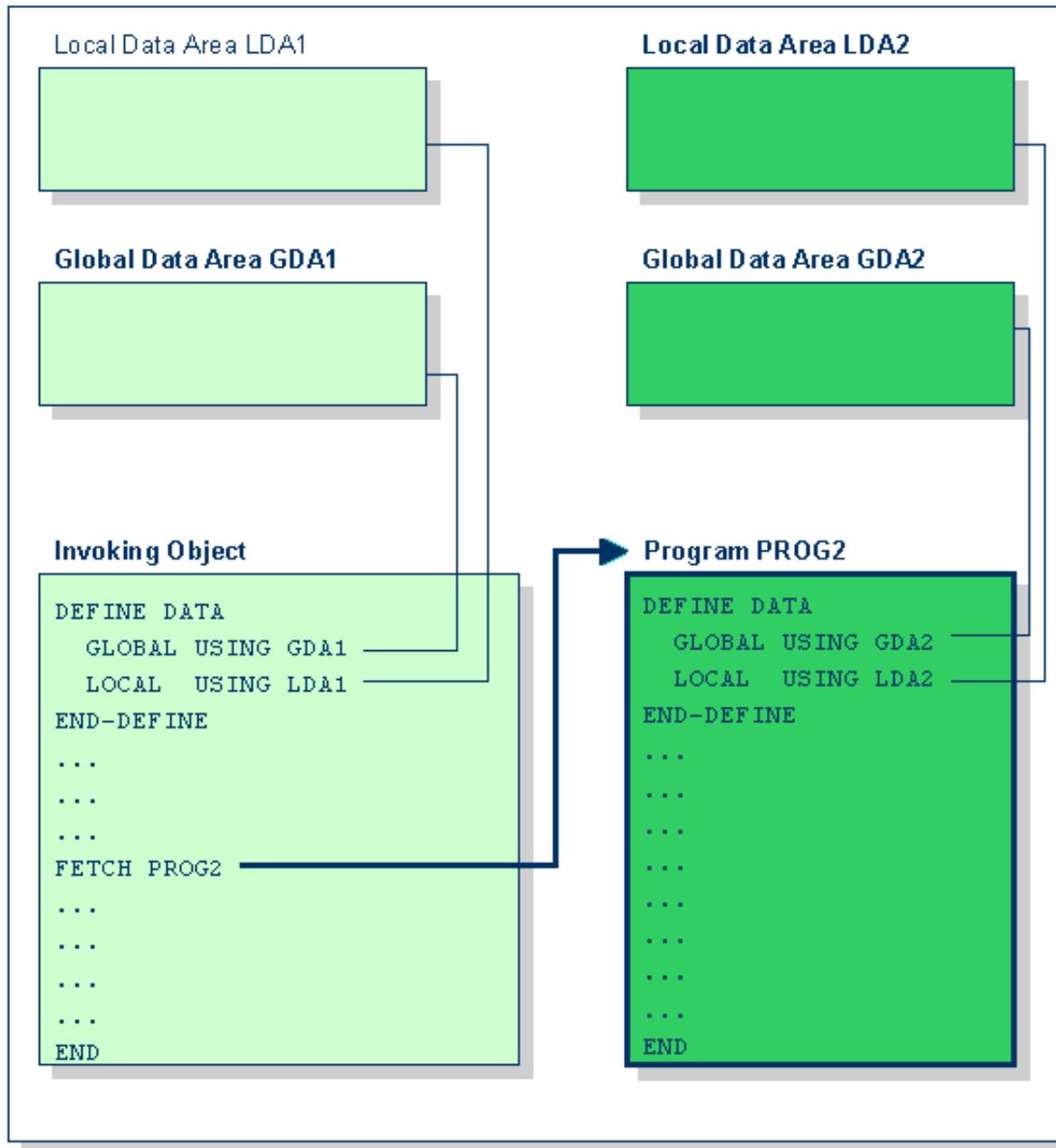
Program Invoked with FETCH RETURN:



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

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

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

Program Invoked with FETCH:

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

Note:

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

Subroutine

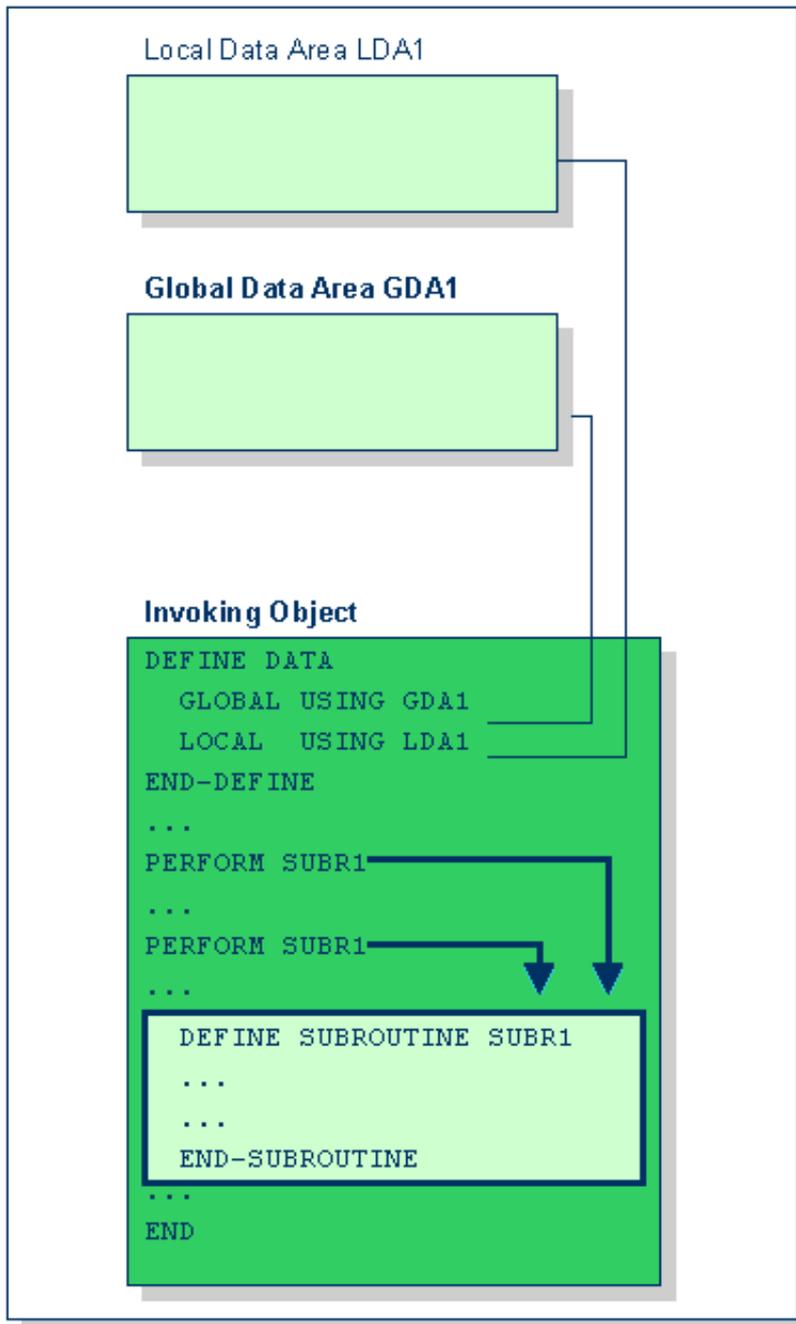
The statements that make up a subroutine must be defined within a DEFINE SUBROUTINE ... END-SUBROUTINE statement block.

A subroutine is invoked with a PERFORM statement.

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

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

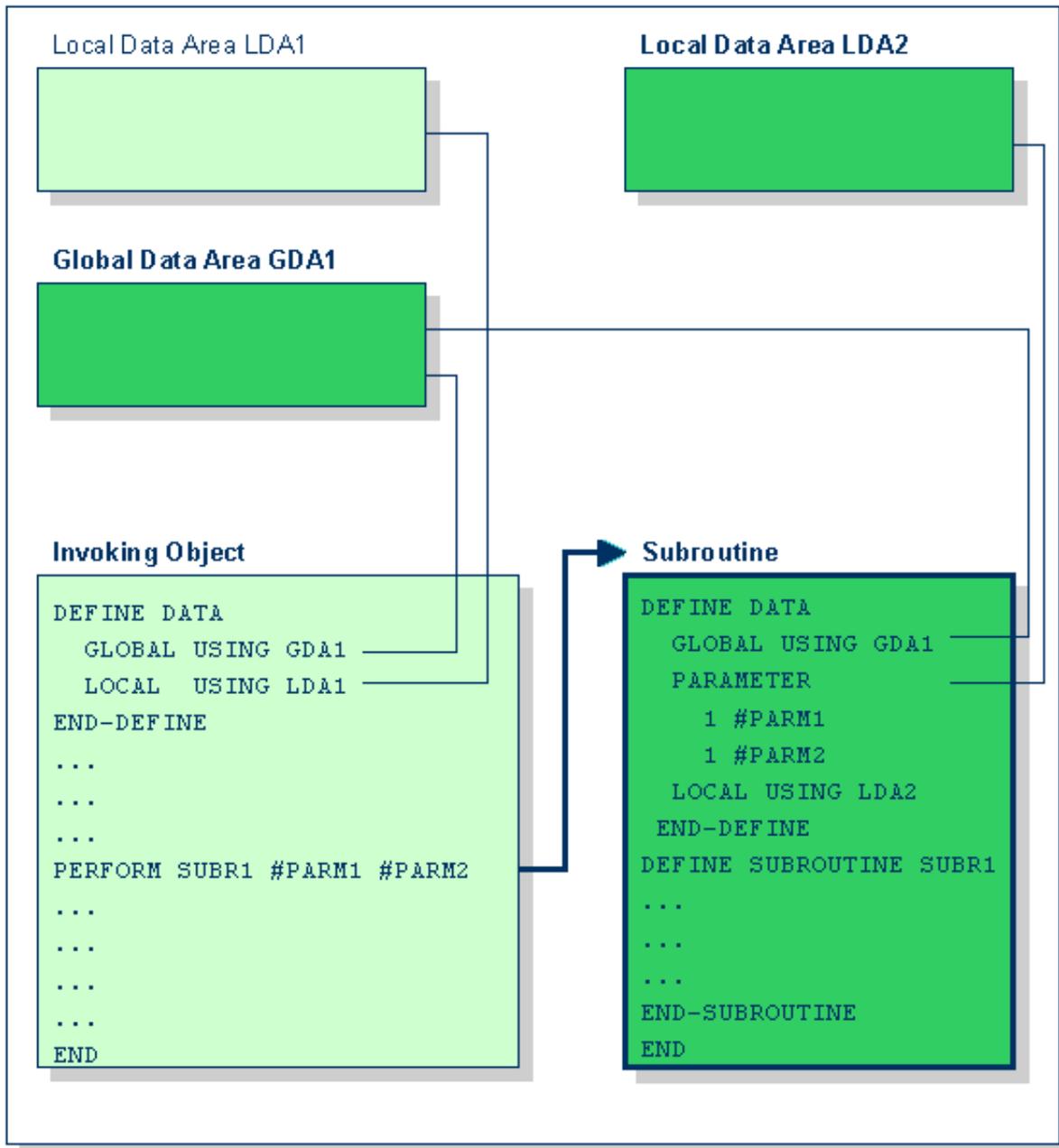
If you have a block of code which is to be executed several times within an object, it is useful to use an inline subroutine. You then only have to code this block once within a DEFINE SUBROUTINE statement block and invoke it with several PERFORM statements.

Inline Subroutine:

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

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

External Subroutine:



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

Data Available to an Inline Subroutine

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

Data Available to an External Subroutine

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

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

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

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

Subprogram

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

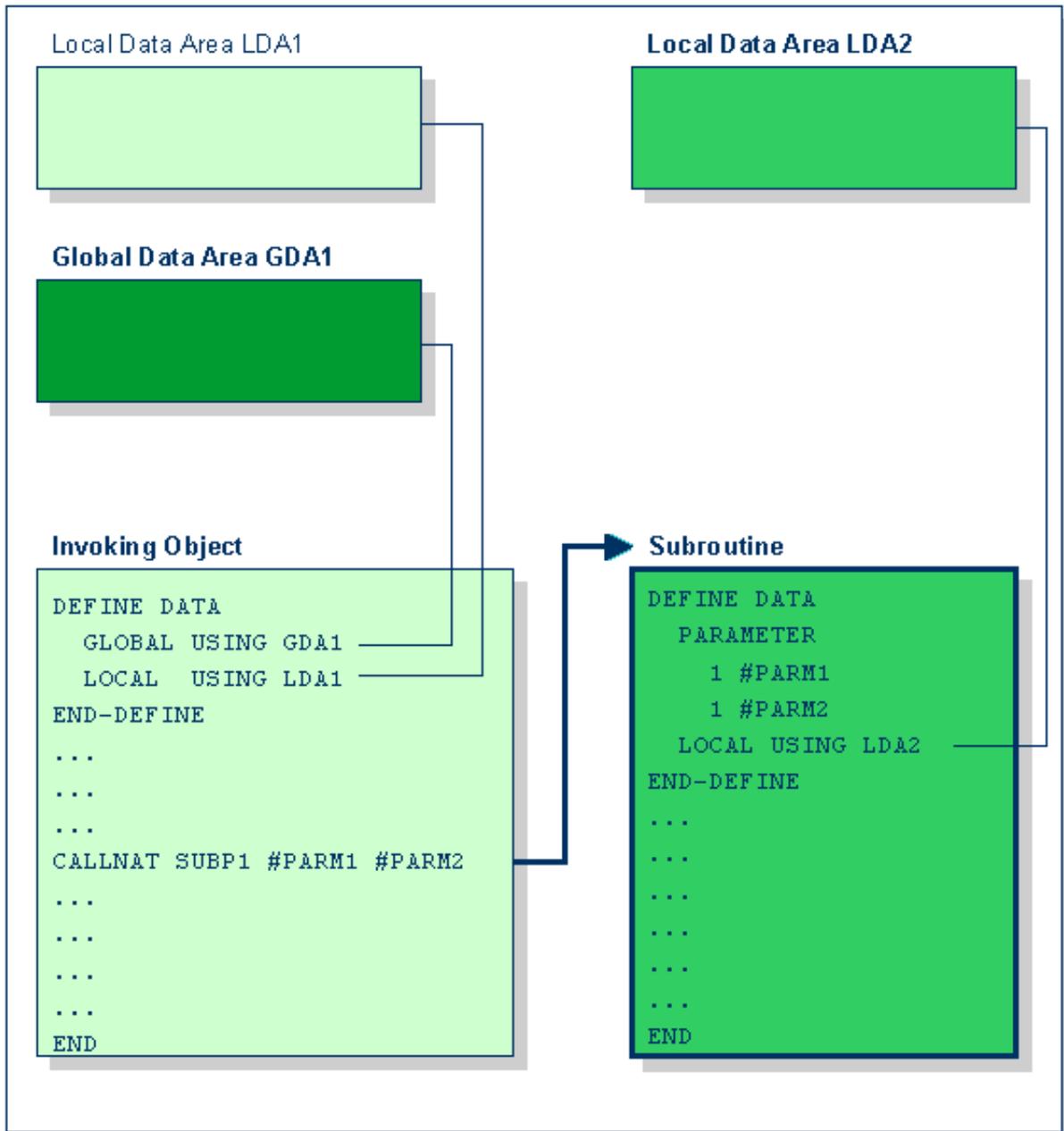
A subprogram cannot be executed by itself. It must be invoked from another object. The invoking object can be a program, subprogram, subroutine or help routine.

A subprogram is invoked with a `CALLNAT` statement.

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

Data Available to a Subprogram

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



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

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

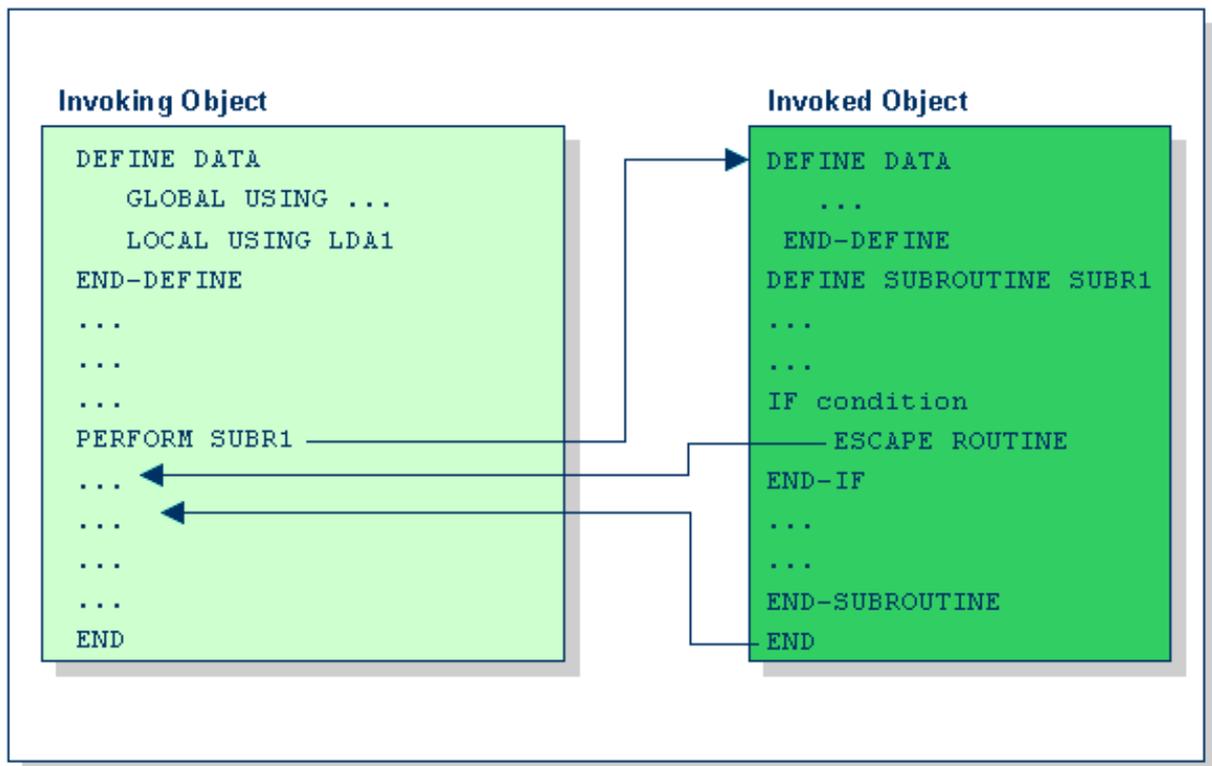
Processing Flow when Invoking a Routine

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

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

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

Example:



Maps

Maps are those parts of an application which the users see on their screens.

The dialogue with the user is done via input maps. An *input map* is invoked with an INPUT USING MAP statement.

If an application produces any output report, this report can be displayed on the screen by using an *output map*. An output map is invoked with a WRITE USING MAP statement.

Maps are created with the map editor, which is described in your Natural User's Guide.

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

Help maps are, in principle, like any other maps, but when they are assigned as help, additional checks are performed to ensure their usability for help purpose. Help maps are created with the map editor.

Help routines

Help routines have specific characteristics to facilitate the processing of help requests.

Help routines are created with the program editor. They may be used to implement complex and interactive help systems.

The following topics are covered below:

- Invoking Help
- Specifying Help routines
- Programming Considerations for Help routines
- Passing Parameters to Help routines
- Help as a Window

Invoking Help

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

Note 1:

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

Note 2:

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

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

```
SET KEY PF1=HELP
```

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

Specifying Helproutines

A helproutine may be specified:

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

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

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

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

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

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

or using the extending field editing facility of the map editor (as described in your Natural User's Guide).

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

Programming Considerations for Help routines

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

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

Passing Parameters to Help routines

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

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

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

The implicit parameter is the field for which the help routine was invoked:

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

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

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

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

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

Note:

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

Once help processing is completed, the screen data will be refreshed: any fields which have been modified by the help routine will be updated - excluding fields which had been modified by the user before the help routine was invoked, but including the field for which help was requested.

(Exception: If the field for which help was requested is split into several parts by dynamic attributes (DY parameter), and the part in which the question mark is entered is *after* a part modified by the user, the field content will not be modified by the help routine.)

Note:

Control variables are not evaluated again after the processing of the help routine, even if they have been modified within the help routine.

Equal Sign Option

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

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

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

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

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

Array Indices

If the field selected by the help character or the help key is an array element, its indices are supplied as implicit parameters (1 - 3 depending on rank, regardless of the explicit parameters). The format/length of these parameters is I2.

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

The corresponding helproutine starts with:

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

```

Help as a Window

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

```

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

```

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

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

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

For further information on window processing, please refer to the DEFINE WINDOW statement in the Natural Statements documentation and the terminal command %W in the Natural Reference documentation.

Multiple Use of Source Code - Copycode

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

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

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

Consequently, when you modify the source code of copycode, you also have to newly compile (STOW) all objects which use that copycode.

Copycode cannot be executed on its own. It cannot be STOWed, but only SAVED.

For further information on copycode, please refer to the description of the INCLUDE statement in the Natural Statements documentation.

Documenting Natural Objects - Text

The Natural object type "text" is used to write text rather than programs. You can write any text you wish (there is no syntax check). You can use this type of object to document Natural objects in more detail than you can, for example, within the source code of a program. "Text" objects may also be useful at sites where Predict is not available for program documentation purposes.

You write the text using the Natural program editor. The only difference in handling as opposed to writing programs, is that the text you write stays as it is, that is, there is no lower to upper case translation or empty line suppression (provided in your editor profile Empty Line Suppression is set to "N" and Editing in Lower Case is set to "Y", see your Natural User's Guide for Windows for more details).

"Text" objects can only be SAVED, they cannot be STOWed. They cannot be RUN, only displayed in the editor.

Creating Event Driven Applications - Dialog

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

For information on dialogs and event-driven programming, please refer to the Natural User's Guide for Windows.

Creating Component Based Applications - Class

Classes are used in conjunction with NaturalX when creating component based applications to be used in a client/server environment.

For information on classes, please refer to the NaturalX documentation.

Using Non-Natural Files - Resource

Resources are only available with Natural under Windows 98 and Windows NT/2000.

Natural distinguishes two kinds of resources:

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

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

Shared Resources

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

Example - Using a shared resource:

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

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

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

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

Note:

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

Private Resources

Private resources are used internally by Natural to store binary data that is part of Natural objects. These files are recognized by the file name extension `NR*`, where `*` is a character that depends on the type of the Natural object. Natural maintains private resource files and their contents automatically. A Natural object can have a maximum of one private resource file. Currently, only Natural dialogs have a private resource file. This file is used to store the configuration of ActiveX controls that are defined in a dialog and are configured with their own property pages. See ActiveX Control Property Pages on how to configure an ActiveX control.

Example - Private resources:

The name of the private resource file of the dialog `MYDLG` is `MYDLG.NR3`. Natural creates, modifies and deletes this file automatically as needed, when the dialog is created, modified, deleted etc. The private resource file is used to store binary data related to the dialog `MYDLG`.

Further Programming Aspects

This section covers the following topics:

- End of Program - The END Statement
 - End of Application - The STOP Statement
 - Conditional Processing - The IF Statement
 - Loop Processing
 - Control Breaks
 - Data Computation
 - System Variables and System Functions
 - Stack
 - Processing of Date Information
-

End of Program - The END Statement

The END statement is used to mark the end of a Natural program, subprogram, external subroutine or help routine.

Every one of these objects must contain an END statement as the last statement.

Every object may contain only one END statement.

End of Application - The STOP Statement

The STOP statement is used to terminate the execution of a Natural application. A STOP statement executed anywhere within an application immediately stops the execution of the entire application.

Conditional Processing - The IF Statement

With the IF statement, you define a logical condition, and the execution of the statement attached to the IF statement then depends on that condition.

The IF statement contains three components: IF, THEN, and ELSE.

- In the IF clause, you specify the logical condition which is to be met.
- In the THEN clause you specify the statement(s) to be executed if this condition is met.
- In the (optional) ELSE clause, you can specify the statement(s) to be executed if this condition is *not* met.

So, an IF statement takes the following general form:

```
IF condition
  THEN execute statement(s)
  ELSE execute other statement(s)
END-IF
```

If you wish a certain processing to be performed only if the IF condition is *not* met, you can specify the clause THEN IGNORE, which means that the IF condition will be ignored if it is met.

For more information on logical conditions, see General Information of the Natural Reference documentation.

Example of IF Statement:

```

** Example Program 'IFX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 BIRTH
  2 CITY
  2 SALARY (1:1)
END-DEFINE
*
LIMIT 7
READ MYVIEW BY CITY STARTING FROM 'C'
  IF SALARY (1) LT 40000 THEN
    WRITE NOTITLE '*****' NAME 30X 'SALARY LT 40000'
  ELSE
    DISPLAY NAME BIRTH (EM=YYYY-MM-DD) SALARY (1)
  END-IF
END-READ
END

```

The IF statement block in the above program causes the following conditional processing to be performed:

- IF the salary is less than 40000, THEN the WRITE statement is to be executed;
- otherwise (ELSE), that is, if the salary is 40000 or more, the DISPLAY statement is to be executed.

The program produces the following output:

NAME	DATE OF BIRTH	ANNUAL SALARY
***** KEEN		SALARY LT 40000
***** FORRESTER		SALARY LT 40000
***** JONES		SALARY LT 40000
***** MELKANOFF		SALARY LT 40000
DAVENPORT	1948-12-25	42000
GEORGES	1949-10-26	182800
***** FULLERTON		SALARY LT 40000

Nested IF Statements

It is possible to use various nested IF statements; for example, you can make the execution of a THEN clause dependent on another IF statement which you specify in the THEN clause.

Example of Nested IF Statements:

```

** Example Program 'IFX02'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 SALARY (1:1)
  2 BIRTH
  2 PERSONNEL-ID
1 MYVIEW2 VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
1 #BIRTH (D)
END-DEFINE
*
MOVE EDITED '19450101' TO #BIRTH (EN=YYYYMMDD)
*
LIMIT 20
FND1. FIND MYVIEW WITH CITY = 'BOSTON'
      SORTED BY NAME
  IF SALARY (1) LESS THAN 20000
    THEN WRITE NOTITLE '*****' NAME 30X 'SALARY LT 20000'
  ELSE
    IF BIRTH GT #BIRTH
      FIND MYVIEW2 WITH PERSONNEL-ID = PERSONNEL-ID (FND1.)
        DISPLAY (IS=ON) NAME BIRTH (EM=YYYY-MM-DD)
          SALARY (1) MAKE (AL=8 IS=OFF)
    END-FIND
  END-IF
END-IF
SKIP 1
END-FIND
END
    
```

The above program with nested IF statements produces the following output:

NAME	DATE OF BIRTH	ANNUAL SALARY	MAKE
***** COHEN			SALARY LT 20000
CREMER	1972-12-14	20000	FORD
***** FLEMING			SALARY LT 20000
***** GREENACRE			SALARY LT 20000
PERREAULT	1950-05-12	30500	CHRYSLER
***** SHAW			SALARY LT 20000
STANWOOD	1946-09-08	31000	CHRYSLER FORD

Further Example of IF Statement:

See program IFX03 in library SYSEXPG.

Loop Processing

A processing loop is a group of statements which are executed repeatedly until a stated condition has been satisfied, or as long as a certain condition prevails.

Processing loops can be subdivided into database loops and non-database loops:

- *Database processing loops* are those created automatically by Natural to process data selected from a database as a result of a READ, FIND or HISTOGRAM statement. These statements are described in the section Database Access.
- *Non-database processing loops* are initiated by the statements REPEAT, FOR, CALL FILE, CALL LOOP, SORT, and READ WORK FILE.

More than one processing loop may be active at the same time. Loops may be embedded or nested within other loops which remain active (open).

A processing loop must be explicitly closed with a corresponding END-... statement (for example, END-REPEAT, END-FOR, etc.)

The SORT statement, which invokes the sort program of the operating system, closes all active processing loops and initiates a new processing loop.

The following topics are covered below:

- Limiting Database Loops
- Limiting Non-Database Loops - The REPEAT Statement
- Terminating a Processing Loop - The ESCAPE Statement
- Loops Within Loops
- Referencing Statements within a Program

Limiting Database Loops

With the statements READ, FIND, or HISTOGRAM, you have three ways of limiting the number of repetitions of the processing loops initiated with these statements:

- with the session parameter LT,
- with a LIMIT statement,
- or with a limit notation in a READ/FIND/HISTOGRAM statement itself.

LT Session Parameter

With the system command GLOBALS, you can specify the session parameter LT, which limits the number of records which may be read in a database processing loop.

Example:

```
GLOBALS LT=100
```

This limit applies to all READ, FIND and HISTOGRAM statements in the entire session.

LIMIT Statement

In a program, you can use the LIMIT statement to limit the number of records which may be read in a database processing loop.

Example:

```
LIMIT 100
```

The LIMIT statement applies to the remainder of the program unless it is overridden by another LIMIT statement or limit notation.

Limit Notation

With a READ, FIND or HISTOGRAM statement itself, you can specify the number of records to be read in parentheses immediately after the statement name.

Example:

```
READ (10) VIEWXYZ BY NAME
```

This limit notation overrides any other limit in effect, but applies only for the statement in which it is specified.

If the limit set with the LT parameter is smaller than a limit specified with a LIMIT statement or a limit notation, the LT limit has priority over any of these other limits.

Limiting Non-Database Loops - The REPEAT Statement

Non-database processing loops begin and end based on logical condition criteria or some other specified limiting condition.

The REPEAT statement is discussed here as representative of a non-database loop statement.

With the REPEAT statement, you specify one or more statements which are to be executed repeatedly. Moreover, you can specify a logical condition, so that the statements are only executed either until or as long as that condition is met. For this purpose you use an UNTIL or WHILE clause:

- If you specify the logical condition in an UNTIL clause, the REPEAT loop will continue *until* the logical condition is met.
- If you specify the logical condition in a WHILE clause, the REPEAT loop will continue *as long as* the logical condition remains true.

If you specify no logical condition, the REPEAT loop must be exited with an ESCAPE, STOP or TERMINATE statement:

- An ESCAPE statement (see next section) terminates the execution of the processing loop and continues processing outside the loop.
- A STOP statement stops the execution of the entire Natural application.
- A TERMINATE statement stops the execution of the Natural application and also ends the Natural session.

Example of REPEAT Statement:

```

** Example Program 'REPEAX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 SALARY (1:1)
1 #PAY1 (N8)
END-DEFINE
*
READ (5) MYVIEW BY NAME WHERE SALARY (1) = 30000 THRU 39999
  MOVE SALARY (1) TO #PAY1
  REPEAT WHILE #PAY1 LT 40000
    MULTIPLY #PAY1 BY 1.1
    DISPLAY NAME (IS=ON) SALARY (1)(IS=ON) #PAY1
  END-REPEAT
  SKIP 1
END-READ
END
    
```

The above program produces the following output:

Page	1	97-08-19 18:42:53	
	NAME	ANNUAL SALARY	#PAY1
	-----	-----	-----
	ADKINSON	34500	37950 41745
		33500	36850 40535
		36000	39600 43560
	AFANASSIEV	37000	40700
	ALEXANDER	34500	37950 41745

Terminating a Processing Loop - The ESCAPE Statement

The ESCAPE statement is used to terminate the execution of a processing loop based on a logical condition.

You can place an ESCAPE statement within loops in conditional IF statement groups, in break processing statement groups (AT END OF DATA, AT END OF PAGE, AT BREAK), or as a stand-alone statement implementing the basic logical conditions of a non-database loop.

The ESCAPE statement offers the options TOP and bottom, which determine where processing is to continue after the processing loop has been left via the ESCAPE statement:

- ESCAPE TOP is used to continue processing at the top of the processing loop.
- ESCAPE bottom is used to continue processing with the first statement following the processing loop.

You can specify several ESCAPE statements within the same processing loop.

For further details and examples of the ESCAPE statement, see the Natural Statements documentation.

Loops Within Loops

A database statement can be placed within a database processing loop initiated by another database statement. When database loop-initiating statements are embedded in this way, a "hierarchy" of loops is created, each of which is processed for each record which meets the selection criteria.

Multiple levels of loops can be embedded. For example, non-database loops can be nested one inside the other. Database loops can be nested inside non-database loops. Database and non-database loops can be nested within conditional statement groups.

Example of Nested FIND Statements:

The following program illustrates a hierarchy of two loops, with one FIND loop nested or embedded within another FIND loop.

```

** Example Program 'FINDX06'
  DEFINE DATA LOCAL
  1 EMPLOY-VIEW VIEW OF EMPLOYEES
    2 CITY
    2 NAME
    2 PERSONNEL-ID
  1 VEH-VIEW VIEW OF VEHICLES
    2 MAKE
    2 PERSONNEL-ID
  END-DEFINE
  *
  FND1. FIND EMPLOY-VIEW WITH CITY = 'NEW YORK' OR = 'BEVERLEY HILLS'
    FIND (1) VEH-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (FND1.)
    DISPLAY NOTITLE NAME CITY MAKE
  END-FIND
  END-FIND
  END

```

The above program selects data from multiple files. The outer FIND loop selects from the EMPLOYEES file all persons who live in New York or Beverley Hills. For each record selected in the outer loop, the inner FIND loop is entered, selecting the car data of those persons from the VEHICLES file. The program produces the following output:

NAME	CITY	MAKE
RUBIN	NEW YORK	FORD
OLLE	BEVERLEY HILLS	GENERAL MOTORS
ADKINSON	BEVERLEY HILLS	FORD
WALLACE	NEW YORK	MAZDA
SPEISER	BEVERLEY HILLS	FORD

Referencing Statements within a Program

Statement reference notation is used to refer to previous statements in a program in order to specify processing over a particular range of data, to override Natural's default referencing (as described for each statement in the Natural Statements documentation, where applicable), or for documentation purposes.

Any Natural statement which causes a processing loop to be initiated and/or causes data elements in a database to be accessed (for example, READ, FIND, HISTOGRAM, SORT, REPEAT, FOR) can be referenced.

When multiple processing loops are used in a program, reference notation is used to uniquely identify the particular database field to be processed by referring back to the statement that originally accessed that field in the database. (If a field can be referenced in such a way, this is indicated in the "Reference Permitted" column of the "Operand Definition Table" in the statement description in the Natural Statements documentation.)

In addition, reference notation can be specified in some statements; for example, AT START OF DATA, AT END OF DATA, AT BREAK and ESCAPE bottom. Without reference notation, an AT START OF DATA, AT END OF DATA or AT BREAK statement will be related to the *outermost* active READ, FIND, HISTOGRAM, SORT or READ WORK FILE loop. With reference notation, you can relate it to another active processing loop.

If reference notation is specified with an ESCAPE bottom statement, processing will continue with the first statement following the processing loop identified by the reference notation.

Statement reference notation may be specified in the form of a *statement label* or a *source-code line number*.

A statement label consists of several characters, the last of which must be a period (.). The period serves to identify the entry as a label.

A statement that is to be referenced is marked with a label by placing the label at the beginning of the line that contains the statement. For example:

```
0030 ...
    0040 READ1. READ VIEWXYZ BY NAME
    0050 ...
```

In the statement that references the marked statement, the label is placed in parentheses at the location indicated in the statement's syntax diagram (as described in the Natural Statements documentation). For example:

```
AT BREAK (READ1.) OF NAME
```

If source-code line numbers are used for referencing, they must be specified as 4-digit numbers (leading zeros must not be omitted) and in parentheses. For example:

```
AT BREAK (0040) OF NAME
```

In a statement where the label/line number relates a particular field to a previous statement, the label/line number is placed in parentheses after the field name. For example:

```
DISPLAY NAME (READ1.) JOB-TITLE (READ1.) MAKE MODEL
```

Line numbers and labels can be used interchangeably.

Example with Line Numbers:

The following program uses line numbers for referencing. In this particular example, the line numbers refer to the statements that would be referenced in any case by default.

```
0010 ** Example Program 'LABELX01'
0020 DEFINE DATA LOCAL
0030 1 MYVIEW1 VIEW OF EMPLOYEES
0040 2 NAME
0050 2 FIRST-NAME
0060 2 PERSONNEL-ID
0070 1 MYVIEW2 VIEW OF VEHICLES
0080 2 PERSONNEL-ID
0090 2 MAKE
0100 END-DEFINE
0110 *
0120 LIMIT 15
0130 READ MYVIEW1 BY NAME STARTING FROM 'JONES'
0140 FIND MYVIEW2 WITH PERSONNEL-ID = PERSONNEL-ID (0130)
0150 IF NO RECORDS FOUND
0160 MOVE '***NO CAR***' TO MAKE
0170 END-NOREC
0180 DISPLAY NOTITLE NAME (0130) (IS=ON) FIRST-NAME (0130) (IS=ON)
0190 MAKE (0140)
0200 END-FIND /* (0140)
0210 END-READ /* (0130)
0220 END
```

Example with Labels:

The following example illustrates the use of statement reference labels. It is identical to the previous program, except that labels are used for referencing instead of line numbers.

```
0010 ** Example Program 'LABELX02'
0020 DEFINE DATA LOCAL
0030 1 MYVIEW1 VIEW OF EMPLOYEES
0040 2 NAME
0050 2 FIRST-NAME
0060 2 PERSONNEL-ID
0070 1 MYVIEW2 VIEW OF VEHICLES
0080 2 PERSONNEL-ID
0090 2 MAKE
0100 END-DEFINE
0110 *
0120 LIMIT 15
0130 RD. READ MYVIEW1 BY NAME STARTING FROM 'JONES'
0140 FD. FIND MYVIEW2 WITH PERSONNEL-ID = PERSONNEL-ID (FD.)
0150 IF NO RECORDS FOUND
0160 MOVE '***NO CAR***' TO MAKE
0170 END-NOREC
0180 DISPLAY NOTITLE NAME (RD.) (IS=ON) FIRST-NAME (RD.) (IS=ON)
0190 MAKE (FD.)
0200 END-FIND /* (FD.)
0210 END-READ /* (RD.)
0220 END
```

Both programs produce the following output:

NAME	FIRST-NAME	MAKE
JONES	VIRGINIA MARSHA	***NO CAR*** CHRYSLER CHRYSLER
	ROBERT LILLY	GENERAL MOTORS ***NO CAR***
	EDWARD MARTHA	GENERAL MOTORS ***NO CAR***
	LAUREL KEVIN	GENERAL MOTORS 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***

Control Breaks

A control break occurs when the value of a control field changes.

The execution of statements can be made dependent on a control break. A control break can also be used for the evaluation of Natural system functions. System functions are discussed later in this section.

- AT BREAK Statement
- Automatic Break Processing
- BEFORE BREAK PROCESSING Statement
- User-Initiated Break Processing - The PERFORM BREAK PROCESSING Statement

AT BREAK Statement

With the statement AT BREAK, you specify the processing which is to be performed whenever a control break occurs, that is, whenever the value of a control field which you specify with the AT BREAK statement changes. As a control field, you can use a database field or a user-defined variable.

Control Break Based on a Database Field

The field specified as control field in an AT BREAK statement is usually a database field.

Example:

```
...
  AT BREAK OF DEPT
    statements
  END-BREAK
...
```

In this example, the control field is the database field DEPT; if the value of the field changes, for example, FROM "SALE01" to "SALE02", the *statements* specified in the AT BREAK statement would be executed.

Instead of an entire field, you can also use only part of a field as a control field. With the notation */n/* you can determine that only the first *n* positions of a field are to be checked for a change in value.

Example:

```
...
  AT BREAK OF DEPT /4/
    statements
  END-BREAK
...
```

In this example, the specified *statements* would only be executed if the value of the first 4 positions of the field DEPT changes, for example, FROM "SALE" to "TECH"; if, however, the field value changes from "SALE01" to "SALE02", this would be ignored and no AT BREAK processing performed.

Example of AT BREAK Statement using a Database Field:

```
** Example Program 'ATBEX01'
  DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEES
    2 NAME
    2 CITY
    2 COUNTRY
    2 JOB-TITLE
    2 SALARY (1:1)
  END-DEFINE
```

```

*
READ (5) MYVIEW BY CITY WHERE COUNTRY = 'USA'
  DISPLAY CITY (AL=9) NAME 'POSITION' JOB-TITLE 'SALARY' SALARY (1)
  AT BREAK OF CITY
    WRITE / OLD(CITY) (EM=X^X^X^X^X^X^X^X^X^X^X)
      5X 'AVERAGE:' T*SALARY AVER(SALARY(1)) //
      COUNT(SALARY(1)) 'RECORDS FOUND' /
  END-BREAK
  AT END OF DATA
    WRITE 'TOTAL (ALL RECORDS):' T*SALARY(1) TOTAL(SALARY(1))
  END-ENDDATA
END-READ
END

```

In the above program, the first WRITE statement is executed whenever the value of the field CITY changes. In the AT BREAK statement, the system functions OLD, AVER and COUNT are evaluated (and output in the WRITE statement). In the AT END OF DATA statement, the system function TOTAL is evaluated. The program produces the following output:

Page	1	97-08-19	18:17:27
CITY	NAME	POSITION	SALARY
-----	-----	-----	-----
AIKEN	SENKO	PROGRAMMER	31500
A I K E N	AVERAGE :		31500
	1 RECORDS FOUND		
ALBUQUERQ	HAMMOND	SECRETARY	22000
ALBUQUERQ	ROLLING	MANAGER	34000
ALBUQUERQ	FREEMAN	MANAGER	34000
ALBUQUERQ	LINCOLN	ANALYST	41000
A L B U Q U E R Q U E	AVERAGE :		32750
	4 RECORDS FOUND		
TOTAL (ALL RECORDS):			162500

Control Break Based on a User-Defined Variable

A user-defined variable can also be used as control field in an AT BREAK statement.

In the following program, the user-defined variable #LOCATION is used as control field.

```

** Example Program 'ATBREX02'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
  2 COUNTRY
  2 JOB-TITLE
  2 SALARY (1:1)
1 #LOCATION (A20)
END-DEFINE
*
READ (5) MYVIEW BY CITY WHERE COUNTRY = 'USA'
  BEFORE BREAK PROCESSING
    COMPRESS CITY 'USA' INTO #LOCATION
  END-BEFORE
  DISPLAY #LOCATION 'POSITION' JOB-TITLE 'SALARY' SALARY (1)
  AT BREAK OF #LOCATION
    SKIP 1
  END-BREAK
END-READ
END

```

The above program produces the following output:

#LOCATION	POSITION	SALARY
AIKEN USA	PROGRAMMER	31500
ALBUQUERQUE USA	SECRETARY	22000
ALBUQUERQUE USA	MANAGER	34000
ALBUQUERQUE USA	MANAGER	34000
ALBUQUERQUE USA	ANALYST	41000

Multiple Control Break Levels

As explained above, the notation "/n/" allows some portion of a field to be checked for a control break. It is possible to combine several AT BREAK statements, using an entire field as control field for one break and part of the same field as control field for another break. In such a case, the break at the lower level (entire field) must be specified before the break at the higher level (part of field); that is, in the first AT BREAK statement the entire field must be specified as control field, and in the second one part of the field.

The following example program illustrates this, using the field DEPT as well as the first 4 positions of that field (DEPT /4/).

```

** Example Program 'ATBREX03'
  DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEES
    2 NAME
    2 JOB-TITLE
    2 DEPT
    2 SALARY      (1:1)
    2 CURR-CODE  (1:1)
  END-DEFINE
  READ MYVIEW BY DEPT STARTING FROM 'SALE40' ENDING AT 'TECH10'
                                WHERE SALARY(1) GT 47000 AND CURR-CODE(1) = 'USD'
    AT BREAK OF DEPT
      WRITE '*** LOWEST BREAK LEVEL ***' /
    END-BREAK
    AT BREAK OF DEPT /4/
      WRITE '*** HIGHEST BREAK LEVEL ***'
    END-BREAK
    DISPLAY DEPT NAME 'POSITION' JOB-TITLE
  END-READ
  END

```

DEPARTMENT CODE	NAME	POSITION
TECH05	HERZOG	MANAGER
TECH05	LAWLER	MANAGER
TECH05	MEYER	MANAGER
*** LOWEST BREAK LEVEL ***		
TECH10	DEKKER	DBA
*** LOWEST BREAK LEVEL ***		
*** HIGHEST BREAK LEVEL ***		

In the following program, one blank line is output whenever the value of the field DEPT changes; and whenever the value in the first 4 positions of DEPT changes, a record count is carried out by evaluating the system function COUNT.

```

** Example Program 'ATBREX04'
  DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEES
    2 DEPT
    2 REDEFINE DEPT
      3 #GENDEP (A4)
    2 NAME
    2 SALARY (1)

```

```

END-DEFINE
WRITE TITLE '** PERSONS WITH SALARY > 30000, SORTED BY DEPARTMENT **' /
LIMIT 9
READ MYVIEW BY DEPT FROM 'A' WHERE SALARY(1) > 30000
  DISPLAY 'DEPT' DEPT NAME 'SALARY' SALARY(1)
  AT BREAK OF DEPT
    SKIP 1
  END-BREAK
  AT BREAK OF DEPT /4/
    WRITE COUNT(SALARY(1)) 'RECORDS FOUND IN:' OLD(#GENDEP) /
  END-BREAK
END-READ
END

```

** PERSONS WITH SALARY > 30000, SORTED BY DEPARTMENT **		
DEPT	NAME	SALARY

ADMA01	JENSEN	180000
ADMA01	PETERSEN	105000
ADMA01	MORTENSEN	320000
ADMA01	MADSEN	149000
ADMA01	BUHL	642000
ADMA02	HERMANSEN	391500
ADMA02	PLOUG	162900
ADMA02	HANSEN	234000
8 RECORDS FOUND IN: ADMA		
COMP01	HEURTEBISE	168800
1 RECORDS FOUND IN: COMP		

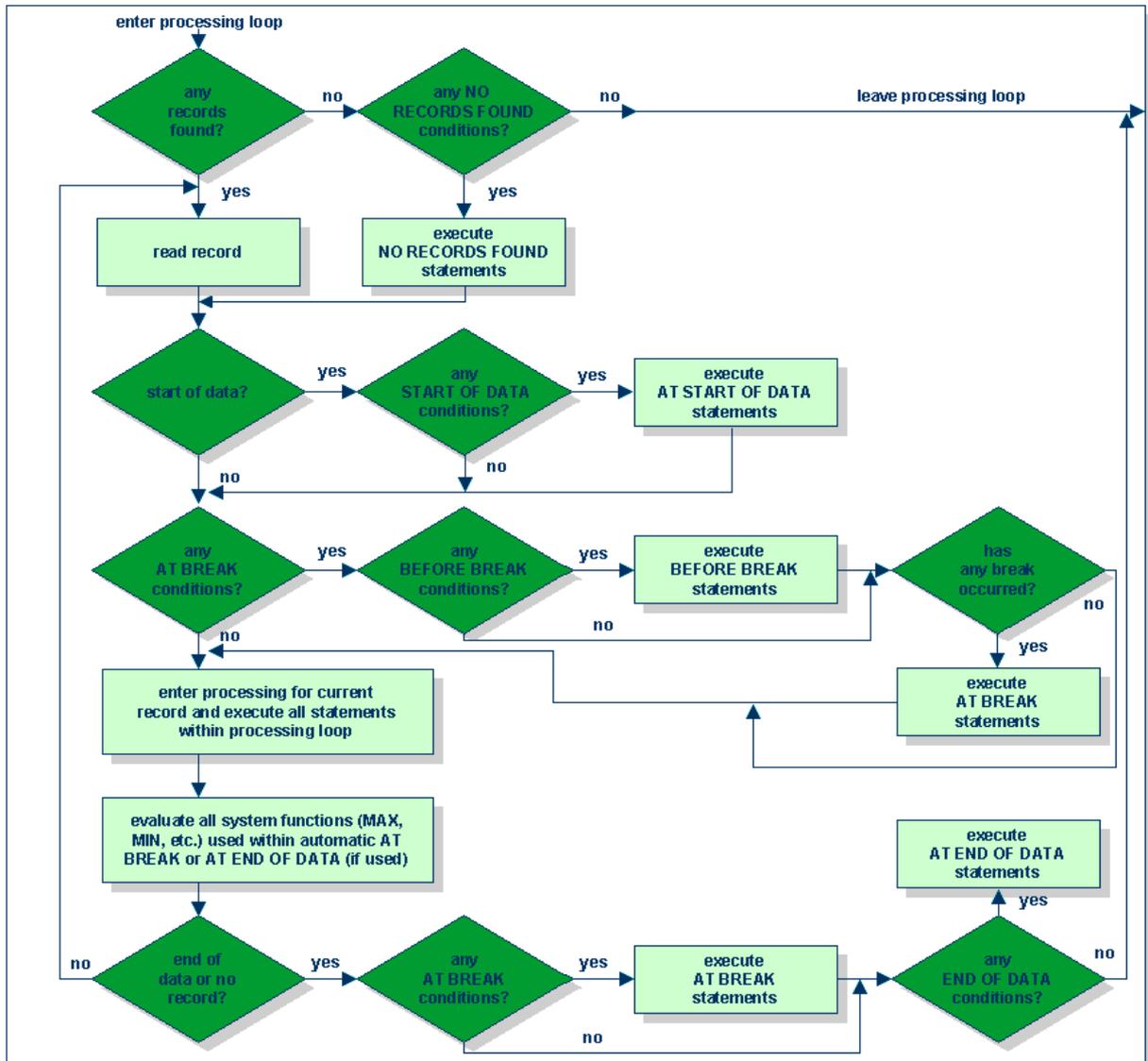
Automatic Break Processing

Automatic break processing is in effect for a FIND, READ, HISTOGRAM, SORT or READ WORK FILE processing loop which contains an AT BREAK statement.

The value of the control field specified with the AT BREAK statement is checked only for records which satisfy the selection criteria of both the WITH clause and the WHERE clause.

Natural system functions (AVER, MAX, MIN, etc.) are evaluated for each record after all statements within the processing loop have been executed. System functions are not evaluated for any record which is rejected by WHERE criteria.

The figure below illustrates the flow logic of automatic break processing.



BEFORE BREAK PROCESSING Statement

With the BEFORE BREAK PROCESSING statement, you can specify statements that are to be executed immediately before a control break; that is, before the value of the control field is checked, before the statements specified in the AT BREAK block are executed, and before any Natural system functions are evaluated.

Example of BEFORE BREAK PROCESSING Statement:

```

** Example Program 'BEFORX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:1)
  2 BONUS (1:1,1:1)
1 #INCOME (P11)
END-DEFINE
*
LIMIT 5
READ MYVIEW BY NAME FROM 'B'
  BEFORE BREAK PROCESSING
    COMPUTE #INCOME = SALARY(1) + BONUS(1,1)
  END-BEFORE
  DISPLAY NOTITLE NAME FIRST-NAME (AL=10)
    'ANNUAL/INCOME' #INCOME
    'SALARY' SALARY(1) (LC==) / '+ BONUS' BONUS(1,1) (IC==)
  AT BREAK OF #INCOME
    WRITE T*#INCOME '-' (24)
  END-BREAK
END-READ
END

```

NAME	FIRST-NAME	ANNUAL INCOME	SALARY + BONUS
BACHMANN	HANS	297546 =	293546 +4000
BAECKER	JOHANNES	420244 =	413644 +6600
BAECKER	KARL	52650 =	48600 +4050
BAGAZJA	MARJAN	152700 =	129700 +23000
BAILLET	PATRICK	198500 =	188000 +10500

User-Initiated Break Processing - The PERFORM BREAK PROCESSING Statement

With automatic break processing, the statements specified in an AT BREAK block are executed whenever the value of the specified control field changes - regardless of the position of the AT BREAK statement in the processing loop.

With a PERFORM BREAK PROCESSING statement, you can perform break processing at a specified position in a processing loop: the PERFORM BREAK PROCESSING statement is executed when it is encountered in the processing flow of the program.

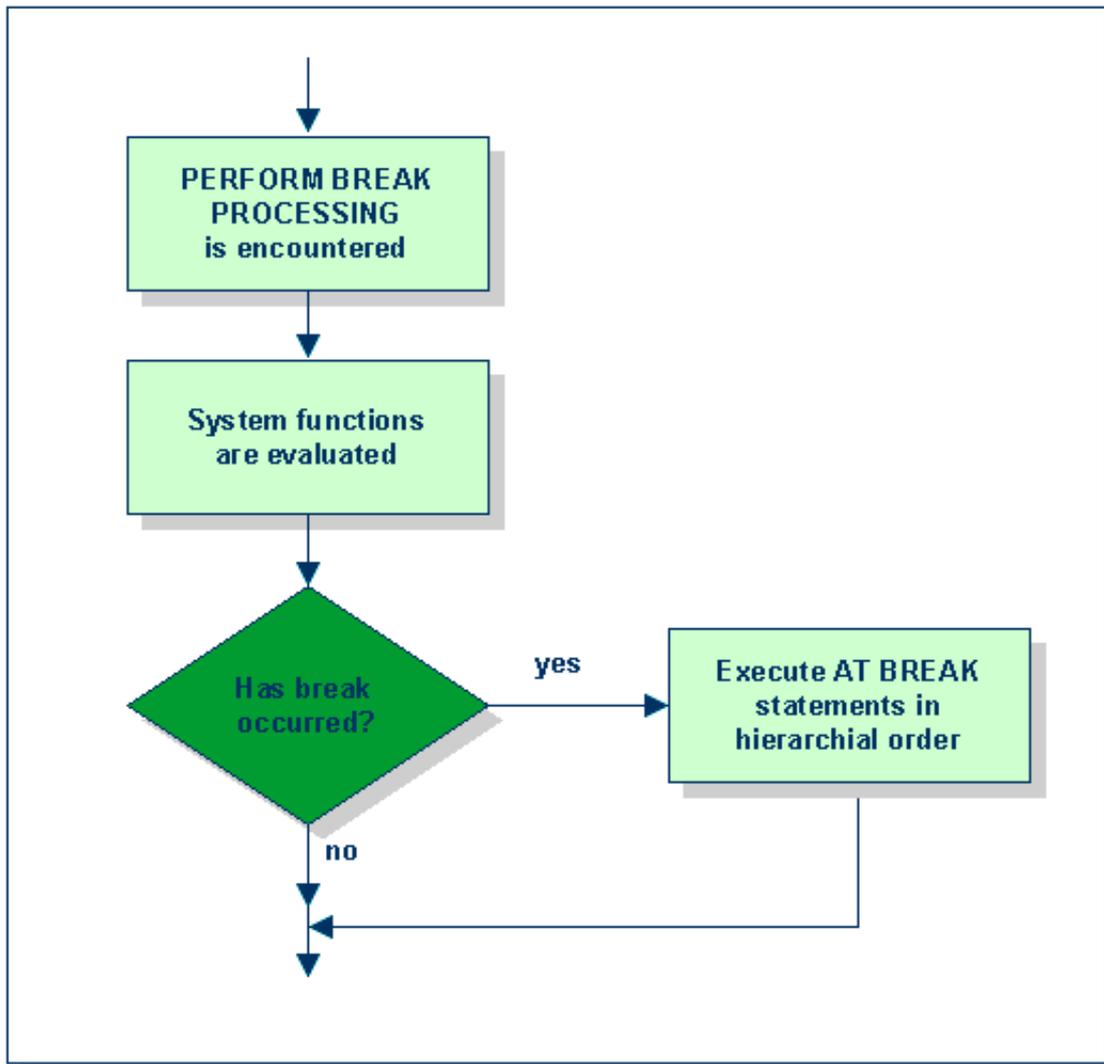
Immediately after the PERFORM BREAK PROCESSING, you specify one or more AT BREAK statement blocks:

```
...  
  PERFORM BREAK PROCESSING  
    AT BREAK OF field1  
      statements  
    END-BREAK  
    AT BREAK OF field2  
      statements  
    END-BREAK  
  ...
```

When a PERFORM BREAK PROCESSING is executed, Natural checks if a break has occurred; that is, if the value of the specified control field has changed; and if it has, the specified statements are executed.

With PERFORM BREAK PROCESSING, system functions are evaluated *before* Natural checks if a break has occurred.

The following figure illustrates the flow logic of user-initiated break processing:



Example of PERFORM BREAK PROCESSING Statement:

```

** Example Program 'PERFBX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 DEPT
  2 SALARY (1:1)
1 #CNTL      (N2)
END-DEFINE
*
LIMIT 7
READ MYVIEW BY DEPT
  AT BREAK OF DEPT          /* <- automatic break processing
  SKIP 1
  WRITE 'SUMMARY FOR ALL SALARIES          '
    'SUM:'  SUM(SALARY(1))
    'TOTAL:' TOTAL(SALARY(1))
  ADD 1 TO #CNTL
END-BREAK
IF SALARY (1) GREATER THAN 100000 OR BREAK #CNTL
  PERFORM BREAK PROCESSING /* <- user-initiated break processing
  AT BREAK OF #CNTL
    WRITE 'SUMMARY FOR SALARY GREATER 100000'
      'SUM:'  SUM(SALARY(1))
      'TOTAL:' TOTAL(SALARY(1))
  END-BREAK
END-IF
IF SALARY (1) GREATER THAN 150000 OR BREAK #CNTL
  PERFORM BREAK PROCESSING /* <- user-initiated break processing
  AT BREAK OF #CNTL
    WRITE 'SUMMARY FOR SALARY GREATER 150000'
      'SUM:'  SUM(SALARY(1))
      'TOTAL:' TOTAL(SALARY(1))
  END-BREAK
END-IF
  DISPLAY NAME DEPT SALARY(1)
END-READ
END
    
```

Page	1	97-08-18	17:11:11
	NAME	DEPARTMENT CODE	ANNUAL SALARY

	JENSEN	ADMA01	180000
	PETERSEN	ADMA01	105000
	MORTENSEN	ADMA01	320000
	MADSEN	ADMA01	149000
	BUHL	ADMA01	642000
	SUMMARY FOR ALL SALARIES	SUM:	1396000 TOTAL: 1396000
	SUMMARY FOR SALARY GREATER 100000	SUM:	1396000 TOTAL: 1396000
	SUMMARY FOR SALARY GREATER 150000	SUM:	1142000 TOTAL: 1142000
	HERMANSEN	ADMA02	391500
	PLOUG	ADMA02	162900
	SUMMARY FOR ALL SALARIES	SUM:	554400 TOTAL: 1950400
	SUMMARY FOR SALARY GREATER 100000	SUM:	554400 TOTAL: 1950400
	SUMMARY FOR SALARY GREATER 150000	SUM:	554400 TOTAL: 1696400

Further Example of AT BREAK Statement:

See program ATBREX06 in library SYSEXP.

Data Computation

This section discusses the arithmetic statements COMPUTE, ADD, SUBTRACT, MULTIPLY and DIVIDE; as well as the statements MOVE and COMPRESS, which are used to transfer values from one field to another.

- Format of Fields
- COMPUTE Statement
- Statements MOVE and COMPUTE
- Statements ADD, SUBTRACT, MULTIPLY and DIVIDE
- COMPRESS Statement
- Mathematical Functions

Format of Fields

For optimum processing, user-defined variables used in arithmetic statements should be defined with format P (packed numeric).

COMPUTE Statement

The COMPUTE statement is used to perform arithmetic operations. The following connecting operators are available:

Exponentiation	**
Multiplication	*
Division	/
Addition	+
Subtraction	-

Parentheses may be used to indicate logical grouping.

Example 1:

```
COMPUTE LEAVE-DUE = LEAVE-DUE * 1.1
```

In this example, the value of the field LEAVE-DUE is multiplied by 1.1, and the result is placed in the field LEAVE-DUE.

Example 2:

```
COMPUTE #A = SQRT (#B)
```

In this example, the square root of the value of the field #B is evaluated, and the result is assigned to the field #A. "SQRT" is a mathematical function supported in the arithmetic statements COMPUTE, ADD, SUBTRACT, MULTIPLY, and DIVIDE. An overview of mathematical functions is provided later in this section.

Example 3:

```
COMPUTE #INCOME = BONUS (1,1) + SALARY (1)
```

In this example, the first bonus of the current year and the current salary amount are added and assigned to the field #INCOME.

Statements MOVE and COMPUTE

The statements MOVE and COMPUTE can be used to transfer the value of an operand into one or more fields. The operand may be a constant such as a text item or a number, a database field, a user-defined variable, a system variable, or, in certain cases, a system function.

The difference between the two statements is that in the MOVE statement the value to be moved is specified on the left; in the COMPUTE statement the value to be assigned is specified on the right, as shown in the following examples.

Examples:

```
MOVE NAME TO #LAST-NAME
COMPUTE #LAST-NAME = NAME
```

Statements ADD, SUBTRACT, MULTIPLY and DIVIDE

The ADD, SUBTRACT, MULTIPLY and DIVIDE statements are used to perform arithmetic operations.

Examples:

```
ADD +5 -2 -1 GIVING #A
SUBTRACT 6 FROM 11 GIVING #B
MULTIPLY 3 BY 4 GIVING #C
DIVIDE 3 INTO #D GIVING #E
```

All four statements have a ROUNDED option, which you can use if you wish the result of the operation to be rounded.

The Natural Statements documentation provides more detailed information on these statements.

Example of MOVE, SUBTRACT and COMPUTE Statements:

The following program demonstrates the use of user-defined variables in arithmetic statements. It calculates the ages and wages of three employees and outputs these.

```
** Example Program 'COMPUX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 BIRTH
  2 JOB-TITLE
  2 SALARY          (1:1)
  2 BONUS           (1:1,1:1)
1 #DATE            (N8)
1 REDEFINE #DATE
  2 #YEAR           (N4)
  2 #MONTH          (N2)
  2 #DAY            (N2)
1 #BIRTH-YEAR     (A4)
1 REDEFINE #BIRTH-YEAR
  2 #BIRTH-YEAR-N (N4)
1 #AGE            (N3)
1 #INCOME         (P9)
END-DEFINE
*
MOVE *DATN TO #DATE
*
READ (3) MYVIEW BY NAME STARTING FROM 'JONES'
MOVE EDITED BIRTH (EM=YYYY) TO #BIRTH-YEAR
```

```

SUBTRACT #BIRTH-YEAR-N FROM #YEAR GIVING #AGE
COMPUTE #INCOME = BONUS (1:1,1:1) + SALARY (1:1)
DISPLAY NAME 'POSITION' JOB-TITLE #AGE #INCOME
END-READ
END
    
```

Page	1		99-01-22	12:42:50
	NAME	POSITION	#AGE	#INCOME
	-----	-----	-----	-----
	JONES	MANAGER	58	55000
	JONES	DIRECTOR	53	50000
	JONES	PROGRAMMER	43	31000

COMPRESS Statement

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

Leading zeros in a numeric field and trailing blanks in an alphanumeric field are suppressed before the field value is moved to the receiving field.

By default, the transferred values are separated from one another by a single blank in the receiving field. Other separating possibilities are described in the Natural Statements documentation.

Example:

```

COMPRESS 'NAME:' FIRST-NAME #LAST-NAME INTO #FULLNAME
    
```

In this example, a text constant ('NAME:'), a database field (FIRST-NAME) and a user-defined variable (#LAST-NAME) are combined into one user-defined variable (#FULLNAME) using a COMPRESS statement.

For further information on the COMPRESS statement, please refer to the Natural Statements documentation.

Example of COMPRESS and MOVE Statements:

```

** Example Program 'ComPRX01'
DEFINE DATA LOCAL
1 MYVIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 MIDDLE-I
1 #LAST-NAME (A15)
1 #FULL-NAME (A30)
END-DEFINE
*
READ (3) MYVIEW BY NAME STARTING FROM 'JONES'
MOVE NAME TO #LAST-NAME
COMPRESS 'NAME:' FIRST-NAME MIDDLE-I #LAST-NAME INTO #FULL-NAME
DISPLAY #FULL-NAME (UC==) FIRST-NAME 'I' MIDDLE-I (AL=1) NAME
END-READ
END
    
```

The above program illustrates the use of the statements MOVE and COMPRESS. Notice the output format of the compressed field:

Page	1		97-08-18	17:47:03
	#FULL-NAME	FIRST-NAME	I	NAME
	=====	-----	-	-----
	NAME: VIRGINIA J JONES	VIRGINIA	J JONES	
	NAME: MARSHA JONES	MARSHA	JONES	
	NAME: ROBERT B JONES	ROBERT	B JONES	

In multiple-line displays, it may be useful to combine fields/text in a user-defined variable by using a COMPRESS statement.

Example of COMPRESS Statement:

In the following program, three user-defined variables are used: #FULLSAL, #FULLNAME, and #FULLCITY. #FULLSAL, for example, contains the text 'SALARY:' and the database fields SALARY and CURR-CODE. The WRITE statement then references only the compressed variables.

```

** Example Program 'COMPRX02'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
2 NAME
2 FIRST-NAME
2 SALARY (1:1)
2 CURR-CODE (1:1)
2 CITY
2 ADDRESS-LINE (1:1)
2 ZIP
1 #FULLSAL (A25)
1 #FULLNAME (A25)
1 #FULLCITY (A25)
END-DEFINE
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
COMPRESS 'SALARY:' CURR-CODE(1) SALARY(1) INTO #FULLSAL
COMPRESS FIRST-NAME NAME INTO #FULLNAME
COMPRESS ZIP CITY INTO #FULLCITY
DISPLAY 'NAME AND ADDRESS' NAME (EM=X^X^X^X^X^X^X^X^X^X^X^X^X)
WRITE 1/5 #FULLNAME 1/37 #FULLSAL
2/5 ADDRESS-LINE (1)
3/5 #FULLCITY
SKIP 1
END-READ
END

```

Page	1	97-08-19	18:01:17
NAME AND ADDRESS			

R U B I N	SYLVIA RUBIN	SALARY: USD 17000	
	2003 SARAZEN PLACE		
	10036 NEW YORK		
W A L L A C E	MARY WALLACE	SALARY: USD 38000	
	12248 LAUREL GLADE C		
	10036 NEW YORK		
K E L L O G G	HENRIETTA KELLOGG	SALARY: USD 52000	
	1001 JEFF RYAN DR.		
	19711 NEWARK		

Mathematical Functions

The following Natural mathematical functions are supported in arithmetic processing statements (ADD, COMPUTE, DIVIDE, MULTIPLY, SUBTRACT).

Function	Meaning
ABS (<i>field</i>)	Absolute value of <i>field</i> .
ATN (<i>field</i>)	Arc tangent of <i>field</i> .
COS (<i>field</i>)	Cosine of <i>field</i> .
EXP (<i>field</i>)	Exponential of <i>field</i> .
FRAC (<i>field</i>)	Fractional part of <i>field</i> .
INT (<i>field</i>)	Integer part of <i>field</i> .
LOG (<i>field</i>)	Natural logarithm of <i>field</i> .
SGN (<i>field</i>)	Sign of <i>field</i> .
SIN (<i>field</i>)	Sine of <i>field</i> .
SQRT (<i>field</i>)	Square root of <i>field</i> .
TAN (<i>field</i>)	Tangent of <i>field</i> .
VAL (<i>field</i>)	Numeric value of an alphanumeric <i>field</i> .

See the Natural Reference documentation for a detailed explanation of each mathematical function.

Further Examples of COMPUTE, MOVE and COMPRESS Statements:

See programs WRITEX11, IFX03 and COMPRX03 in library SYSEXPG.

System Variables and System Functions

The following topics are covered below:

- System Variables
- System Functions

System Variables

Natural system variables contain information about the current Natural session, such as: the current library, the user and terminal identification; the current status of a loop processing; the current report processing status; the current date and time.

This information may be used in Natural programs by specifying the appropriate system variables. For example:

System Variable	Content
*INIT-USER	The user ID of the terminal user.
*LANGUAGE	The language in effect.
*LIBRARY-ID	The current library ID.
*INIT-ID	The terminal ID.
*ERROR-NR	The Natural error number.
*PAGE-NUMBER	The current value for page number.
*COUNTER	The number of times a processing loop has been entered.
*NUMBER	The number of records selected.

Some date and time system variables include the following:

System Variable	Content
*DATU	Current date in format MM/DD/YY
*DAT4U	Current date in format MM/DD/YYYY
*DATE	Current date in format DD/MM/YY
*DAT4E	Current date in format DD/MM/YYYY
*DATI	Current date in format YY-MM-DD
*DAT4I	Current date in format YYYY-MM-DD
*DATD	Current date in format DD.MM.YY
*DAT4D	Current date in format DD.MM.YYYY
*TIME	Time of day in format HH:MM:SS.T
*TIMN	Time of day in format HHMMSSST

The names of all system variables begin with an asterisk (*).

Date and time system variables may be specified in a DISPLAY, WRITE, PRINT, MOVE or COMPUTE statement.

For further information on system variables, see System Variables in the Natural Reference documentation.

System Functions

Natural system functions are a set of statistical and mathematical functions that can be applied to the data after a record has been processed but before break processing occurs.

System functions may be specified in a WRITE, DISPLAY, PRINT, COMPUTE or MOVE statement that is used in conjunction with an AT END OF PAGE, AT END OF DATA or AT BREAK statement.

In the case of an AT END OF PAGE statement, the corresponding DISPLAY statement must include the GIVE SYSTEM FUNCTIONS clause (as shown in the example below).

The following system functions are available:

System Function	Information Returned
AVER (<i>field</i>)	Average of all values for <i>field</i> .
NAVER (<i>field</i>)	Average of all values for <i>field</i> , not counting null values.
MAX (<i>field</i>)	Maximum value of <i>field</i> .
MIN (<i>field</i>)	Minimum value of <i>field</i> .
NMIN (<i>field</i>)	Minimum value of <i>field</i> , not counting null values.
OLD (<i>field</i>)	Value of <i>field</i> value prior to change in control value (AT BREAK condition).
SUM (<i>field</i>)	Sum of all <i>field</i> values (reset when control value in AT BREAK changes).
TOTAL (<i>field</i>)	Total of all <i>field</i> values (not reset when control value in AT BREAK changes).
COUNT (<i>field</i>)	Number of passes through a processing loop.
NCOUNT (<i>field</i>)	Number of passes through a processing loop, not counting passes where the control <i>field</i> contains a null value.

For further information on system functions, see Natural System Functions in the Natural Reference documentation.

Example of System Variables and System Functions:

```

** Example Program 'SYSVAX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
  2 INCOME (1:1)
  3 CURR-CODE
  3 SALARY
  3 BONUS (1:1)
END-DEFINE
*
WRITE TITLE 'EMPLOYEE SALARY REPORT AS OF' *DATE /
READ (3) MYVIEW BY CITY STARTING FROM 'E'
  DISPLAY GIVE SYSTEM FUNCTIONS
    NAME (AL=15) JOB-TITLE (AL=15) INCOME (1:1)
  AT START OF DATA
    WRITE 'REPORT CREATED AT:' *TIME 'HOURS' /
  END-START
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD (NAME) /
  END-ENDDATA
END-READ
AT END OF PAGE
  WRITE 'AVERAGE SALARY:' AVER(SALARY(1))
END-ENDPAGE
END

```

The above program illustrates the use of system variables and system functions:

The system variable *DATE is output with the WRITE TITLE statement; the system variable *TIME is output with the AT START OF DATA statement.

The system function OLD is used in the AT END OF DATA statement; the system function AVER is used in the AT END OF PAGE statement.

Note how the system variables and system function are displayed:

EMPLOYEE SALARY REPORT AS OF 18/01/1999				
NAME	CURRENT POSITION	INCOME		
		CURRENCY CODE	ANNUAL SALARY	BONUS

REPORT CREATED AT: 11:51:29.3 HOURS				
DUYVERMAN	PROGRAMMER	USD	34000	0
PRATT	SALES PERSON	USD	38000	9000
MARKUSH	TRAINEE	USD	22000	0
LAST PERSON SELECTED: MARKUSH				
AVERAGE SALARY:		31333		

Further Examples of System Variables:

See programs EDITMX05, READX04 and WTITLX01 in library SYSEXP.

Further Examples of System Functions:

See programs ATBREX06 and ATENPX01 in library SYSEXP.

Stack

The Natural stack is a kind of "intermediate storage" in which you can store Natural commands, user-defined commands, and input data to be used by an INPUT statement. In the stack you can store a series of functions which are frequently executed one after the other, such as a series of logon commands.

The data/commands stored in the stack are "stacked" on top of one another. You can decide whether to put them on top or at the bottom of the stack. The data/command in the stack can only be processed in the order in which they are stacked, beginning from the top of the stack.

In a program, you may reference the system variable *DATA to determine the content of the stack (see the Natural Reference documentation for further information).

The total size of the stack is defined by the remaining portion in the ESIZE buffer after allocation for the global data area and the program source area.

The following topics are covered below:

- Stack Processing
- Placing Data in the Stack
- Clearing the Stack

Stack Processing

The processing of the commands/data stored in the stack differs depending on the function being performed.

If a command is expected, that is, the NEXT prompt is about to be displayed, Natural first checks if a command is on the top of the stack. If there is, the NEXT prompt is suppressed and the command is read and deleted from the stack; the command is then executed as if it had been entered manually in response to the NEXT prompt.

If an INPUT statement containing input fields is being executed, Natural first checks if there are any input data on the top of the stack. If there are, these data are passed to the INPUT statement (in delimiter mode); the data read from the stack must be format-compatible with the variables in the INPUT statement; the data are then deleted from the stack.

If an INPUT statement was executed using data from the stack, and this INPUT statement is re-executed via a REINPUT statement, the INPUT statement screen will be re-executed displaying the same data from the stack as when it was executed originally. With the REINPUT statement, no further data are read from the stack.

When a Natural program terminates normally, the stack is flushed beginning from the top until either a command is on the top of the stack or the stack is cleared. When a Natural program is terminated via the terminal command "%%" or with an error, the stack is cleared entirely.

Placing Data in the Stack

The following methods can be used to place data/commands on the stack:

STACK Parameter

The Natural profile parameter `STACK` may be used to place data/commands on the stack. The `STACK` parameter, which is described in the Natural Operations documentation, can be specified by the Natural administrator in the Natural parameter module at the installation of Natural; or you can specify it as a dynamic parameter when you invoke Natural.

When data/commands are to be placed on the stack via the `STACK` parameter, multiple commands must be separated from one another by a semicolon (;). If a command is to be passed within a sequence of data or command elements, it must be preceded by a semicolon.

Data for multiple `INPUT` statements must be separated from one another by a colon (:). Data that are to be read by a separate `INPUT` statement must be preceded by a colon. If a command is to be stacked which requires parameters, no colon is to be placed between the command and the parameters.

Semicolon and colon must not be used within the input data themselves as they will be interpreted as separation characters.

STACK Statement

The `STACK` statement can be used within a program to place data/commands in the stack. The data elements specified in one `STACK` statement will be used for one `INPUT` statement, which means that if data for multiple `INPUT` statements are to be placed on the stack, multiple `STACK` statements must be used.

Data may be placed on the stack either unformatted or formatted:

- If unformatted data are read from the stack, the data string is interpreted in delimiter mode and the characters specified with the session parameters `IA` (Input Assignment character) and `ID` (Input Delimiter character) are processed as control characters for keyword assignment and data separation.
- If formatted data are placed on the stack, each content of a field will be separated and passed to one input field in the corresponding `INPUT` statement.

See the Natural Statements documentation for further information on the `STACK` statement.

FETCH and RUN Statements

The execution of a `FETCH` or `RUN` statement that contains parameters to be passed to the invoked program will result in these parameters being placed on top of the stack.

Clearing the Stack

The contents of the stack can be deleted with the `RELEASE` statement. See the Natural Statements documentation for details on the `RELEASE` statement.

Processing of Date Information

This section covers various aspects concerning the handling of dates in your Natural applications:

- Edit Masks for Date Fields and Date System Variables
- Default Edit Mask for Date - The DTFORM Parameter
- Date Format for Alphanumeric Representation - The DF Parameter
- Date Format for Output - The DFOUT Parameter
- Date Format for Stack - The DFSTACK Parameter
- Year Sliding Window - The YSLW Parameter
- Combinations of DFSTACK and YSLW
- Date Format for Default Page Title - The DFTITLE Parameter

Edit Masks for Date Fields and Date System Variables

If you wish the value of a date field to be output in a specific representation, you usually specify an edit mask for the field. With an edit mask, you determine character by character what the output is to look like.

If you wish to use the current date in a specific representation, you need not define a date field and specify an edit mask for it; instead you can simply use a *date system variable*. Natural provides various date system variables, which contain the current date in different representations. Some of these representations contain a 2-digit year component, some a 4-digit year component.

For more information see the examples of date system variables. For more information and a list of all date system variables, see the Natural Reference documentation.

Default Edit Mask for Date - The DTFORM Parameter

The profile parameter DTFORM determines the default format used for dates as part of the default title on Natural reports, for date constants and for date input.

This date format determines the sequence of the day, month and year components of a date, as well as the delimiter characters to be used between these components.

Possible DTFORM settings are:

Setting	Date Format*	Example
DTFORM=I	<i>yyyy-mm-dd</i>	1997-12-31
DTFORM=G	<i>dd.mm.yyyy</i>	31.12.1997
DTFORM=E	<i>dd/mm/yyyy</i>	31/12/1997
DTFORM=U	<i>mm/dd/yyyy</i>	12/31/1997

* *dd* = day, *mm* = month, *yyyy* = year.

The DTFORM parameter can be set in the Natural parameter module/file or dynamically when Natural is invoked. By default, DTFORM=I applies.

Date Format for Alphanumeric Representation - The DF Parameter

The session parameter DF only applies to date fields for which no edit mask is specified.

If an edit mask is specified, the representation of the field value is determined by the edit mask. If no edit mask is specified, the representation of the field value is determined by the session parameter DF in combination with the DTFORM profile parameter.

With the DF parameter, you can choose one of the following date representations:

DF=S	8-byte representation with 2-digit year component and delimiters (<i>yy-mm-dd</i>).
DF=I	8-byte representation with 4-digit year component without delimiters (<i>yyyymmdd</i>).
DF=L	10-byte representation with 4-digit year component and delimiters (<i>yyyy-mm-dd</i>).

For each representation, the sequence of the day, month and year components, and the delimiter characters used, are determined by the DTFORM parameter.

By default, DF=S applies (except for INPUT statements; see below).

The DF parameter is evaluated at compilation. It can be specified with the FORMAT statement, the statements INPUT, DISPLAY, WRITE and PRINT (at statement and field level), and the statements MOVE, COMPRESS, STACK, RUN and FETCH (at field level).

The DF parameter applies to the following:

- **DISPLAY, WRITE and PRINT:** When the value of a date variable is output with one of these statements, the value is converted to an alphanumeric representation before it is output. The DF parameter determines which representation is used.
- **MOVE and COMPRESS:** When the value of a date variable is transferred to an alphanumeric field with a MOVE or COMPRESS statement, the value is converted to an alphanumeric representation before it is transferred. The DF parameter determines which representation is used.
- **STACK, FETCH and RUN:** When the value of a date variable is placed on the stack, it is converted to alphanumeric representation before it is placed on the stack. The DF parameter determines which representation is used.
The same applies when a date variable is specified as a parameter in a FETCH or RUN statement (as these parameters are also passed via the stack).
- **INPUT:** When a data variable is used in an INPUT statement, the DF parameter determines how a value must be entered in the field.
However, when a date variable for which *no* DF parameter is specified is used in an INPUT statement, the date can be entered either with a 2-digit year component and delimiters or with a 4-digit year component and no delimiters. In this case, too, the sequence of the day, month and year components, and the delimiter characters to be used, are determined by the DTFORM parameter.

With DF=S, only 2 digits are provided for the year information; this means that if a date value contained the century, this information would be lost during the conversion. To retain the century information, you set DF=I or DF=L.

Examples of DF Parameter with WRITE Statements:

```

/* DF=S (default)
  WRITE *DATX /* Output has this format: dd.mm.yy
  END

FORMAT DF=I
  WRITE *DATX /* Output has this format: ddmmyyyy
  END

FORMAT DF=L
  WRITE *DATX /* Output has this format: dd.mm.yyyy
  END

```

These examples assume that DTFORM=G applies.

Example of DF Parameter with MOVE Statement:

```

DEFINE DATA LOCAL
  1 #DATE (D) INIT <D'31/12/1997'>
  1 #ALPHA (A10)
END-DEFINE
...
MOVE #DATE TO #ALPHA /* Result: #ALPHA contains 31/12/97
MOVE #DATE (DF=I) TO #ALPHA /* Result: #ALPHA contains 31121997
MOVE #DATE (DF=L) TO #ALPHA /* Result: #ALPHA contains 31/12/1997
...

```

This example assumes that DTFORM=E applies.

Example of DF Parameter with STACK Statement:

```

DEFINE DATA LOCAL
  1 #DATE (D) INIT <D'1997-12-31'>
  1 #ALPHA1 (A10)
  1 #ALPHA2 (A10)
  1 #ALPHA3 (A10)
END-DEFINE
...
STACK TOP DATA #DATE (DF=S) #DATE (DF=I) #DATE (DF=L)
...
INPUT #ALPHA1 #ALPHA2 #ALPHA3
...
/* Result: #ALPHA1 contains 97-12-31
/*          #ALPHA2 contains 19971231
/*          #ALPHA3 contains 1997-12-31
...

```

This example assumes that DTFORM=I applies.

Example of DF Parameter with INPUT Statement:

```
DEFINE DATA LOCAL
  1 #DATE1 (D)
  1 #DATE2 (D)
  1 #DATE3 (D)
  1 #DATE4 (D)
END-DEFINE
...
INPUT #DATE1 (DF=S) /* Input must have this format: yy-mm-dd
      #DATE2 (DF=I) /* Input must have this format: yyyymmdd
      #DATE3 (DF=L) /* Input must have this format: yyyy-mm-dd
      #DATE4      /* Input must have this format: yy-mm-dd or yyyymmdd
...
```

This example assumes that DTFORM=I applies.

Date Format for Output - The DFOUT Parameter

The session/profile parameter DFOUT only applies to date fields in INPUT, DISPLAY, PRINT and WRITE statements for which no edit mask is specified, and for which no DF parameter applies.

For date fields which are displayed by INPUT, DISPLAY, PRINT and WRITE statements and for which neither an edit mask is specified nor a DF parameter applies, the profile/session parameter DFOUT determines the format in which the field values are displayed.

Possible DFOUT settings are:

DFOUT=S	Date variables are displayed with a 2-digit year component, and delimiters as determined by the DTFORM parameter (<i>yy-mm-dd</i>).
DFOUT=I	Date variables are displayed with a 4-digit year component and no delimiters (<i>yyyymmdd</i>).

By default, DFOUT=S applies. For either DFOUT setting, the sequence of the day, month and year components in the date values is determined by the DTFORM parameter.

The lengths of the date fields are not affected by the DFOUT setting, as either date value representation fits into an 8-byte field.

The DFOUT parameter can be set in the Natural parameter module/file, dynamically when Natural is invoked, or with the system command GLOBALS. It is evaluated at runtime.

Example:

```

DEFINE DATA LOCAL
  1 #DATE (D) INIT <D'1997-12-31'>
  END-DEFINE
  ...
  WRITE #DATE          /* Output if DFOUT=S is set ...: 97-12-31
                        /* Output if DFOUT=I is set ...: 19971231
  WRITE #DATE (DF=L) /* Output (regardless of DFOUT): 1997-12-31
  ...

```

This example assumes that DTFORM=I applies.

Date Format for Stack - The DFSTACK Parameter

The session/profile parameter DFSTACK only applies to date fields used in STACK, FETCH and RUN statements for which no DF parameter has been specified.

The DFSTACK parameter determines the format in which the values of date variables are placed on the stack via a STACK, RUN or FETCH statement.

Possible DFSTACK settings are:

DFSTACK=S	Date variables are placed on the stack with a 2-digit year component, and delimiters as determined by the profile DTFORM parameter (<i>yy-mm-dd</i>).
DFSTACK=C	Same as DFSTACK=S. However, a change in the century will be intercepted at runtime.
DFSTACK=I	Date variables are placed on the stack with a 4-digit year component and no delimiters (<i>yyyymmdd</i>).

By default, DFSTACK=S applies. DFSTACK=S means that when a date value is placed on the stack, it is placed there without the century information (which is lost). When the value is then read from the stack and placed into another date variable, the century is either assumed to be the current one or determined by the setting of the YSLW parameter (see below). This might lead to the century being different from that of the original date value; however, Natural would not issue any error in this case.

DFSTACK=C works the same as DFSTACK=S in that a date value is placed on the stack without the century information. However, if the value is read from the stack and the resulting century is different from that of the original date value (either because of the YSLW parameter, or the original century not being the current one), Natural issues a runtime error.

Note:

This runtime error is already issued at the time when the value is placed on the stack.

DFSTACK=I allows you to place a date value on the stack in a length of 8 bytes without losing the century information.

The DFSTACK parameter can be set in the Natural parameter module/file, dynamically when Natural is invoked, or with the system command GLOBALS. It is evaluated at runtime.

Example:

```

DEFINE DATA LOCAL
  1 #DATE (D) INIT <D'1997-12-31'>
  1 #ALPHA1 (A8)
  1 #ALPHA2 (A10)
END-DEFINE
...
STACK TOP DATA #DATE #DATE (DF=L)
...
INPUT #ALPHA1 #ALPHA2
...
/* Result if DFSTACK=S or =C is set: #ALPHA1 contains 97-12-31
/* Result if DFSTACK=I is set .....: #ALPHA1 contains 19971231
/* Result (regardless of DFSTACK) .: #ALPHA2 contains 1997-12-31
...

```

This example assumes that DTFORM=I and YSLW=0 apply.

Year Sliding Window - The YSLW Parameter

The profile parameter YSLW allows you determine the century of a 2-digit year value.

The YSLW parameter can be set in the Natural parameter module/file or dynamically when Natural is invoked. It is evaluated at runtime when an alphanumeric date value with a 2-digit year component is moved into a date variable. This applies to data values which are:

- used with the mathematical function VAL,
- used with the IS(D) option in a logical condition,
- read from the stack as input data, or
- entered in an input field as input data.

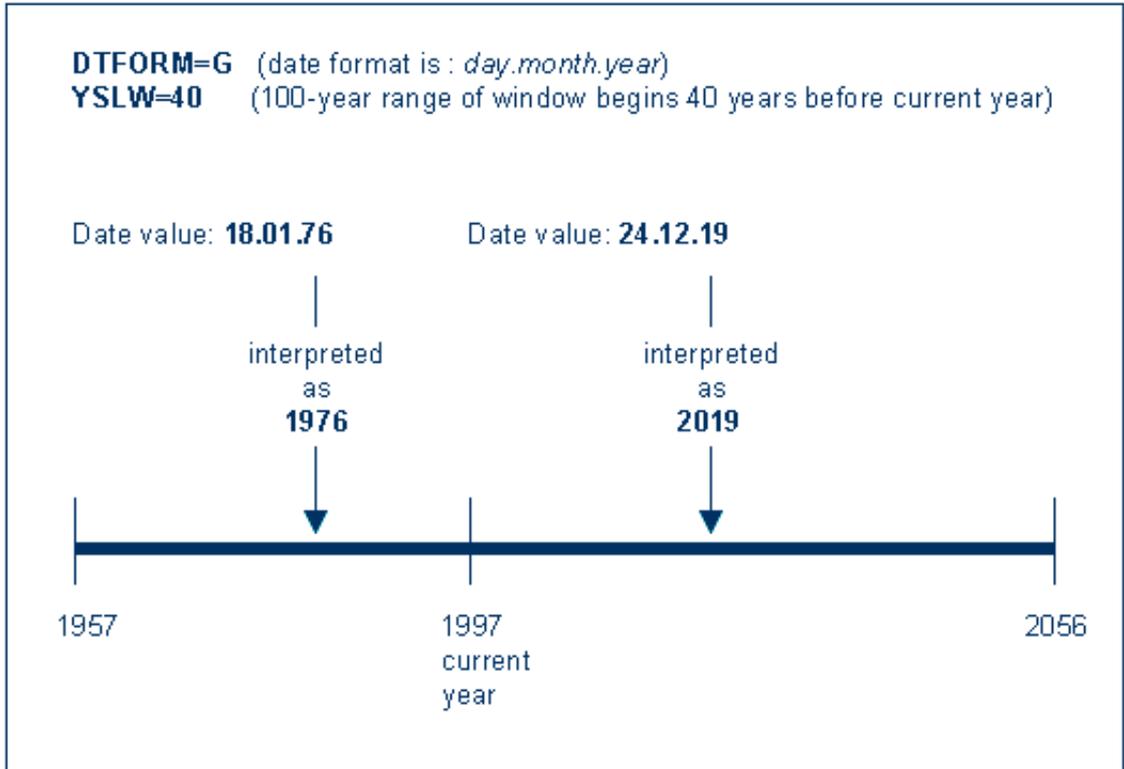
The YSLW parameter determines the range of years covered by a so-called "year sliding window". The sliding-window mechanism assumes a date with a 2-digit year to be within a "window" of 100 years. Within these 100 years, every 2-digit year value can be uniquely related to a specific century.

With the YSLW parameter, you determine how many years in the past that 100-year range is to begin: The YSLW value is subtracted from the current year to determine the first year of the window range.

Possible values of the YSLW parameter are 0 to 99. The default value is YSLW=0, which means that no sliding-window mechanism is used; that is, a date with a 2-digit year is assumed to be in the current century.

Example 1:

If the current year is 1997 and you specify YSLW=40, the sliding window will cover the years 1957 to 2056. A 2-digit year value *nm* from 57 to 99 is interpreted accordingly as 19*nm*, while a 2-digit year value *nm* from 00 to 56 is interpreted as 20*nm*.



Combinations of DFSTACK and YSLW

The following examples illustrate the effects of using various combinations of the parameters DFSTACK and YSLW.

All these examples assume that DTFORM=I applies.

Example 1:

This example assumes the current year to be 1997, and the following parameter settings:

DFSTACK=S (default)

YSLW=20

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'1956-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2           /* year sliding window determines 56 to be 2056
...
/* Result: #DATE2 contains 2056-12-31
```

In this case, the year sliding window is not set appropriately, so that the century information is (inadvertently) changed.

Example 2:

This example assumes the current year to be 1997, and the following parameter settings:

DFSTACK=S (default)

YSLW=50

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'1956-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2           /* year sliding window determines 56 to be 1956
...
/* Result: #DATE2 contains 1956-12-31
```

In this case, the year sliding window is set appropriately, so that the original century information is correctly restored.

Example 3:

This example assumes the current year to be 1997, and the following parameter settings:

DFSTACK=C

YSLW=0 (default)

```

DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'2056-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2           /* 56 is assumed to be in current century -> 1956
...
/* Result: RUNTIME ERROR (UNINTENDED CENTURY CHANGE)

```

In this case, the century information is (inadvertently) changed. However, this change is intercepted by the DFSTACK=C setting.

Example 4:

This example assumes the current year to be 1997, and the following parameter settings:

DFSTACK=C

YSLW=20

```

DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'1956-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2           /* year sliding window determines 56 to be 2056
...
/* Result: RUNTIME ERROR (UNINTENDED CENTURY CHANGE)

```

In this case, the century information is changed due to the year sliding window. However, this change is intercepted by the DFSTACK=C setting.

Date Format for Default Page Title - The DFTITLE Parameter

The session/profile parameter DFTITLE determines the format of the date in a default page title (as output with a DISPLAY, WRITE or PRINT statement).

DFTITLE=S	The date is output with a 2-digit year component and delimiters (<i>yy-mm-dd</i>).
DFTITLE=L	The date is output with a 4-digit year component and delimiters (<i>yyyy-mm-dd</i>).
DFTITLE=I	The date is output with a 4-digit year component and no delimiters (<i>yyyymmdd</i>).

For each of these output formats, the sequence of the day, month and year components, and the delimiter characters used, are determined by the DTFORM parameter.

The DFTITLE parameter can be set in the Natural parameter module/file, dynamically when Natural is invoked, or with the system command GLOBALS. It is evaluated at runtime.

Example:

```
WRITE 'HELLO'
  END
/*
/* Date in page title if DFTITLE=S is set ...: 98-10-31
/* Date in page title if DFTITLE=L is set ...: 1998-10-31
/* Date in page title if DFTITLE=I is set ...: 19981031
```

This example assumes that DTFORM=I applies.

Note:

The DFTITLE parameter has no effect on a user-defined page title as specified with a WRITE TITLE statement.

Reporting Mode and Structured Mode

The following topics are covered below:

- General Information
 - Setting the Programming Mode
 - Functional Differences
 - Closing a Processing Loop in Reporting Mode
 - Closing a Processing Loop in Structured Mode
 - Database Reference
-

General Information

Natural offers two ways of programming: *reporting mode* and *structured mode*.

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

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

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

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

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

Setting the Programming Mode

The default programming mode is set by the Natural administrator. You can change the mode by using the system command GLOBALS:

- **GLOBALS SM=ON** - Structured Mode.
- **GLOBALS SM=OFF** - Reporting Mode.

Functional Differences

The major functional differences between reporting mode and structured mode are summarized below:

- The syntax related to closing loops and functional blocks differs in the two modes.
In structured mode, every loop or logical construct must be explicitly closed with a corresponding END-... statement. Thus, it becomes immediately clear, which loop/logical constructs ends where.
Reporting mode uses (CLOSE) LOOP and DO ... DOEND statements for this purpose.
END-... statements (except END-DEFINE, END-DECIDE and END-SUBROUTINE) cannot be used in reporting mode, while LOOP and DO/DOEND statements cannot be used in structured mode.
- In reporting mode, you can use database fields without having to define them in a DEFINE DATA statement; also, you can define user-defined variables anywhere in a program, which means that they can be scattered all over the program.
In structured mode, *all* data elements to be used have to be defined in one central location (either in the DEFINE DATA statement at the beginning of the program, or in a data area outside the program).

The Natural Statements documentation provides separate syntax diagrams for each mode-sensitive statement.

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

Reporting Mode Example:

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

```

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

```

Structured Mode Example:

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

```

DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 POSITION
END-DEFINE
READ MYVIEW BY PERSONNEL-ID
  DISPLAY NAME BIRTH POSITION
  AT END OF DATA
  SKIP 2
  WRITE / 'LAST SELECTED:' OLD(NAME)
END-ENDDATA
END-READ
END

```

Closing a Processing Loop in Reporting Mode

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

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

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

Example 1 - LOOP:

```
FIND ...
  FIND ...
  ...
  ...
  LOOP (closes inner FIND loop)
LOOP   (closes outer FIND loop)
...
...
```

Example 2 - END:

```
FIND ...
  FIND ...
  ...
  ...
  END (closes all loops and ends processing)
```

Example 3 - SORT:

```
FIND ...
  FIND ...
  ...
  ...
  SORT ... (closes all loops, initiates loop)
  ...
  END (closes SORT loop and ends processing)
```

Closing a Processing Loop in Structured Mode

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

Example 1 - FIND:

```
FIND ...
  FIND ...
  ...
  ...
  END-FIND (closes inner FIND loop)
END-FIND (closes outer FIND loop)
...
```

Example 2 - READ:

```
READ ...
  AT END OF DATA
  ...
  END-ENDDATA
  ...
  END-READ (closes READ loop)
  ...
  ...
  END
```

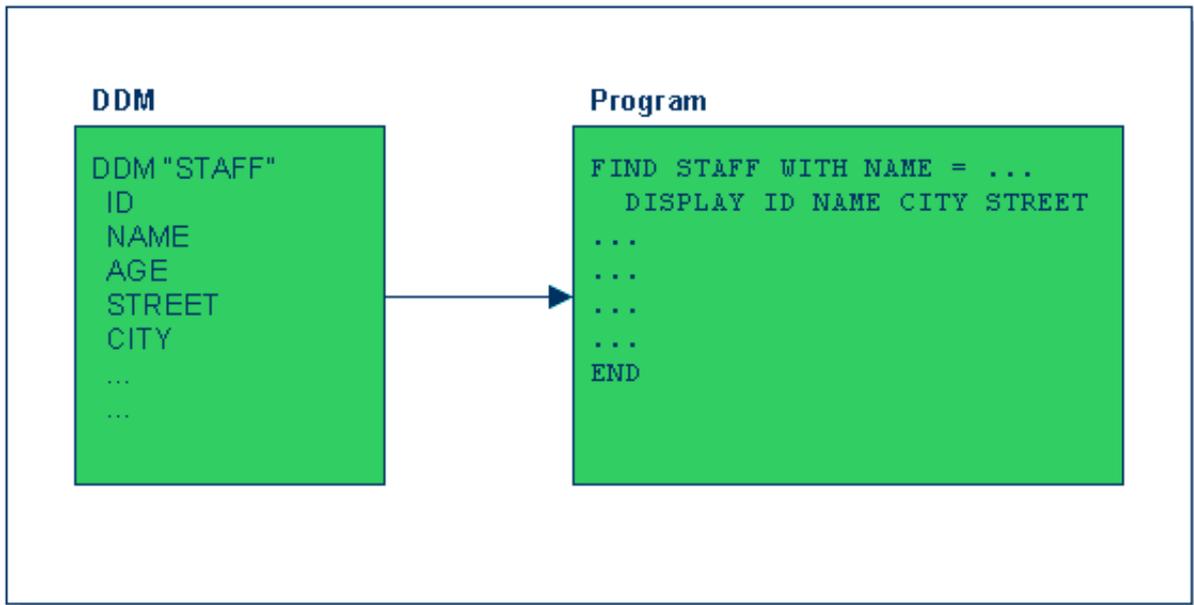
Example 3 - SORT:

```
READ ...
  FIND ...
  ...
  ...
  END-ALL (closes all loops)
  SORT (opens loop)
  ...
  ...
  END-SORT (closes SORT loop)
  END
```

Database Reference

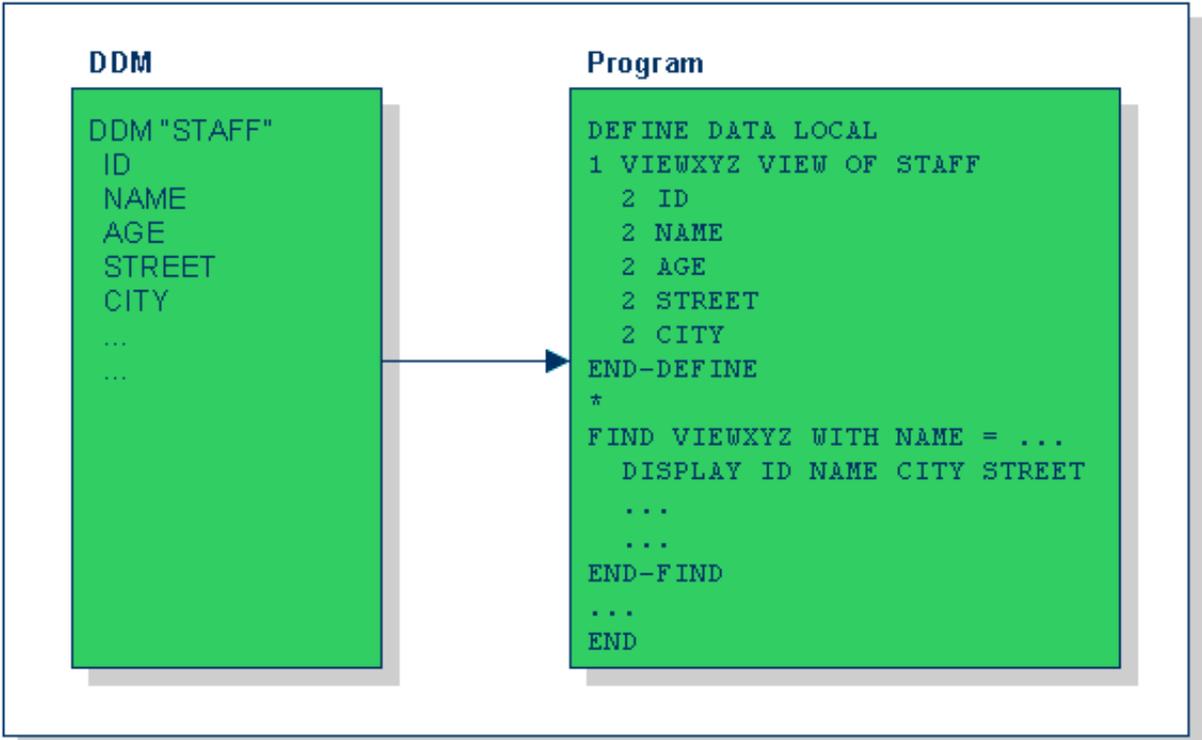
In reporting mode, database fields and DDMs may be referenced without having been defined in a data area.

Reporting Mode:



In structured mode, however, each database field to be used must be specified in a DEFINE DATA statement (as described in the sections Defining Fields and Database Access).

Structured Mode:



Portable Natural Generated Programs

As of Natural Version 5, Natural generated programs (GPs) are portable across the platforms UNIX, OpenVMS and Windows.

This document covers the following topics:

- Compatibility
 - Endian Mode Considerations
 - ENDIAN Parameter
 - Transferring Natural Generated Programs
-

Compatibility

A GP which is cataloged with Natural Version 5 on any Natural-supported UNIX, OpenVMS and Windows platform is then executable with Natural Version 5 on these platforms without recompilation. This feature simplifies the deployment of applications across open systems platforms.

Natural applications generated with Natural Version 4 or Natural Version 3 can be executed with Natural Version 5 without cataloging the applications again (upward compatibility). In this case, the portable GP functionality is not available. To make use of the portable GP and other improvements, cataloging with Natural Version 5 is required.

Command processor GPs and Natural Expert GPs are not portable. The portable GP feature is not available for mainframe platforms. This means that Natural GPs which are generated on mainframe computers are not executable on UNIX, OpenVMS and Windows platforms without recompilation and vice versa.

Endian Mode Considerations

Depending on which UNIX, OpenVMS or Windows platform Natural Version 5 is running, Natural Version 5 will consider the byte order in which multi-byte numbers are stored in the GP. The two byte order modes are called "Little Endian" and "Big Endian".

- "Little Endian" means that the low-order byte of the number is stored in memory at the lowest address, and the high-order byte at the highest address (the little end comes first).
- "Big Endian" means that the high-order byte of the number is stored in memory at the lowest address, and the low-order byte at the highest address (the big end comes first).

The UNIX, OpenVMS and Windows platforms use both endian modes: Intel processors and AXP computers (Natural on Windows or OpenVMS) have "Little Endian" byte order and HP machines have "Big Endian" mode.

Natural Version 5 converts a portable GP automatically into the Endian mode of the execution platform if necessary. This endian conversion is not performed if the GP is already generated in the endian mode of the platform.

ENDIAN Parameter

In order to increase execution performance of portable GPs, the profile parameter ENDIAN has been introduced. ENDIAN determines the Endian mode in which a GP is generated during compilation:

DEFAULT	The endian mode of the machine on which the GP is generated.
BIG	Big endian mode (high order byte first).
LITTLE	Little endian mode (low order byte first).

The values DEFAULT, BIG and LITTLE are alternatives whereby the default value is DEFAULT.

The ENDIAN mode parameter may be set

- as a profile parameter with the Natural Configuration Utility,
- as a start-up parameter,
- as a session parameter or with the GLOBALS command.

Transferring Natural Generated Programs

To make use of the portable GP on different platforms (UNIX, OpenVMS, Windows), the generated Natural objects must be transferred to the target platform or must be accessible from the target platform, for example, via NFS.

Using the Object Handler SYSOBJH SYSOBJH is the recommended way to distribute Natural generated objects or even entire Natural applications. This is done by unloading the objects in the source environment into a work file, transferring the work file to the target environment and loading the objects from the work file.

To deploy your Natural generated objects across open systems platforms

1. Start the Natural Object Handler.

Unload all necessary cataloged objects into a work file of type "portable".

Error messages, if needed, can also be unloaded to the work file.

Important:

The specified work file type must be portable. PORTABLE performs an automatic Endian conversion of a work file when it is transferred to a different machine.

See also Work File Type in the section Define Work File in the Natural Statements documentation.

2. Transfer the work file to the target environment.

Depending on the transfer mechanism (network, CD, diskette, tape, email, download, etc.), the use of a compressed archive such as a ZIP file or encoding with UUENCODE/UUDECODE or similar may make sense. Copying via FTP requires binary transfer type.

Note:

According to the transfer method used, it may be necessary to adjust the record format and attributes or block size of the transferred work file depending on the specific target platform, before continuing with the load function. The work file should have the same format and attributes on the target platform as a work file of the same type that was generated on the target platform itself. Use operating system tools if an adaptation is necessary.

3. Start the Natural Object Handler in the target environment.

Select portable as work file type.

Load the Natural Objects and error messages from the work file.

For more details on how to use the Natural Object Handler, refer to the SYSOBJH SYSOBJH utility documentation.

Beside the aforementioned preferred method, there are various other ways of "moving" or copying single Natural generated objects or even entire libraries or parts thereof, using operating system tools and different transfer methods. In all of these cases, to make the objects executable by Natural, they have to be imported into the Natural system file FUSER so that the FILEDIR.SAG structure is adapted.

This can be done with either of the following methods:

- Using the Import function of the SYSMAIN utility.
- Using the FTOUCH utility.
This utility can be used without entering Natural.

The same applies when direct access is possible from a target platform to the generated objects in the source environment, for example, via NSF, network file server, etc. In this case, the objects have to be imported, too.

Note:

With Natural Version 5.1, it is not yet supported to share a common FNAT or FUSER system file among different open system platforms. The FILEDIR.SAG file is not yet platform-independent.