Natural

Programming Guide

Version 9.3.2

September 2025

ADABAS & NATURAL

This document applies to Natural Version 9.3.2 and all subsequent releases.

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

Document ID: NATUX-NNATPROGRAMMING-932-20250930

Table of Contents

Preface

This document is complementary to the Natural documentation listed in the **Language** section (main documentation overview) and provides basic information essential for writing applications in Natural.

Other Related Documentation:

- *First Steps* - Tutorial with a series of sessions which introduce you to some of the basics of Natural programming.
- *Using Natural* - Tools, commands and customization options for managing Natural objects and applications.
- For information on Natural application programming interfaces (APIs), see: *SYSEXT - Natural Application Programming Interfaces* in the *Utilities* documentation.

Natural Programming Modes	Reporting mode and structured mode
Objects for Natural Application Management	Objects (for example, programs and data areas) used for Natural application management
Function Call	Definition of function calls
Field Definitions	Variable, constant and array definitions
Database Access	Natural access in an Adabas or non-Adabas database
Report Format and Control	Format and control of output report data
Further Programming Aspects	Other programming aspects: **Text notation** **User comments** **Data computation** **Rules for arithmetic assignment** **Conditional processing - IF statement** **Logical condition criteria** **Loop processing** **Control breaks** **Stack processing** **System variables and system functions** **Processing of date information** **Processing of store clock values** **End of statement, program or application** **Processing of application errors** **Invoking Natural subprograms from 3GL programs** **Issuing operating system commands from within a Natural program**
Statements for Internet Access and Parsing	Natural statements for internet, JSON, and XML access

Portable Natural Generated Programs	Programs portable across UNIX, OpenVMS and Windows
Application User Interfaces	Natural character-based application user interfaces for dialog and screen design
Natural Native Interface	Non-Natural applications executing Natural code with C function calls.
NaturalX	Object-based programming with NaturalX components and dedicated Natural statements
Natural Reserved Keywords	List of all keywords reserved for the Natural language
Referenced Example Programs	Natural program examples referenced in the `Programming Guide`

Notation *vrs* **or** *vr*

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

1 About this Documentation

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
`Monospace font`	Identifies service names and locations in the format *folder.subfolder.service*, APIs, Java classes, methods, properties.
Italic	Identifies: Variables for which you must supply values specific to your own situation or environment. New terms the first time they occur in the text. References to other documentation sources.
`Monospace font`	Identifies: Text you must type in. Messages displayed by the system. Program code.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information and Support

Product Documentation

You can find the product documentation on our documentation website at **https://documentation.softwareag.com**.

Product Training

You can find helpful product training material on our Learning Portal at **https://learn.softwareag.com**.

Tech Community

You can collaborate with Software GmbH experts on our Tech Community website at **https://tech-community.softwareag.com**. From here you can, for example:

- Browse through our vast knowledge base.
- Ask questions and find answers in our discussion forums.
- Get the latest Software GmbH news and announcements.
- Explore our communities.
- Go to our public GitHub and Docker repositories at **https://github.com/softwareag** and **https://hub.docker.com/publishers/softwareag** and discover additional Software GmbH resources.

Product Support

Support for Software GmbH products is provided to licensed customers via our Empower Portal at **https://empower.softwareag.com**. Many services on this portal require that you have an account. If you do not yet have one, you can request it at **https://empower.softwareag.com/register**. Once you have an account, you can, for example:

- Download products, updates and fixes.
- Search the Knowledge Center for technical information and tips.
- Subscribe to early warnings and critical alerts.
- Open and update support incidents.
- Add product feature requests.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

I Natural Programming Modes

2 Natural Programming Modes

This chapter describes the two programming modes offered by Natural.

Note: Generally, it is recommended to use structured mode exclusively, because it provides for more clearly structured applications. Therefore, the explanations and examples in the *Programming Guide* usually refer to structured mode only.

Purpose of Programming Modes

Natural offers two ways of programming:

- Reporting Mode
- Structured Mode

Reporting Mode

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

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

Structured Mode

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

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

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

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

Setting/Changing the Programming Mode

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

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

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

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

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

Functional Differences

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

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

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

Syntax Related to Closing Loops and Functional Blocks

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

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

Reporting Mode Example:

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

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

Structured Mode Example:

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

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

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

```
END-READ
END
```

Closing a Processing Loop in Reporting Mode

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

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

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

Example 1 - LOOP:

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

Example 2 - END:

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

Example 3 - SORT:

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

Closing a Processing Loop in Structured Mode

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

Example 1 - FIND:

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

Example 2 - READ:

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

Example 3 - SORT:

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

Location of Data Elements in a Program

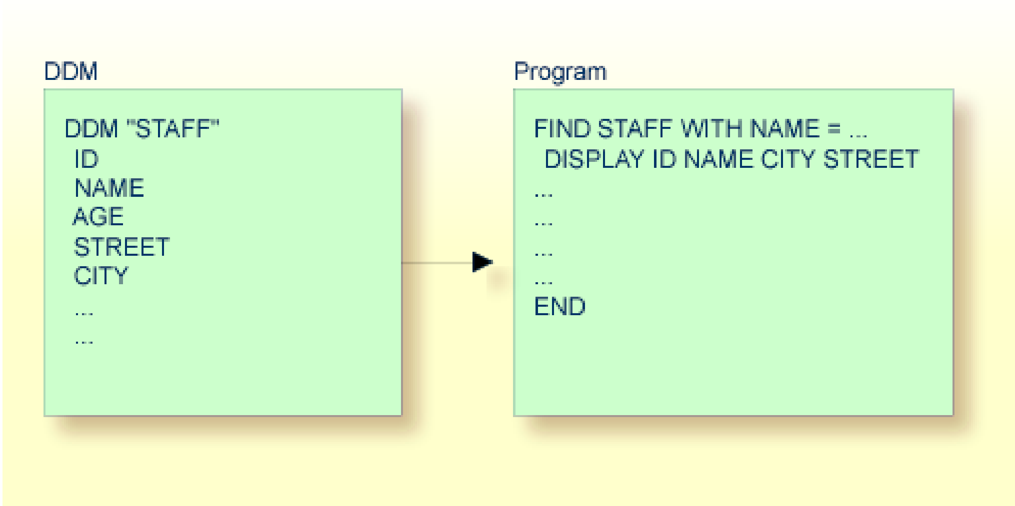
In reporting mode, you can use database fields without having to define them in a `DEFINE DATA` statement; also, you can define user-defined variables anywhere in a program, which means that they can be scattered all over the program.

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

Database Reference

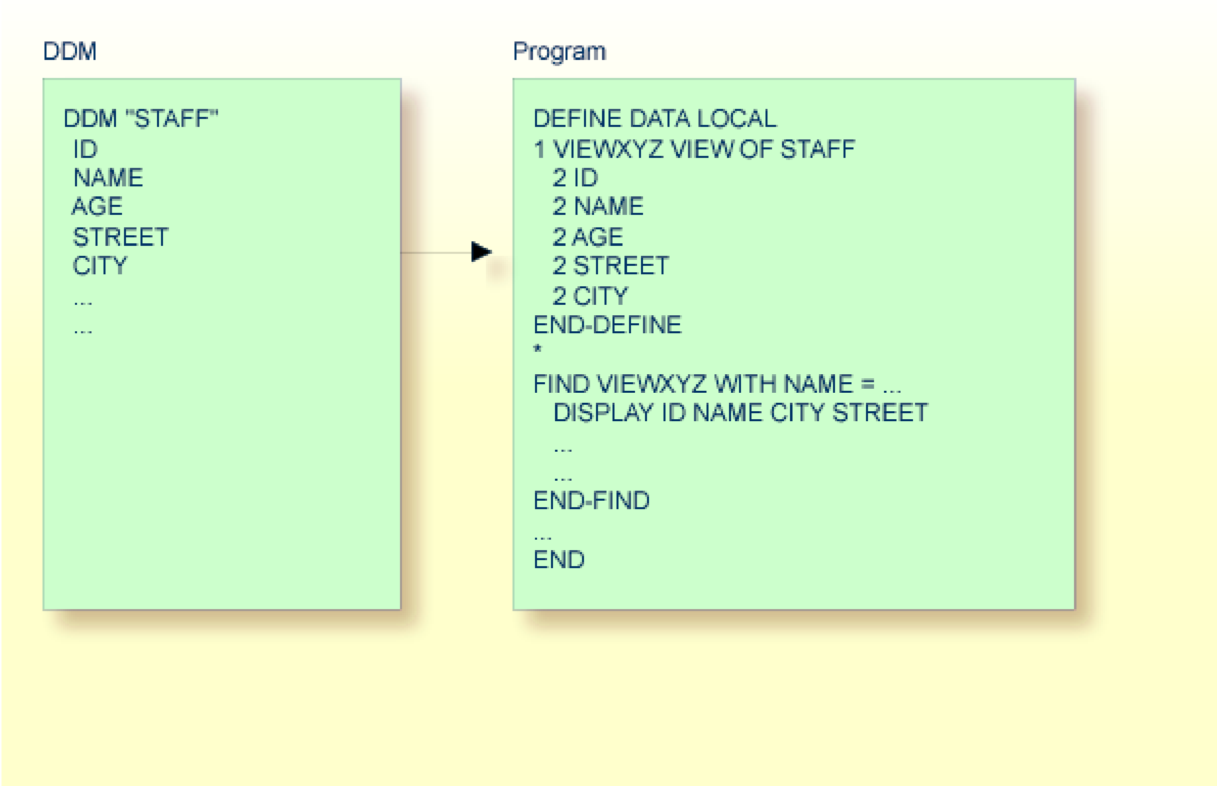
Reporting Mode:

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



Structured Mode:

In structured mode, each database field to be used must be specified in a `DEFINE DATA` statement as described in ***Field Definitions*** and *Database Access*.


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

II Objects for Natural Application Management

This document describes the objects available to build, maintain and control applications with Natural.

Note: The Natural program, data area and map editor have been disabled in your environment by default. For more information, see *NaturalONE as the Default Development Environment* in the *Editors* documentation.

The following table is an overview of Natural and non-Natural objects, their use, and the Natural editors or utilities provided to create and maintain them.

Object Type	Use	Editor or Utility
Data Areas: Local Data Area Global Data Area Parameter Data Area	Variable and parameter definitions for other Natural objects	Data Area Editor
Data Definition Module	Natural data definitions for database file access	DDM Editor
Programs and Subordinate Routines: Program Subroutine Subprogram Function	Main programs, invoked routines, and functions	Program Editor
Helproutine	Help requests for applications	
Copycode	Source code for repeated use in other Natural objects	
Text	Documentation for Natural objects	
Class	Component-based applications	Program Editor
Map	Character-based screen layouts	Map Editor

Object Type	Use	Editor or Utility
Adapter and GUI Layout	Complex graphical user interfaces and rich GUI pages generated from external page layout	Natural for Ajax Developer (see the *Natural for Ajax* documentation)
Dialog	Event-driven applications	n/a (storage and display only)
Resource	Non-Natural objects such as HTML files or bitmaps	n/a (storage and display only)
Error Message	Natural system and user-defined messages	SYSERR Utility
Command Processor	Command-driven navigation	SYSNCP Utility

Related Topic:

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

3 Data Areas

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

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

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

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

Natural supports three types of data area:

Local Data Area (LDA)

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

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

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

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

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

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

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

Example 2 - Fields Defined in a Separate Data Area:

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

Program:

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

Local Data Area `LDA39`:

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

Global Data Area (GDA)

The following topics are covered below:

- Creating and Referencing a Global Data Area
- Creating and Deleting GDA Instances
- Data Blocks

Creating and Referencing a Global Data Area

A global data area (GDA) is created and modified with the data area editor. For further information, refer to *Data Area Editor* in the *Editors* documentation.

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

Note: Using a GDA named `COMMON` for startup:

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

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

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

Creating and Deleting GDA Instances

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

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

A new GDA instance is created if the following applies:

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

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

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

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

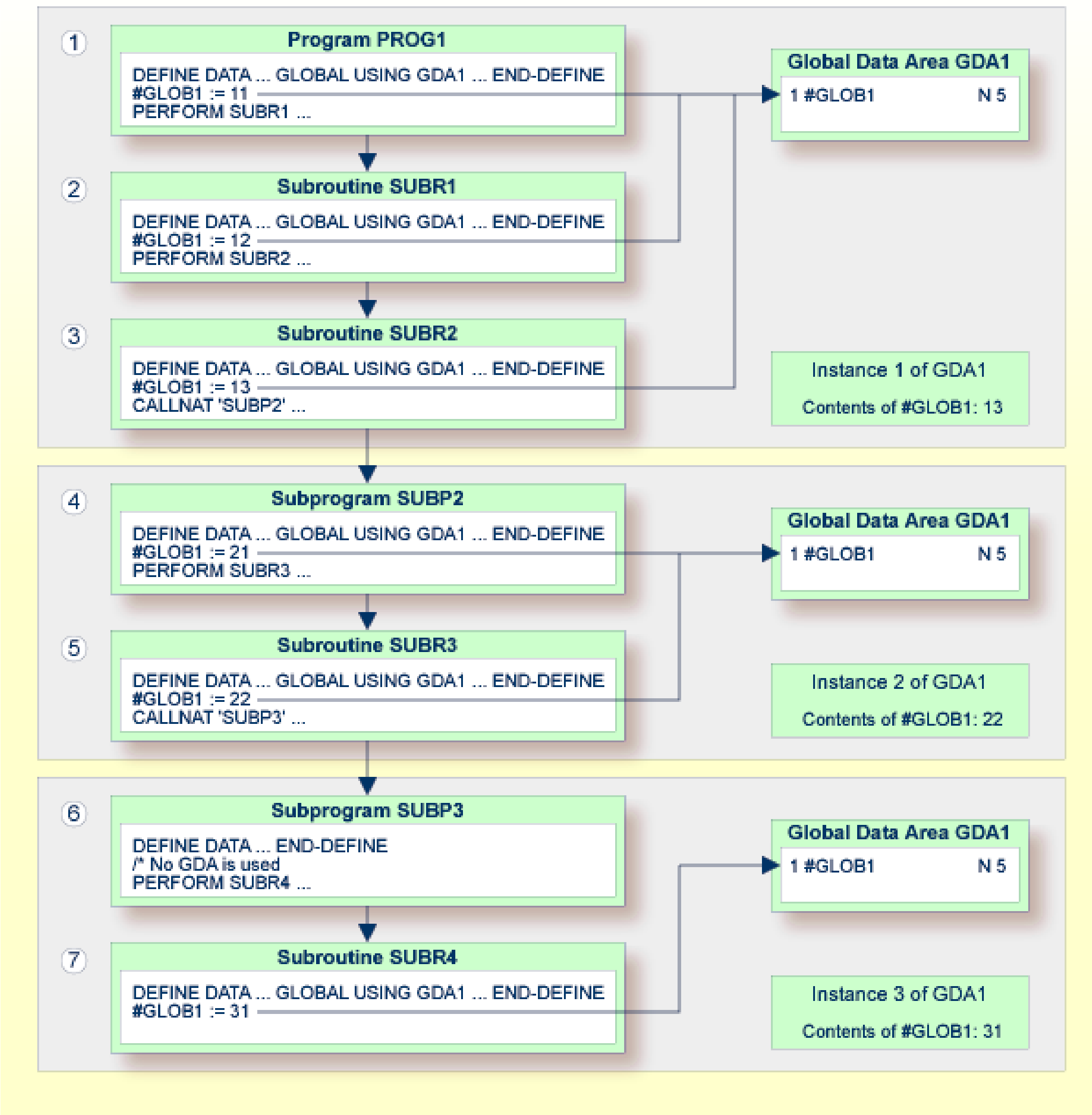
- The next `LOGON` is performed.
- Another GDA is referenced on the same level (levels are described later in this section).
- A `RELEASE VARIABLES` statement is executed. In this case, the data values in a GDA instance are reset either when a program at the level 1 finishes executing, or if the program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

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

Sharing GDA Instances

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

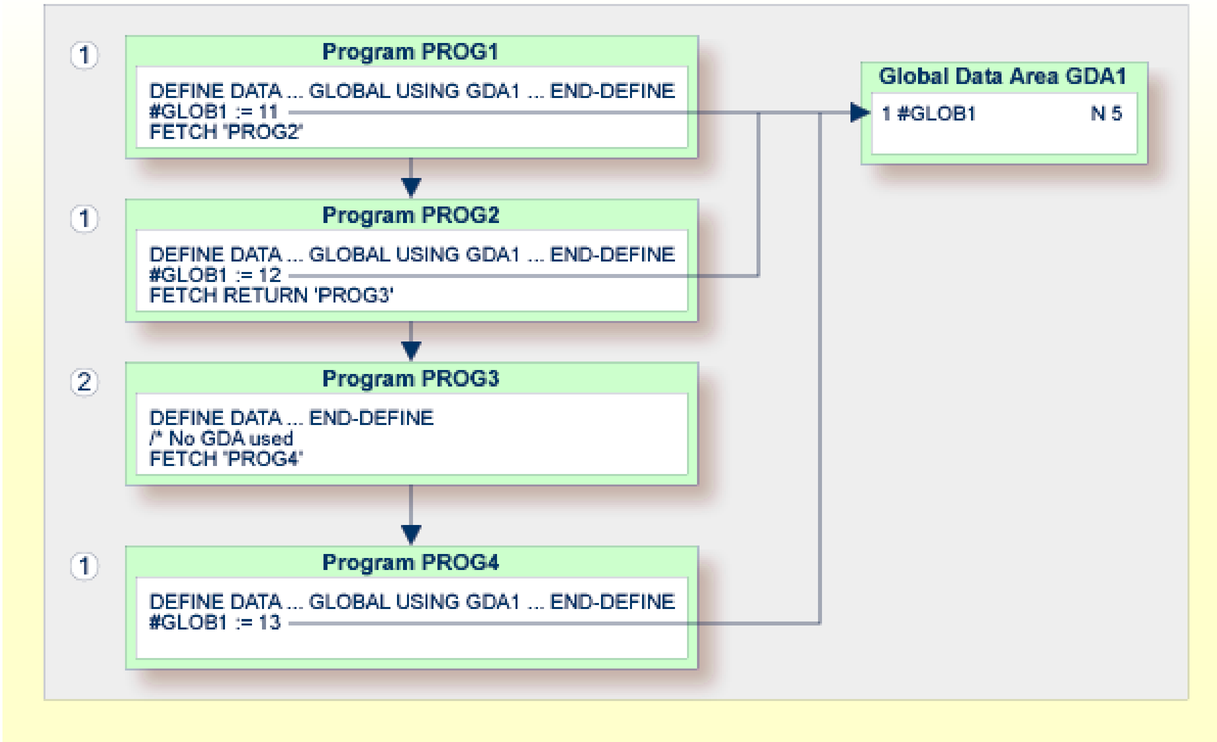
The graphic below shows three GDA instances of `GDA1` and the final values each GDA instance is assigned by the data element `#GLOB1`. The numbers 1 to 7 indicate the hierarchical levels of the objects.



Using FETCH or FETCH RETURN

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

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



Using FETCH with different GDAs

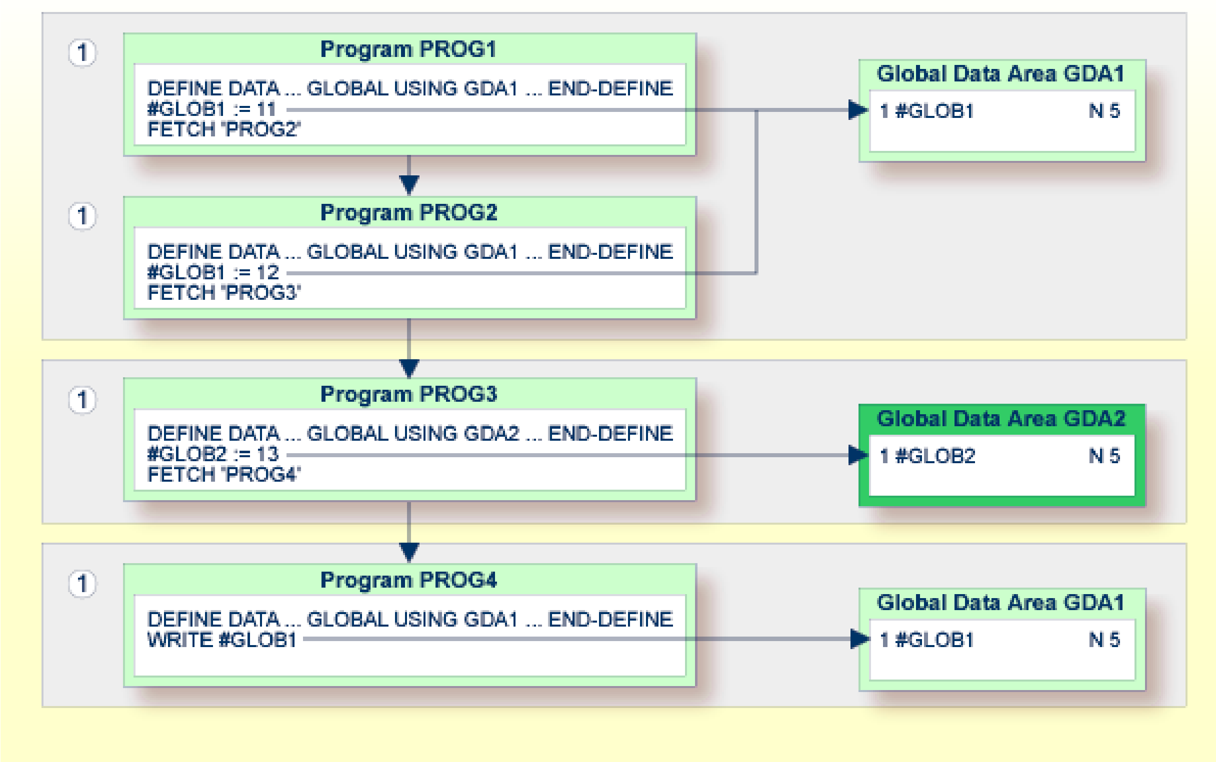
The graphic below illustrates that if a program uses the `FETCH` statement to invoke another program that references a different GDA, the current instance of the GDA (here: `GDA1`) referenced by the invoking program is deleted. If this GDA is then referenced again by another program, a new instance of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

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

The number ① indicates the hierarchical level of the objects.

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

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



Data Blocks

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

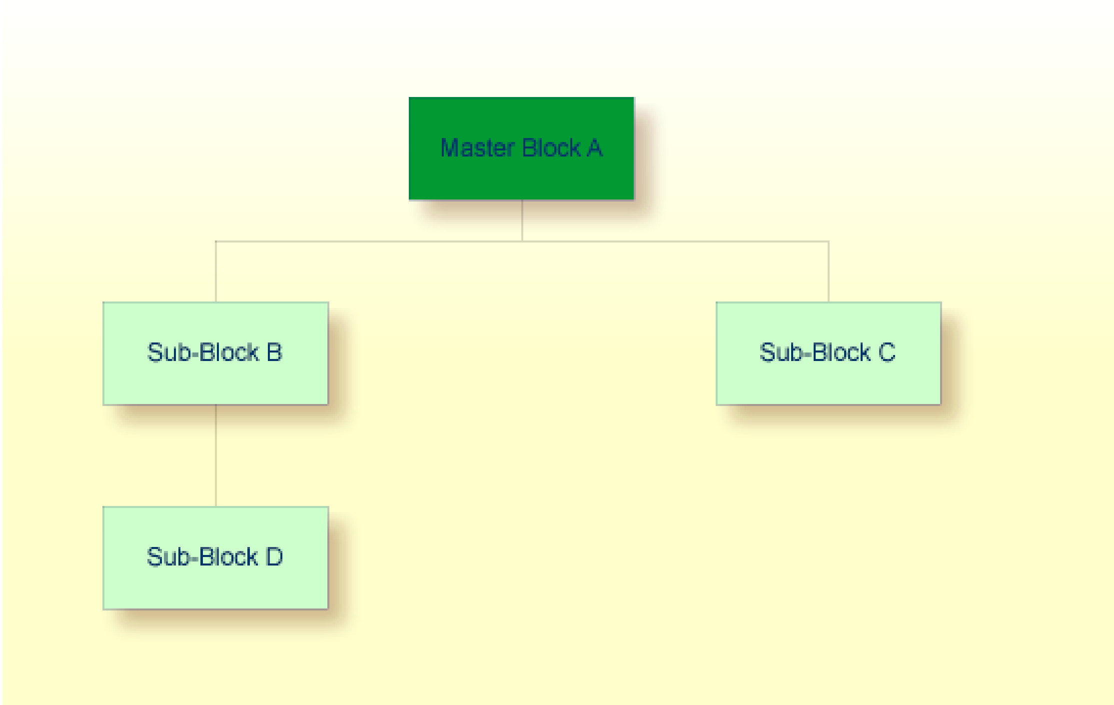
The following topics are covered below:

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

Example of Data Block Usage

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

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



Defining Data Blocks

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

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

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

Example:

Global Data Area `G-BLOCK`:

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

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

Program 1:

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

Program 2:

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

Program 3:

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

Program 4:

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

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

Block Hierarchies

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

Program 1:

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

Program 2:

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

Program 3:

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

Explanation:

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

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

Parameter Data Area (PDA)

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

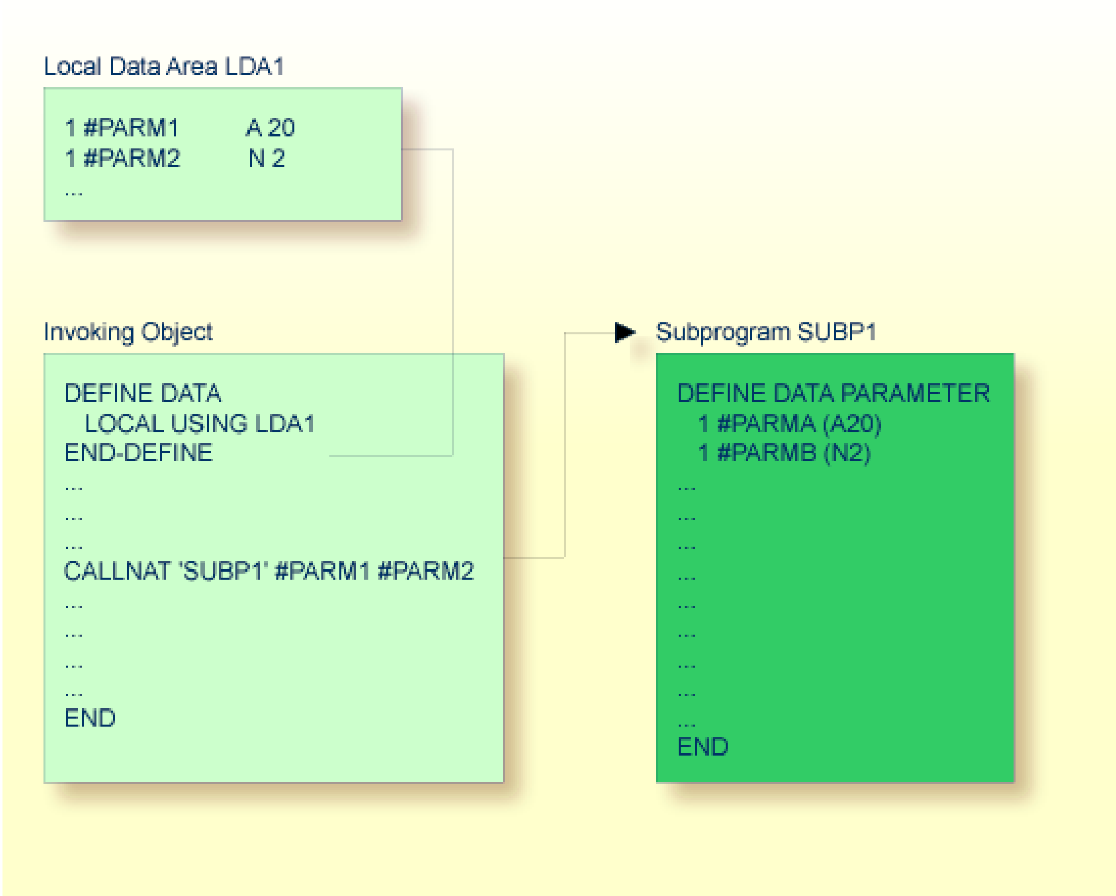
These parameters must be defined with a `DEFINE DATA PARAMETER` statement in the subprogram:

- 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 (PDA), with the `DEFINE DATA PARAMETER` statement referencing that PDA.

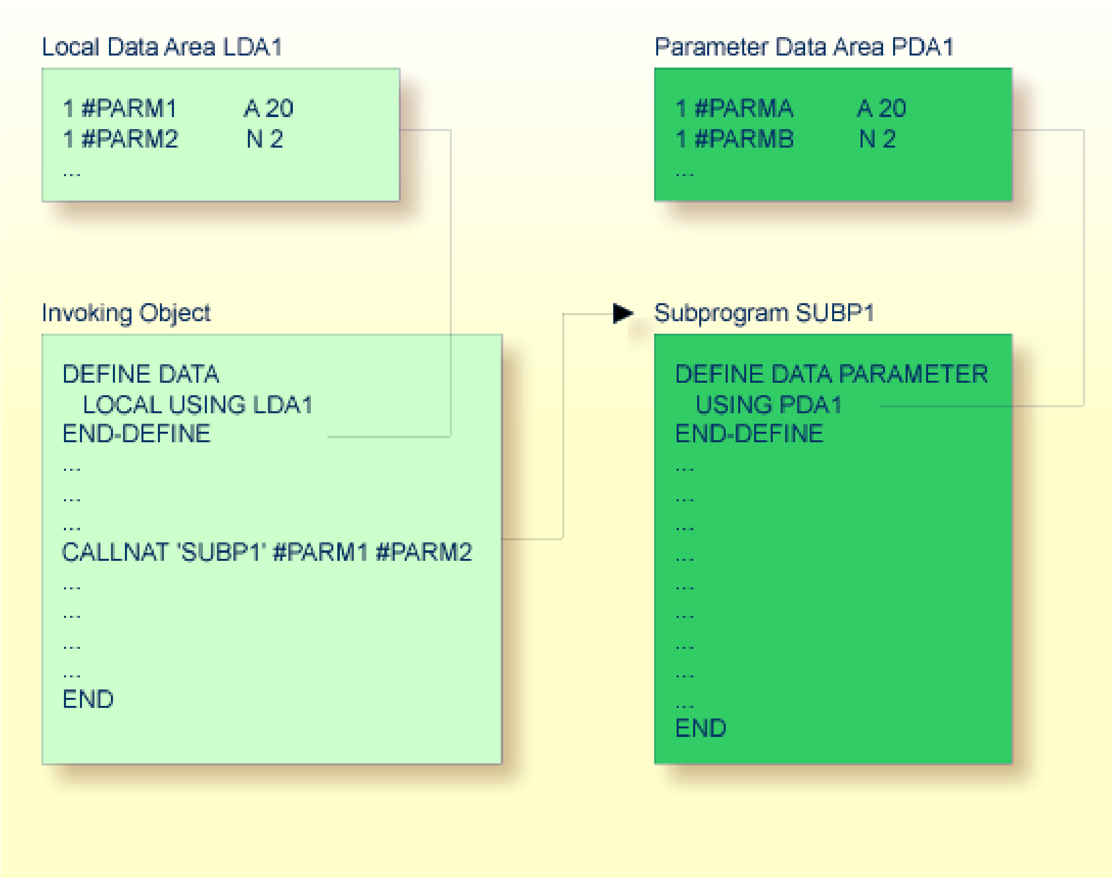
The following topics are covered below:

- Matching Format Specification of Array Dimensions

Parameters Defined within DEFINE DATA PARAMETER Statement



Parameters Defined in Parameter Data Area



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

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

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

To guarantee that the data element definitions used in the invoking program are identical to the data element definitions used in the subprogram or external subroutine, you can specify a PDA in a `DEFINE DATA LOCAL USING` statement. By using a PDA as an LDA you can avoid the extra effort of creating an LDA that has the same structure as the PDA.

Matching Format Specification of Array Dimensions

When you pass an array as a parameter, its dimension must match the dimension of the array specified in the `DEFINE DATA PARAMETER` statement of the invoked subprogram or subroutine. A dimension mismatch generates an error even if the number of occurrences matches.

Example:

Called subprogram `SUB`:

```
DEFINE DATA PARAMETER
1 B (A5/1:5)
END-DEFINE
...
```

Calling program with NAT0937 compiler error:

```
DEFINE DATA LOCAL
1 A (A5/1:1,1:5)
END-DEFINE
CALLNAT 'SUB' A(1,*)
...
```

Calling program without compiler error:

```
DEFINE DATA LOCAL
1 A (A5/1:5)
END-DEFINE
CALLNAT 'SUB' A(*)
```

4 Data Definition Module (DDM)

A data definition module (DDM) contains the description of a database file and the fields therein. Natural requires this description to access the data stored in the file from a Natural program.

For further information, see ***Data Definition Modules - DDMs*** and ***Natural and Database Access***.

Related Topics:

- ***Accessing Data in an Adabas Database***
- ***Accessing Data in an SQL Database***
- *Protecting DDMs On Linux and Windows* in the *Natural Security* documentation

5 Programs and Subordinate Routines

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

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

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.

Modular Application Structure

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

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

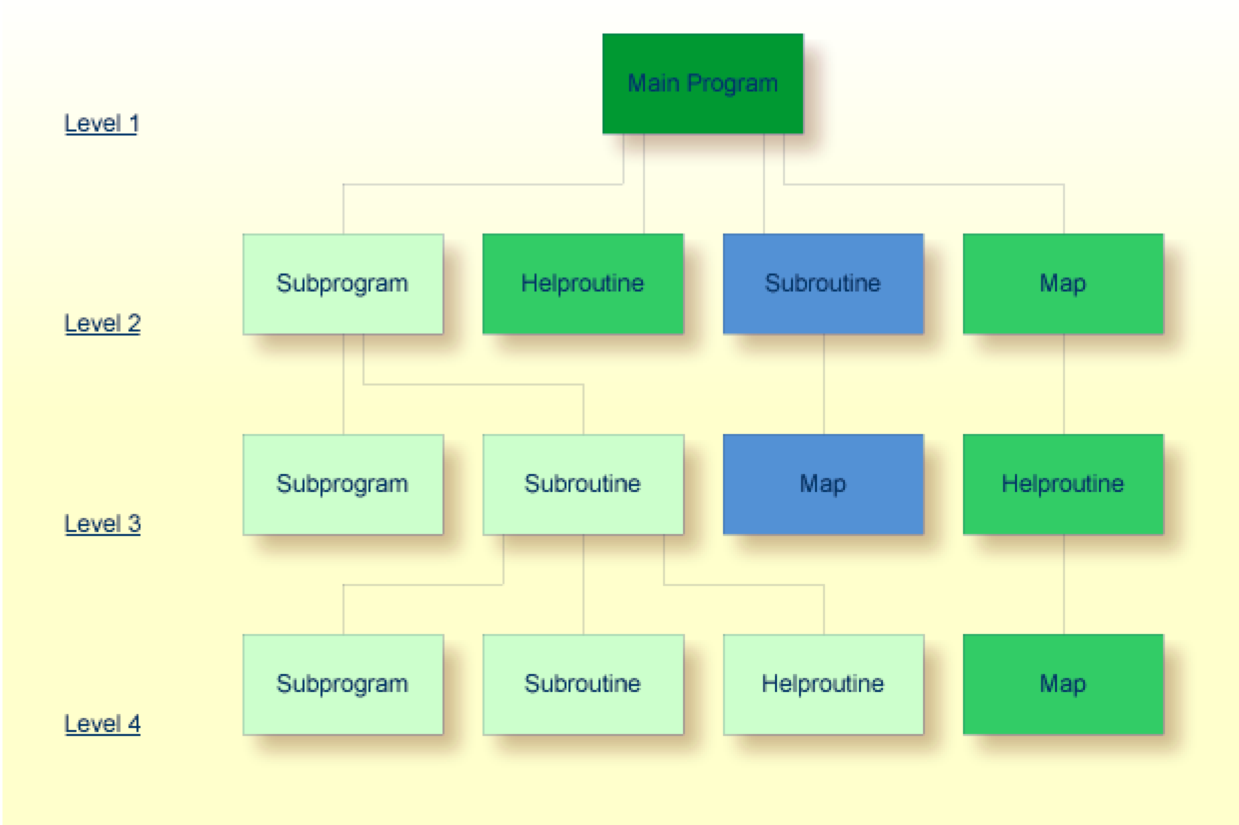
Multiple Levels of Invoked Objects

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

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

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

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



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

Processing Flow when Invoking a Subordinate Routine

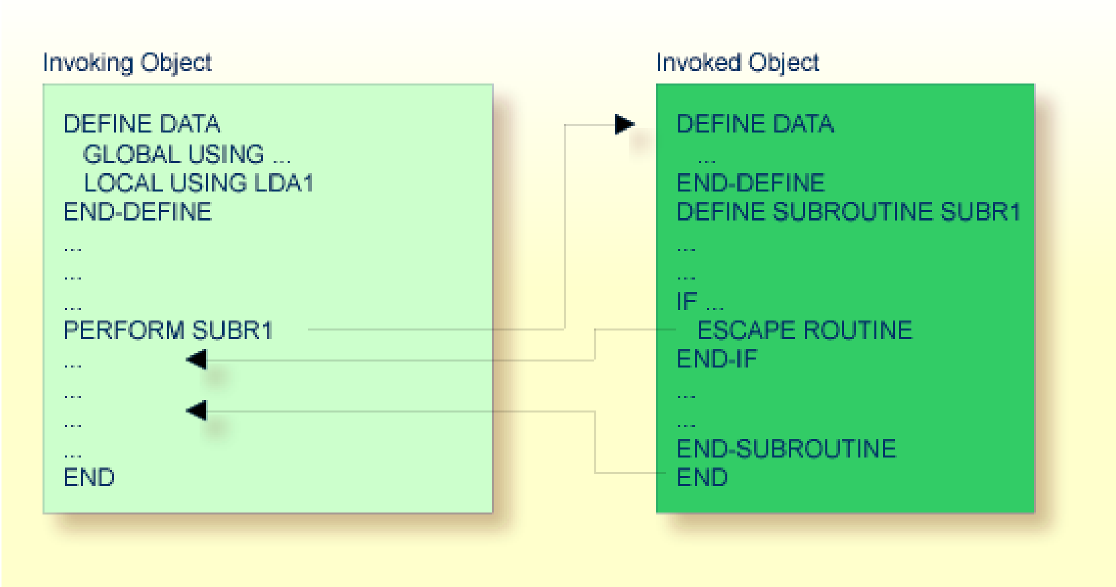
When the `CALLNAT`, `PERFORM` or `FETCH RETURN` statement or the function call that invokes a subordinate routine - a subprogram, an external subroutine, a program or a function respectively - is executed, the execution of the invoking object is suspended and the execution of the subordinate routine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

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

In the case of a function call, processing of the invoking object will then continue with the statement that contains the function call.

Example:



Program

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

- To catalog (compile) and execute a source program, you use the system command `RUN`.
- To execute a program that already exists as a cataloged object, you use the system command `EXECUTE`.

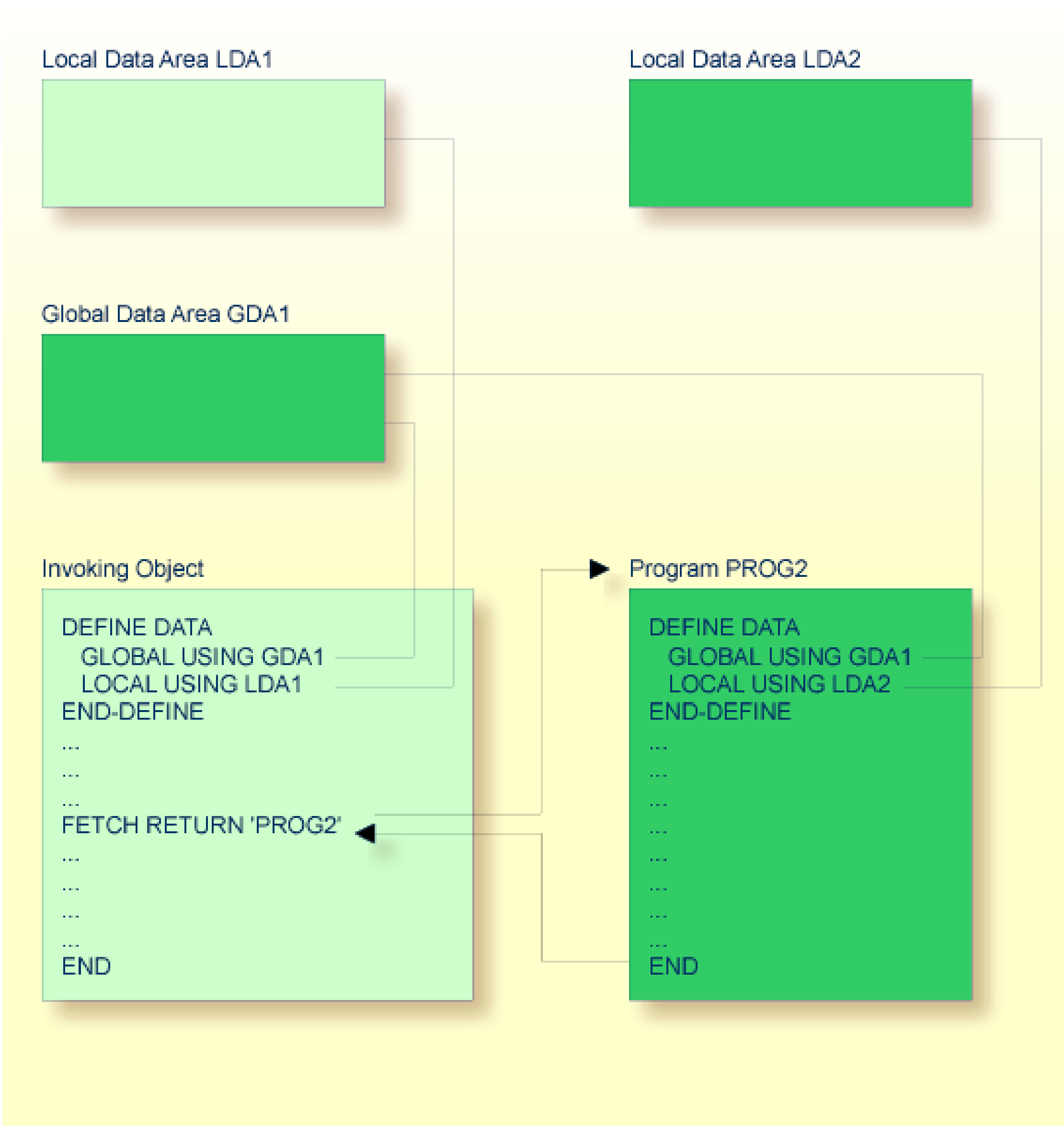
A program can also be invoked from another object with a `FETCH` or `FETCH RETURN` statement. The invoking object can be another program, a **subroutine**, **subprogram**, **function**, a **helproutine** or a processing rule in a map.

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

The following topics are covered below:

- Program Invoked with FETCH

Program Invoked with FETCH RETURN

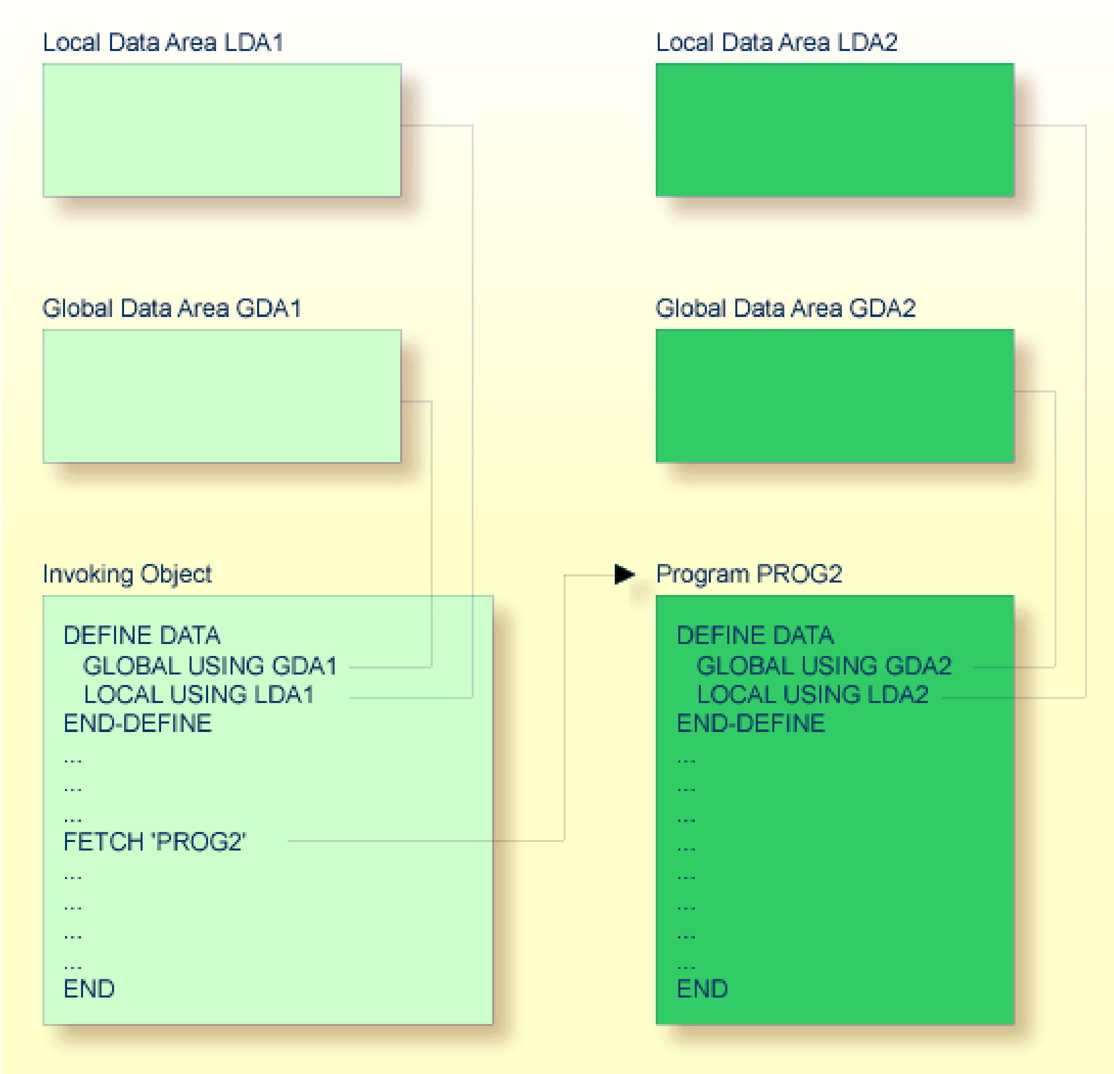


A program invoked with `FETCH RETURN` can access the **global data area** (GDA) used by the invoking object.

In addition, every program can have its own **local data area** (LDA) which defines the fields that are to be used within the program only. Furthermore, a program can access application-independent variables (AIVs); see *Defining Application-Independent Variables* in the *Statements* documentation for details.

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

Program Invoked with FETCH



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

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

Subroutine

Typically, a subroutine implements functionality that is used by different objects in an application.

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

A subroutine is invoked with a `PERFORM` statement.

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

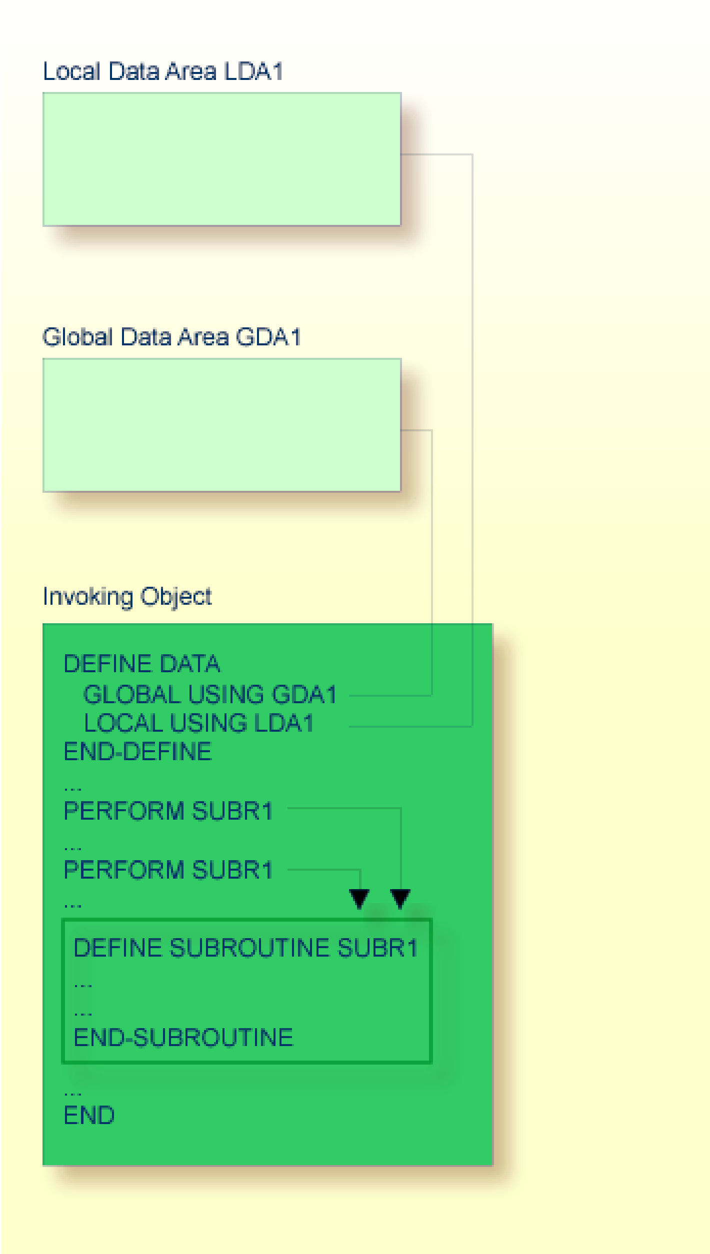
- **Inline Subroutine**
  An inline subroutine is defined within the object which contains the `PERFORM` statement that invokes it.
- **External Subroutine**
  An external subroutine is defined in a separate object - of type subroutine - outside the object which invokes it.

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

The following topics are covered below:

- Data Available to an External Subroutine

Inline Subroutine

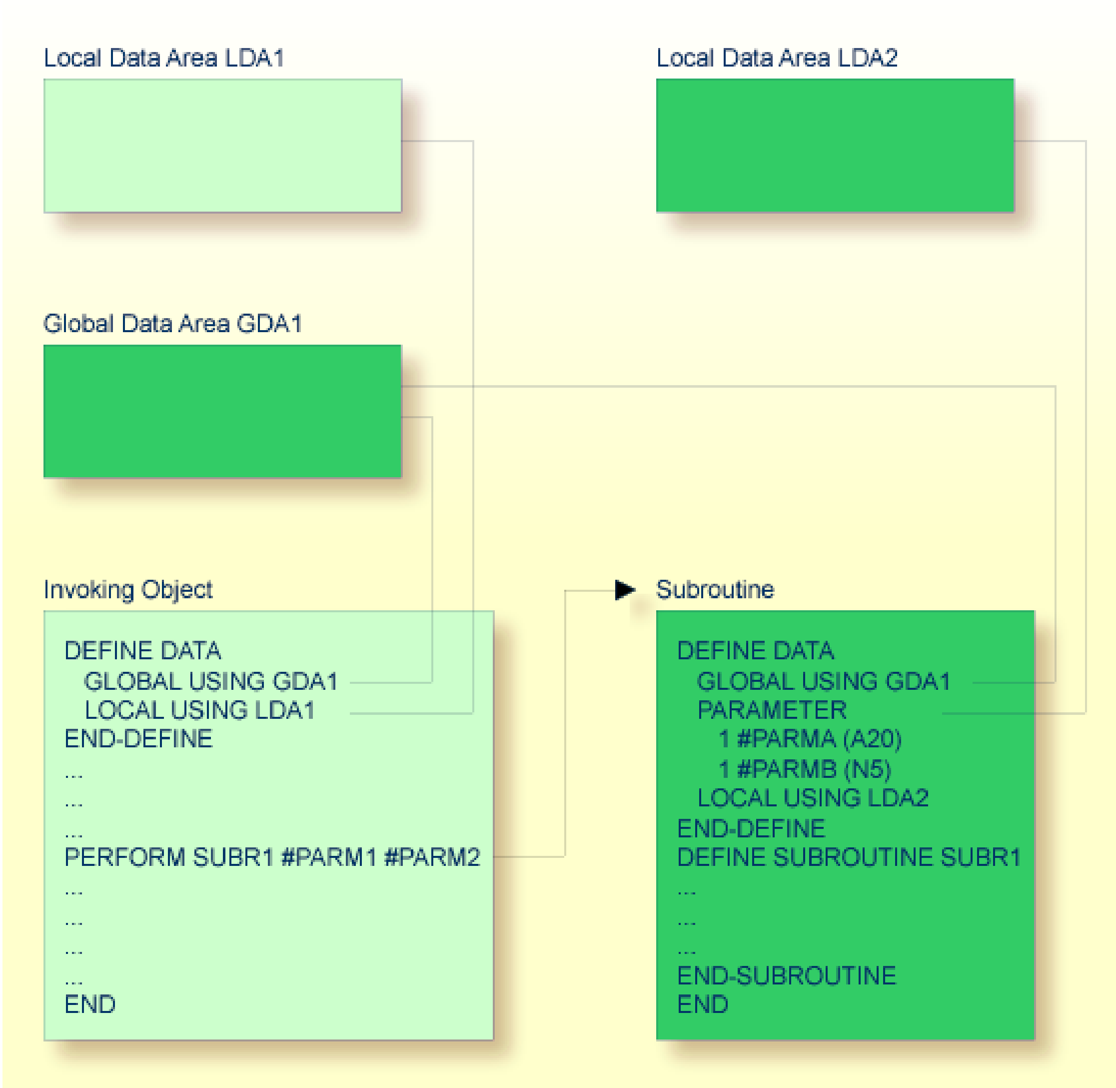


An inline subroutine can be contained within an object of type program, function, subprogram, subroutine or helproutine.

Data Available to an Inline Subroutine

An inline subroutine has access to all data fields within the object in which it is contained.

External Subroutine



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

Data Available to an External Subroutine

An external subroutine can access the **global data area** (GDA) 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** (PDA) used by the subroutine.

In addition, an external subroutine can have its **local data area** (LDA) 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** (GDA).

An external subroutine can also access application-independent variables (AIVs); see *Defining Application-Independent Variables* in the *Statements* documentation for details.

Subprogram

Typically, a subprogram implements functionality that is used by different objects in an application.

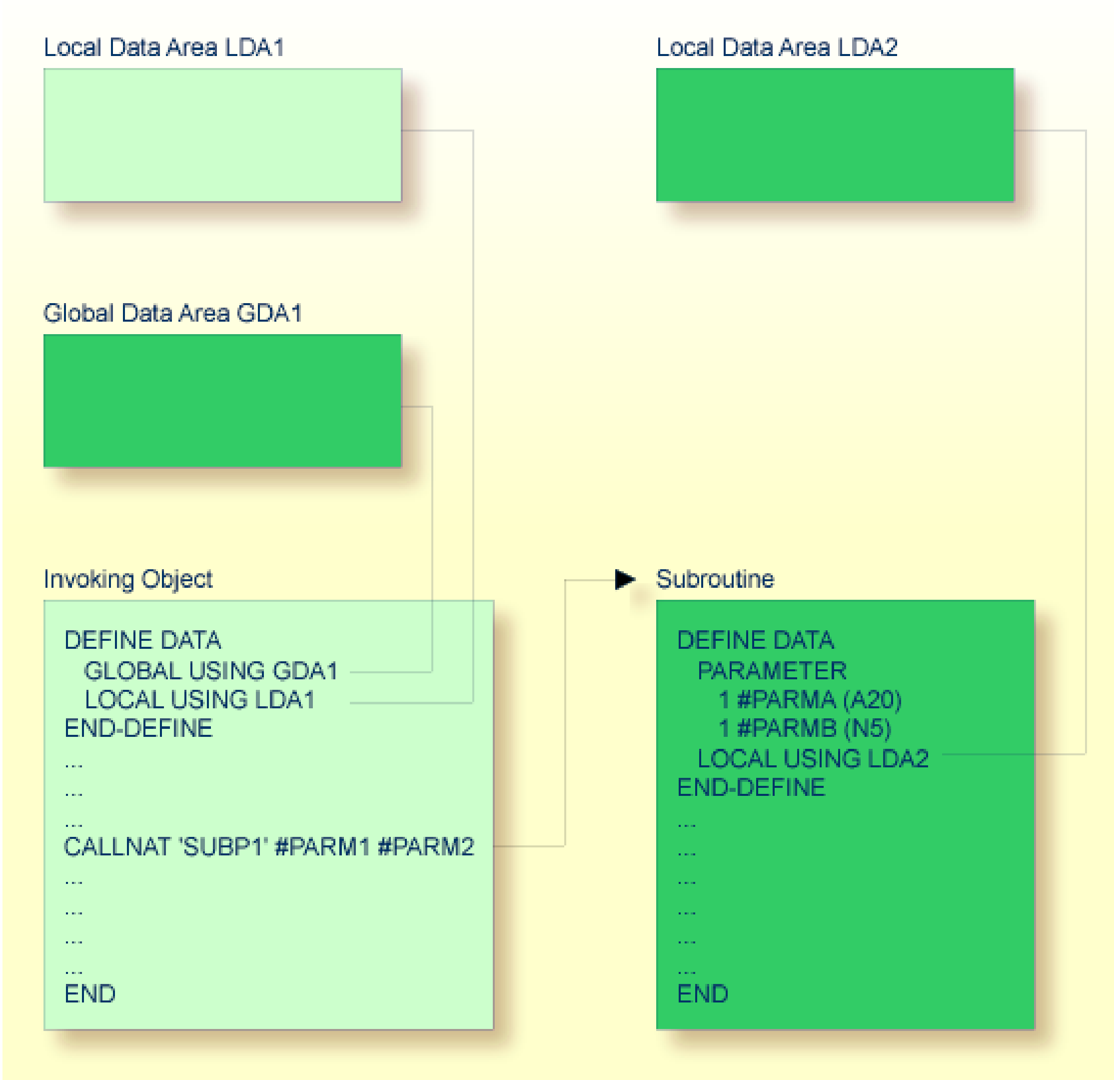
A subprogram cannot be executed by itself. It must be invoked from another object. The invoking object can be a program, function, subprogram, subroutine or helproutine.

A subprogram is invoked with a `CALLNAT` statement.

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

Data Available to a Subprogram

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



In addition, a subprogram can have its own **local data area** (LDA) 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 (GDA) to be shared with the subroutine/helproutine.

Furthermore, a subprogram can access application-independent variables (AIVs); see *Defining Application-Independent Variables* in the *Statements* documentation for details.

Function

Typically, a function implements functionality that is used by different objects in an application.

A function provides user-defined functionality as opposed to the standard system functions (see the relevant documentation) supplied by Natural.

A function returns a result value that is used by the invoking object. The result value is computed from the data available to the function.

A function object contains a single function defined with a `DEFINE FUNCTION` and an `END` statement.

A function itself is invoked by a **function call**.

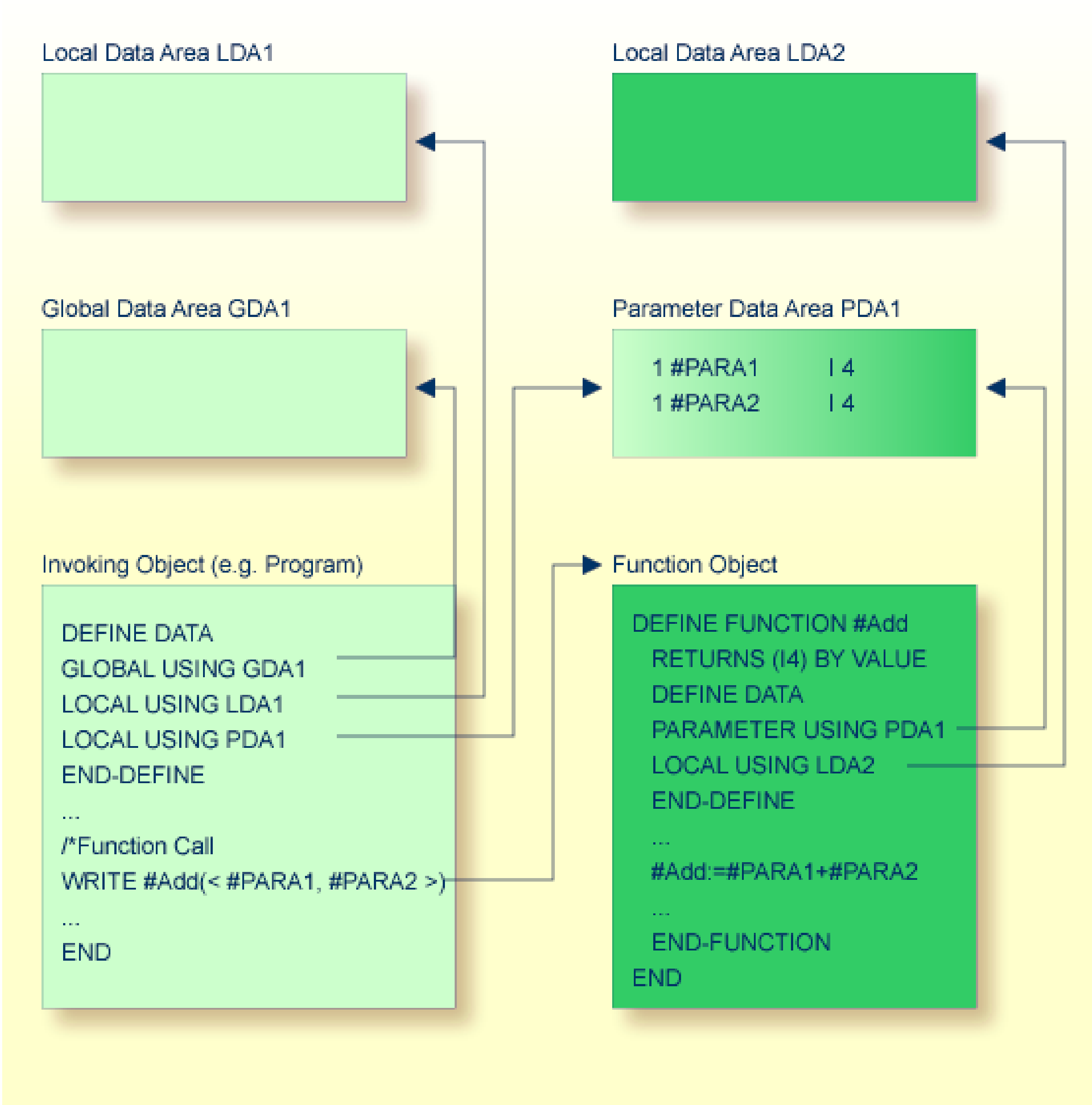
Data Available to a Function

With the function call, parameters can be passed from the invoking object to the function. These parameters are the only data available to the function from the invoking object. They must be defined in the `DEFINE FUNCTION` statement.

In addition, a function can have its own **local data area** (LDA) in which the fields to be used within the function are defined. However, a function cannot have its own **global data area** (GDA).

A function can also access application-independent variables (AIVs); see *Defining Application-Independent Variables* in the *Statements* documentation for details.

If required, you can define the result and parameter layouts for the object calling a function by using the `DEFINE PROTOTYPE` statement.



For further information, see the section ***Function Call***.

Comparison of External Subroutine, Subprogram and Function

This section is a summarized feature comparison between external subroutines, subprograms and functions.

This is the same for all of them:

- The programming code forming the routine logic is coded in a separate object which is stored in a Natural library.
- Parameters are defined in the object using a `DEFINE DATA PARAMETER` statement.

The differences between an external subroutine, a subprogram and a function are indicated in the following table:

Subject	External Subroutine	Subprogram	Function
Maximum length of name	32 characters	8 characters	32 characters
Use of **global data area** (GDA)	Shares a GDA with its caller	Creates an **instance** of a GDA	A GDA is not allowed.
Check of format/length of passed parameters against definition in called object at compile time	Only checked if the compiler option `PCHECK` is set to `ON`	Only checked if the compiler option `PCHECK` is set to `ON`	Only checked if a cataloged function object exists at compile time
Invoked by	Invoked by the `PERFORM` statement	Invoked by the `CALLNAT` statement	Invoked by a **function call** A function call can be used in statements instead of read-only operands; a function call can also be used as a statement.
Determination of the object to be called at compile/execution time	Determined at compile time	Determined at compile or execution time depending on the operand used for the `CALLNAT` statement	Determined at compile or execution time depending on the operand used for the function call
Use of result value in a statement	A result value must be assigned to a parameter to be used as an operand in a statement.	A result value must be assigned to a parameter to be used as an operand in a statement.	The result of a function call is used as an operand in the statement that contains the function call.

The following examples compare a function call with a subprogram call:

- Example of a Function Call

- Example of a Subprogram Call

Example of a Function Call

The following example shows a program calling a function, and the function definition created with a DEFINE FUNCTION statement.

Program Calling the Function

```
** Example 'FUNCAX01': Calling a function  (Program)
************************************************************************
*
WRITE 'Function call' F#ADD(< 2,3 >)  /* Function call.
                                      /* No temporary variables needed.
*
END
```

Definition of Function F#ADD

```
** Example 'FUNCAX02': Calling a function  (Function)
************************************************************************
DEFINE FUNCTION F#ADD
  RETURNS #RESULT (I4)
  DEFINE DATA PARAMETER
    1 #SUMMAND1 (I4) BY VALUE
    1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE
  /*
  #RESULT := #SUMMAND1 + #SUMMAND2
  /*
END-FUNCTION
*
END ↩
```

Example of a Subprogram Call

To implement the same functionality as shown in the example of a function call by using a subprogram call instead, you need to specify temporary variables.

Program Calling the Subprogram

The following example shows a program calling a subprogram, involving the use of a temporary variable.

```
** Example 'FUNCAX03': Calling a subprogram  (Program)
************************************************************************
DEFINE DATA LOCAL
  1 #RESULT (I4) INIT <0>
END-DEFINE
*
CALLNAT 'FUNCAX04' #RESULT 2 3    /* Result is stored in #RESULT.
*
WRITE '=' #RESULT                 /* Print out the result of the
                                  /* subprogram.
*
END
```

Called Subprogram FUNCAX04

```
** Example 'FUNCAX04': Calling a subprogram  (Subprogram)
************************************************************************
DEFINE DATA PARAMETER
  1 #RESULT   (I4) BY VALUE RESULT
  1 #SUMMAND1 (I4) BY VALUE
  1 #SUMMAND2 (I4) BY VALUE
END-DEFINE
*
#RESULT := #SUMMAND1 + #SUMMAND2
*
END ↩
```

6 Helproutine

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

Invoking Help

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

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

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

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

```
SET KEY PF1=HELP
```

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

Specifying Helproutines

A helproutine may be specified:

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

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

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

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

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

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

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

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

Programming Considerations for Helproutines

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

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

Passing Parameters to Helproutines

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

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

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

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

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

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

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

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

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

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

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

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

Note: Numeric constant parameters are internally represented in packed form (format P). For further information, see the *Programming Guide* > *User-Defined Constants* > ***Numeric Constants***.

Equal Sign Option

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

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

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

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

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

Array Indices

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

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

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

The corresponding helproutine starts with:

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

Help as a Window

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

```
*******************************************************************************
                           PERSONNEL INFORMATION
PLEASE ENTER NAME: ?_________________
PLEASE ENTER CITY: __________________
                   +----------------------------+
                   !                            !
                   ! Type in the name of an     !
                   ! employee in the first      !
                   ! field and press ENTER.     !
                   ! You will then receive      !
                   ! a list of all employees    !
```

```
                   ! of that name.              !
                   !                            !
                   ! For a list of employees    !
                   ! of a certain name who      !
                   ! live in a certain city,    !
                   ! type in a name in the      !
                   ! first field and a city     !
                   ! in the second field        !
                   ! and press ENTER.           !
*******************!                            !*******************************
                   +----------------------------+
```

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

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

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

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

7 Copycode

This chapter describes the advantages and the use of copycode.

Use of Copycode

An object of type copycode contains 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 a copycode object instead of coding the statement block several times. This reduces the coding effort and also ensures that the blocks are really identical.

Processing of Copycode

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

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

Attention:

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

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

8 Text

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

Use of Text Objects

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

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

Writing Text

You write the text using the program editor.

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

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

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

9 Class

Classes are used to apply an object based programming style.

For details, refer to the ***NaturalX*** section of the *Programming Guide*.

10 Map

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

Benefits of Using Maps

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

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

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

Types of Maps

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

The following types of maps exist:

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

The object type “map” comprises

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

Related Topics:

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

Creating Maps

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

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

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

Related Topics:

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

Starting/Stopping Map Processing

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

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

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

11 Adapter

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

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

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

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

12 Dialog

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

Note: Dialogs cannot be created or modified with Natural for Mainframes or Natural for Linux, but can be stored in a Natural system file for display and other purposes.

13 Resource

This section describes the Natural object of type resource.

Non-Natural file types, such as HTML files, XML style sheets, etc., supported by Natural for Linux are located in the different libraries in the *RES* directory. Natural delivers some with the product, but you can also save your own resources for your applications in *RES*.

Use of Resources

Natural distinguishes two kinds of resources:

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

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

Shared Resources

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

Example of Using a Shared Resource

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

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

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

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

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

Private Resources

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

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

Example of Private Resources

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

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

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

API for Processing Resources

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

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

14 Error Message

Objects of type error message are used to manage application-specific messages defined by the user, or customize the texts of Natural system messages supplied by Software AG.

Error message are created and maintained with the SYSERR utility with the following options:

- Define message ranges for different categories of messages.
- Standardize messages.
- Translate messages into other languages.
- Attach extended (long) message texts for further explanations.

15 Command Processor

Command processors are used to define command-driven navigation systems for Natural applications as an alternative to navigating through hierarchies of menus.

The Natural command processor (NCP) consists of two components: maintenance and runtime. The SYSNCP utility is the maintenance part which comprises all facilities used to define command processor sources and control the navigation within an application. The `PROCESS COMMAND` statement (see the *Statements* documentation) is the runtime part used to invoke Natural programs.

III Function Call

16 Function Call

```
function-name
 (< [([prototype-clause] [intermediate-result-clause])]
   [parameter] [,[parameter]] ... >)
 [array-index-expression]
```

For an explanation of the symbols used in the syntax diagram, see *Syntax Symbols*.

Related Statements: DEFINE PROTOTYPE | DEFINE FUNCTION

Function

A function call invokes a Natural object of the type **function**.

A function is defined with the DEFINE FUNCTION statement which contains the parameters, local and application-independent variables, the result value to be used and the statements to be executed when the function is called.

A function is called by specifying either of the following:

- the function name as defined in the DEFINE FUNCTION statement, or
- an alphanumeric variable that contains the name of the function at execution time. In this case, it is necessary to reference the variable in a DEFINE PROTOTYPE statement with the VARIABLE keyword.

A function call can be used within a Natural statement instead of a read-only operand. In this case, the function has to return a result which is then processed by the statement like a field containing the same value.

It is also possible to use a function call in place of a Natural statement. In this case, the function need not return a result value; if returned, the value result is discarded.

Restrictions

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

- in positions where the operand value is changed by the Natural statement, for example:

  ```
  MOVE 1 TO #FCT(<..>);
  ```

- in a DEFINE DATA statement;
- in a database access statement, such as READ, FIND, SELECT, UPDATE and STORE;
- in an AT BREAK or IF BREAK statement;

- as an argument of Natural system functions, such as `AVER`, `SUM` and `*TRIM`;
- in an array index expression;
- as a parameter of a function call.

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

Syntax Description

Operand Definition Table:

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

Syntax Element Description:

Syntax Element	Description
function-name	**Function Name:** *function-name* is either of the following: - the name of the function to be called as referenced in the `DEFINE FUNCTION` statement, or - the name of an alphanumeric variable which contains the name of the called function at execution time. This variable has to be referenced in a prototype definition with the `VARIABLE` keyword of the `DEFINE PROTOTYPE` statement. If this prototype does not contain the correct parameter and result field definitions, another prototype can be assigned with the *prototype-clause*.
prototype-clause	**Prototype Clause:** See ***prototype-clause* (PT=)**.
intermediate-result-clause	**Intermediate Result Clause:** See ***intermediate-result-clause* (IR=)**.
parameter	**Parameter Specification:** See *parameter*.
array-index-expression	**Array Index Notation:** If the result returned by the function call is an array, an index notation must be provided to address the demanded array occurrences.

Syntax Element	Description
	For details, refer to ***Index Notation*** in *User-Defined Variables*.

prototype-clause (PT=)

```
PT= prototype-name
```

Natural requires parameter definitions and the function result to resolve a function call at compile time. If no prototype matches *function-name*, the parameters or the function result defined for the called function, you can assign a matching prototype with the *prototype-clause*. In this case, the referenced prototype steps in place and is used to resolve the parameter and function result definitions. The *function-name* declared in the referenced prototype is ignored.

Syntax Element Description:

Syntax Element	Description
prototype-name	**Prototype Name:** *prototype-name* is either of the following: ■ the name of the prototype whose result and parameters layouts are to be used, or ■ the name of an alphanumeric field specified as *function-name* in a function call. This field must contain the name of the function to be called at execution time. An array index expression must not be specified with the field name.

intermediate-result-clause (IR=)

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

This clause can be used to specify the *format-length*/*array definition* of the result value for a function call if neither the cataloged object of the function nor a prototype definition is available. If a prototype is available for this function call or if a cataloged object of the called function exists, the result value format specified with the *intermediate-result-clause* is checked for data transfer compatibility.

Syntax Element Description:

Syntax Element	Description
format-length	**Format/Length Definition:** The format and length of the field. For information on the format/length definition of user-defined variables; see ***Format and Length of User-Defined Variables***.
array-definition	**Array Dimension Definition:** With an *array-definition*, you define the lower and upper bounds of the dimensions in an array definition. See *Array Dimension Definition* in the *Statements* documentation.
HANDLE OF OBJECT	**Handle of Object:** Used in conjunction with NaturalX. For further information, see *NaturalX* in the *Programming Guide*.
A, B or U	**Data Format:** Possible formats are alphanumeric, binary or Unicode for dynamic variables.
DYNAMIC	**Dynamic Variable:** A field can be defined as DYNAMIC. For further information on processing dynamic variables, see ***Introduction to Dynamic Variables and Fields***.

parameter

```
{ nX                               }
{                                  }
{ operand [ (AD= { M } ) ]         }
{                { O }             }
{                { A }             }
```

You can specify single or multiple parameters to pass data values to the function. They can be provided as constant values or variables, depending on the DEFINE DATA PARAMETER definition within the function.

The semantic and syntactic rules which apply to the function parameters are the same as described in the parameters section of subprograms; see *Parameters* in the description of the CALLNAT statement.

Operand Definition Table:

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

Note: The options marked with an asterisk only apply on Windows and Linux platforms.

Syntax Element Description:

<table>
<tr><th>Syntax Element</th><th colspan="2">Description</th></tr>
<tr><td>nX</td><td colspan="2">Parameters to be Skipped:
With the notation nX you can specify that the next n parameters are to be skipped (for example, 1X to skip the next parameter, or 3X to skip the next three parameters); this means that for the next n parameters no values are passed to the function.
A parameter that is to be skipped must be defined with the keyword OPTIONAL in the function's DEFINE DATA PARAMETER statement. OPTIONAL means that a value can - but need not - be passed from the invoking object to such a parameter.</td></tr>
<tr><td rowspan="5">AD=</td><td colspan="2">Attribute Definition:
If operand is a variable, you can mark it in one of the following ways:</td></tr>
<tr><td>AD=O</td><td>Non-modifiable:
See session parameter AD=O.
Note: Internally, AD=O is processed in the same way as BY VALUE (see the section parameter-data-definition in the description of the DEFINE DATA statement).</td></tr>
<tr><td>AD=M</td><td>Modifiable:
See session parameter AD=M.
This is the default setting.</td></tr>
<tr><td>AD=A</td><td>Input only:
See session parameter AD=A.</td></tr>
<tr><td colspan="2">Note: If operand is a constant, the attribute definition AD cannot be explicitly specified. For constants, AD=O always applies.</td></tr>
</table>

Example

The example program FUNCEX01 uses the functions F#ADDITION, F#CHAR, F#EVEN and F#TEXT.

All example sources shown in this section are provided as source objects and cataloged objects in the Natural SYSEXPG system library.

Invoking Program FUNCEX01:

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

```
WRITE F#TEXT(<(IR=A12) 'Good'>)     /* Function with intermediate result.
*
END
```

Output of Program `FUNCEX01`

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

Called Function F#ADDITION

The function `F#ADDITION` is defined in the example function `FUNCEX02`.

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

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

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

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

Called Function F#CHAR

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

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

Called Function F#EVEN

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

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

Called Function F#TEXT

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

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

Function Result

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

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

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

Parameter and Result Specifications

In order to properly resolve a function call at compile time, the compiler requires the format, length and array structure of the parameters and the function result. The parameters specified in the function call are checked against the corresponding definitions in the function to ensure that they match. If a function is used within a statement instead of an operand, the function result must match the format, length and array structure of the operand.

You have three options to provide this information:

1. Retrieve the parameter and result specifications implicitly from the cataloged object (if available) of the called function if no `DEFINE PROTOTYPE` statement is executed earlier.

   This method requires the least amount of programming effort.
2. Use a `DEFINE PROTOTYPE` statement. You have to use a `DEFINE PROTOTYPE` statement if the cataloged object of the called function is not available or if the function name is not known at compile time, that is, instead of a function name the name of an alphanumeric variable is specified in the function call.
3. Specify an explicit **(`IR=`)** clause in the function call.

The first two methods comprise a full validation of the format, length and array structure of the parameters and the function result.

- Additional Clauses for the Function Call
- Validation of Parameters and Function Result
- Example with Multiple Definitions in a Function Call

Additional Clauses for the Function Call

If neither a `DEFINE PROTOTYPE` statement nor a cataloged function object exists, you can use the following clauses in your function call:

- The **(`IR=`)** clause specifies the function result format/length/array structure.

  This clause determines which format/length/array structure the compiler should assume for the result field (the intermediate result as used by the statement that contains the function call). If a prototype definition is available for a function call, the (`IR=`) clause overrules the specifications in the prototype.

  The (`IR=`) clause does not enforce any parameter checks.

- The **(PT=)** clause uses a previously defined prototype with a name other than the function name. This clause validates the parameters and the function result by using a DEFINE PROTOTYPE statement with the referenced name.

  In the following example, the function #MULT is called, but the parameter and result specifications from the prototype whose name is #ADD apply:

```
#I := #MULT(<(PT=#ADD) 2 , 3>)
```

Validation of Parameters and Function Result

The first of the following definitions found is used to check the specified parameters:

- the prototype definition referenced in the (PT=) clause;
- the prototype definition in the DEFINE PROTOTYPE statement where the prototype name matches the function name used in the function call;
- the parameter specifications in the cataloged function object which are supplied with the DEFINE FUNCTION statement.

If none of the above is specified, no parameter validation is performed. This provides you the option to supply any number and layout of parameters in the function call without receiving a syntax error.

The first of the following definitions found is used to check the function result:

- the definition provided in the **(IR=)** clause;
- the RETURNS definition in the prototype referenced in the (PT=) clause;
- the prototype definition in the DEFINE PROTOTYPE statement where the prototype name matches the function name used in the function call;
- the function result specification in the cataloged function object.

If none of the above is specified, a syntax error occurs.

Example with Multiple Definitions in a Function Call

Program:

```
** Example 'FUNCBX01': Declare result value and parameters  (Program)
************************************************************************
*
DEFINE DATA LOCAL
  1 #PROTO-NAME (A20)
  1 #PARM1      (I4)
  1 #PARM2      (I4)
END-DEFINE
*
DEFINE PROTOTYPE VARIABLE #PROTO-NAME
  RETURNS (I4)
  DEFINE DATA PARAMETER
    1 #P1 (I4) BY VALUE OPTIONAL
    1 #P2 (I4) BY VALUE
  END-DEFINE
END-PROTOTYPE
*
#PROTO-NAME := 'F#MULTI'
#PARM1      := 3
#PARM2      := 5
*
WRITE #PROTO-NAME(<#PARM1, #PARM2>)
WRITE #PROTO-NAME(<1X ,5>)
*
WRITE F#MULTI(<(PT=#PROTO-NAME) #PARM1,#PARM2>)
*
WRITE F#MULTI(<(IR=N20) #PARM1, #PARM2>)
*
END
```

Function F#MULTI:

```
** Example 'FUNCBX02': Declare result value and parameters  (Function)
************************************************************************
DEFINE FUNCTION F#MULTI
  RETURNS #RESULT (I4) BY VALUE
  DEFINE DATA PARAMETER
    1 #FACTOR1 (I4) BY VALUE OPTIONAL
    1 #FACTOR2 (I4) BY VALUE
  END-DEFINE
  /*
  IF #FACTOR1 SPECIFIED
    #RESULT := #FACTOR1 * #FACTOR2
  ELSE
    #RESULT := #FACTOR2 * 10
  END-IF
  /*
END-FUNCTION
*
END ↩
```

Evaluation Sequence of Functions in Statements

All function calls used within a Natural statement are evaluated before the statement execution starts. They are evaluated in the same order in which they appear in the statement. Function result values are stored in temporary fields that are later used as operands for execution of the statement.

Calling a function that has modifiable parameters which are repeatedly used within the same statement can cause different function results as indicated in the following example.

Example:

Before the `COMPUTE` statement is started, variable `#I` has the value `1`. In a first step, function `F#RETURN` is executed. This changes the value of `#I` to `2` and returns a value of `2` as the function result. After this, the `COMPUTE` operation starts and sums up the values of `#I (2)` and the temporary field `(2)` to a value of `4`.

Program:

```
** Example 'FUNCCX01': Parameter changed within function (Program)
************************************************************************
DEFINE DATA LOCAL
  1 #I      (I2) INIT <1>
  1 #RESULT (I2)
END-DEFINE
*
COMPUTE #RESULT := #I + F#RETURN(<#I>)  /* First evaluate function call,
                                        /* then execute the addition.
*
WRITE '#I     :' #I /
      '#RESULT:' #RESULT
*
END
```

Function:

```
** Example 'FUNCCX02': Parameter changed within function (Function)
************************************************************************
DEFINE FUNCTION F#RETURN
  RETURNS #RESULT (I2) BY VALUE
  DEFINE DATA PARAMETER
    1 #PARM1 (I2) BY VALUE RESULT
  END-DEFINE
  /*
  #PARM1  := #PARM1 + 1       /* Increment parameter.
  #RESULT := #PARM1           /* Set result value.
  /*
END-FUNCTION
```

```
*
END
```

Output of Program `FUNCCX01`**:**

```
#I     :      2
#RESULT:      4
```

Using a Function as a Statement

You can also use a function call in place of a Natural statement without embedding the function call in a statement. In this case, the function call need not return a result value; if returned, the result value is discarded.

You can avoid that such a function call is considered to be part of a previous statement by separating the function call from the previous statement with a semicolon (`;`) as shown in the following example.

Example:

Program:

```
** Example 'FUNCDX01': Using a function as a statement (Program)
************************************************************************
DEFINE DATA LOCAL
  1 #A (I4) INIT <1>
  1 #B (I4) INIT <2>
END-DEFINE
*
*
WRITE 'Write:' #A #B
F#PRINT-ADD(< 2,3 >)     /* Function call belongs to operand list
                        /* immediately preceding it.
*
WRITE // '************************' //
*
WRITE 'Write:' #A #B;    /* Semicolon separates operands and function.
F#PRINT-ADD(< 2,3 >)     /* Function call does not belong to the
                         /* operand list.
*
END
```

Function:

```
** Example 'FUNCDX02': Using a function as a statement (Function)
************************************************************************
DEFINE FUNCTION F#PRINT-ADD
  RETURNS (I4)
  DEFINE DATA PARAMETER
    1 #SUMMAND1 (I4) BY VALUE
    1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE
  /*
  F#PRINT-ADD := #SUMMAND1 + #SUMMAND2    /* Result of function call.
  WRITE 'Function call:' F#PRINT-ADD
  /*
END-FUNCTION
*
END ↩
```

Output of Program FUNCDX01:

```
Function call:          5
Write:           1           2           5

*************************

Write:           1           2
Function call:          5 ↩
```

IV Field Definitions

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

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

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

17 Use and Structure of DEFINE DATA Statement

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

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

Field Definitions in DEFINE DATA Statement

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

There are two ways to define the fields:

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

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

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

Data areas are created and maintained with the data area editor, which is described in the *Editors* documentation.

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

Defining Fields within a DEFINE DATA Statement

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

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

```
END-DEFINE
...
```

Defining Fields in a Separate Data Area

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

Program:

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

Local Data Area `LDA39`:

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

Structuring a DEFINE DATA Statement Using Level Numbers

The following topics are covered:

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

- Level Numbers in Redefinitions

Structuring and Grouping Your Definitions

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

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

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

Generally, variable definitions are on Level 1.

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

Level Numbers in View Definitions

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

Example of Level Numbers in View Definition:

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

Level Numbers in Field Groups

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

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

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

Example of Level Numbers in Group:

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

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

Level Numbers in Redefinitions

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

Example of Level Numbers in Redefinition:

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

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

18 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.

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

Definition of Variables

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

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

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

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

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

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

Example:

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

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

Notes:

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

Referencing of Database Fields Using (r) Notation

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

The following topics are covered below:

Default Referencing of Database Fields

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

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

Referencing with Statement Labels

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

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

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

Example:

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

Referencing with Source-Code Line Numbers

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

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

Example:

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

Note: Due to technical reasons, the line numbers shown on the program editor screen consist of six digits, but actually only the last four digits are processed internally.

Renumbering of Source-Code Line Number References

Line number references (see ***Referencing of Database Fields Using (r) Notation*** and ***Referencing Statements within a Program***) within a source are changed if a related line number is changed by the `RENUMBER` command. Renumbering applies to all line reference patterns, except those within an alphanumeric or a Unicode constant. For example:

```
#FIELD1 := '(1150)'    /* is not renumbered
RESET NAME(1150)       /* is renumbered
```

Note: By default, line number references in alphanumeric and Unicode constants are not renumbered. If they are also to be renumbered, you have to set the profile parameter `RNCONST` to `ON`.

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

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

If the left parenthesis is not immediately followed by *nnnn* or if *nnnn* is followed by any character other than a right parenthesis, a comma or a slash, the pattern is not considered a line number reference and will not be changed.

Note: Due to technical reasons, the line numbers shown on the program editor screen consist of six digits, but actually only the last four digits are processed internally.

Format and Length of User-Defined Variables

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

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

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

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

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

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

Notes:

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

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

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

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

Example of Format/Length Definition - Structured Mode:

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

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

Example of Format/Length Definition - Reporting Mode:

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

Special Formats

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

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

- Format: Handle

Format C - Attribute Control

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

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

Example:

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

For further information, see the session parameter CV.

Formats D - Date, and T - Time

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

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

Example:

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

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

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

Format L - Logical

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

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

Example:

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

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

Format: Handle

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

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

Index Notation

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

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

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

Array Definition - Examples:

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

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

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

Notes:

1. For compatibility with earlier Natural versions, an array range may be specified using a hyphen (-) instead of a colon (:).
2. A mix of both notations, however, is *not* permitted.
3. The hyphen notation is not allowed in a `DEFINE DATA` statement.
4. For new code it is recommended to use the colon (:) notation.

The maximum index value is 1,073,741,824 (1 GB).

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

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

Example of Group Array Resolution:

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

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

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

the occurrences would be output in the following order:

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

and *not*:

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

Array Referencing - Examples:

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

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

Using a Slash before an Array Occurrence

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

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

not:

```
#ARRAY(1000)
```

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

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

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

not:

```
#ARRAY (N7)
```

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

Referencing a Database Array

The following topics are covered below:

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

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

Referencing Multiple-Value Fields and Periodic-Group Fields

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

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

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

or:

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

Notes:

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

Referencing Arrays Defined with Constants

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

Reporting Mode Example:

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

Structured Mode Example:

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

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

Reporting Mode Example:

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

Structured Mode Example:

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

Referencing Arrays Defined with Variables

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

Reporting Mode Example:

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

Structured Mode Example:

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

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

Reporting Mode Example:

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

Structured Mode Example:

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

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

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

Referencing Multiple-Defined Arrays

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

Reporting Mode Example:

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

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

Structured Mode Example:

```
** Example 'INDEX5S': Array definition with constants (structured mode)
**                    (multiple definition of same database field)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 LANG (1:10)
  2 LANG (5:10)
*
1 I (I2)
1 J (I2)
END-DEFINE
*
*
I := 1
J := 2
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  WRITE 'LANG(1.1:10) :' LANG (1.1:10) /
        'LANG(1.I:I+2):' LANG (1.I:I+2) //
  WRITE 'LANG(5.1:5)  :' LANG (5.1:5)  /
        'LANG(5.J)    :' LANG (5.J)
END-READ
END
```

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

Reporting Mode Example:

```
** Example 'INDEX6R': Array definition with variables (reporting mode)
**                    (multiple definition of same database field)
************************************************************************
DEFINE DATA LOCAL
1 I (I2) INIT <1>
1 J (I2) INIT <2>
1 N (I2) INIT <1>
1 EMPLOY-VIEW VIEW OF EMPLOYEES    /* For reporting mode programs
  2 NAME                           /* DEFINE DATA is recommended
```

```
  2 CITY                           /* to use multiple definitions
  2 LANG (I:I+10)                  /* of same database field
  2 LANG (J:J+5)
  2 LANG (4:5)
*
END-DEFINE
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
*
  WRITE 'LANG(I.I)    :' LANG (I.I) /
        'LANG(1.I:I+2):' LANG (I.I:I+10) //
*
  WRITE 'LANG(J.N)    :' LANG (J.N) /
        'LANG(J.2:4)  :' LANG (J.2:4) //
*
  WRITE 'LANG(4.N)    :' LANG (4.N) /
        'LANG(4.N:N+1):' LANG (4.N:N+1) /
LOOP
END
```

Structured Mode Example:

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

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

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

Note concerning databases other than Adabas:

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

See also the data-area-editor line command `*` (in the *Editors* documentation).

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

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

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

Examples:

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

Example Program Using the C* Variable:

```
** Example 'CNOTX01': C* Notation
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 C*INCOME
  2 INCOME
    3 SALARY  (1:5)
    3 C*BONUS (1:2)
    3 BONUS   (1:2,1:2)
  2 C*LANG
  2 LANG      (1:2)
*
1 #I (N1)
```

```
END-DEFINE
*
LIMIT 2
READ EMPL-VIEW BY CITY
  /*
  WRITE NOTITLE 'NAME:' NAME /
       'NUMBER OF LANGUAGES SPOKEN:' C*LANG 5X
       'LANGUAGE 1:' LANG (1) 5X
       'LANGUAGE 2:' LANG (2)
  /*
  WRITE 'SALARY DATA:'
  FOR #I FROM 1 TO C*INCOME
    WRITE 'SALARY' #I SALARY (1.#I)
  END-FOR
  /*
  WRITE 'THIS YEAR BONUS:' C*BONUS(1)  BONUS (1,1) BONUS (1,2)
      / 'LAST YEAR BONUS:' C*BONUS(2)  BONUS (2,1) BONUS (2,2)
  SKIP 1
END-READ
END
```

Output of Program `CNOTX01`:

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

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

C* for Multiple-Value Fields Within Periodic Groups

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

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

Example 1 - Reporting Mode:

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

Example 2 - Structured Mode:

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

Example 3 - Structured Mode:

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

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

Qualifying Data Structures

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

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

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

Example:

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

```
MOVE FULL-NAME.LAST-NAME TO OUTPUT-NAME.LAST-NAME
...
```

The qualifier must be a level-1 data element.

Example:

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

Qualifying a Database Field:

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

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

Examples of User-Defined Variables

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

```
                    /*
END-DEFINE ↩
```

19 Introduction to Dynamic Variables and Fields

Purpose of Dynamic Variables

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

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

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

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

Definition of Dynamic Variables

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

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

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

Restrictions:

The following restrictions apply to a dynamic variable:

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

Value Space Currently Used for a Dynamic Variable

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

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

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

Size Limitation Check

Profile Parameter USIZE

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

Allocating/Freeing Memory Space for a Dynamic Variable

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

Syntax:

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

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

EXPAND

Function

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

Changing the Specified Size

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

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

REDUCE

Function

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

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

Changing the Specified Length

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

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

RESIZE

Function

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

Changing the Specified Length

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

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

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

20 Using Dynamic and Large Variables

General Remarks

Generally, the following rules apply:

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

Exception:

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

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

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

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

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

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

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

Assignments with Dynamic Variables

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

Example:

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

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

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

Example:

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

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

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

Valid syntax:

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

Invalid syntax:

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

Assignment Compatibility

Example:

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

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

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

Initialization of Dynamic Variables

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

Example:

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

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

Example:

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

String Manipulation with Dynamic Alphanumeric Variables

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

Example:

```
** Example 'DYNAMX01': Dynamic variables (with COMPRESS and SEPARATE)
************************************************************************
DEFINE DATA LOCAL
1 #MYDYNTEXT1 (A)   DYNAMIC
1 #TEXT       (A20)
1 #DYN1       (A)   DYNAMIC
1 #DYN2       (A)   DYNAMIC
1 #DYN3       (A)   DYNAMIC
END-DEFINE
*
MOVE ' HELLO WORLD ' TO #MYDYNTEXT1
WRITE #MYDYNTEXT1 (AL=25) 'with length' *LENGTH (#MYDYNTEXT1)
/*  dynamic variable with leading and trailing blanks
*
```

```
MOVE ' HELLO WORLD ' TO #TEXT
*
MOVE #TEXT TO #MYDYNTEXT1
WRITE #MYDYNTEXT1 (AL=25) 'with length' *LENGTH (#MYDYNTEXT1)
/*  dynamic variable with whole variable length of #TEXT
*
COMPRESS #TEXT INTO #MYDYNTEXT1
WRITE #MYDYNTEXT1 (AL=25) 'with length' *LENGTH (#MYDYNTEXT1)
/*  dynamic variable with leading blanks of #TEXT
*
*
#MYDYNTEXT1 := 'HERE COMES THE SUN'
SEPARATE #MYDYNTEXT1 INTO #DYN1 #DYN2 #DYN3 IGNORE
*
WRITE / #MYDYNTEXT1 (AL=25) 'with length' *LENGTH (#MYDYNTEXT1)
WRITE #DYN1 (AL=25) 'with length' *LENGTH (#DYN1)
WRITE #DYN2 (AL=25) 'with length' *LENGTH (#DYN2)
WRITE #DYN3 (AL=25) 'with length' *LENGTH (#DYN3)
/*  #DYN1, #DYN2, #DYN3 are automatically extended or reduced
*
EXAMINE #MYDYNTEXT1 FOR 'SUN' REPLACE 'MOON'
WRITE / #MYDYNTEXT1 (AL=25) 'with length' *LENGTH (#MYDYNTEXT1)
/*  #MYDYNTEXT1 is automatically extended or reduced
*
END
```

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

Logical Condition Criterion (LCC) with Dynamic Variables

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

Example:

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

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

Example:

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

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

Example:

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

Comparison Compatibility

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

Example:

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

AT/IF-BREAK of Dynamic Control Fields

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

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

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

Parameter Transfer with Dynamic Variables

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

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

Call By Reference

Caller	Parameter	
	Static	Dynamic
Static	Yes	No
Dynamic	No	Yes

The formats of dynamic variables A or B must match.

Call by Value (Result)

Caller	Parameter	
	Static	Dynamic
Static	Yes	Yes
Dynamic	Yes	Yes

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

Example 1:

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

Subprogram `DYNAMX03`:

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

Example 2:

```
** Example 'DYNAMX04': Dynamic variables (as parameters)
************************************************************************
DEFINE DATA LOCAL
1 #MYTEXT (A) DYNAMIC
END-DEFINE
*
#MYTEXT := '123456'      /* extended to 6 bytes, *LENGTH(#MYTEXT) = 6
*
CALLNAT 'DYNAMX05' USING #MYTEXT
*
WRITE *LENGTH(#MYTEXT)          /* *LENGTH(#MYTEXT) = 8
                                /* at least 10 bytes are
                                /* allocated (extended in DYNAMX05)
*
END
```

Subprogram DYNAMX05:

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

CALL 3GL Program

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

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

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

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

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

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

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

Example:

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

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

Work File Access with Large and Dynamic Variables

The following topics are covered below:

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

PORTABLE and UNFORMATTED

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

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

The work file type `UNFORMATTED` can be used, for example, to read a video from a database and store it in a file directly playable by other utilities. In the `WRITE WORK` statement, the fields are written to the file with their byte length. All data types (`DYNAMIC` or not) are treated the same. No structural information is inserted. Note that Natural uses a buffering mechanism, so you can expect the data to be completely written only after a `CLOSE WORK`. This is especially important if the file is to be processed with another utility while Natural is running.

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

ASCII, ASCII-COMPRESSED and SAG

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

Special Conditions for TRANSFER and ENTIRE CONNECTION

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

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

DDM Generation and Editing for Varying Length Columns

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

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

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

Example DDM:

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

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

Example:

```
  T  L  Name                 F   Leng          S   D   Remark
     :
     1  L@PICTURE1           I   4                                         /* ↩
length indicator
     1  PICTURE1             B   1000000000          IMAGE
     1  N@PICTURE1          I   2                                       /* NULL ↩
indicator
     1  L@TEXT1              I   4                                         /* ↩
length indicator
     1  TEXT1                A   5000                TEXT
     1  N@TEXT1             I   2                                       /* NULL ↩
indicator
     1  L@DESCRIPTION        I   2                                         /* ↩
length indicator
  M  1  DESCRIPTION          A   100                 VARCHAR(1000)
       :
       :
  ~~~~~~~~~~~~~~~~~~~~Extended Attributes~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/* ↩
concerning PICTURE1
```

```
   Header              :    ---
   Edit Mask           :    ---
   Remarks             :   IMAGE
```

Accessing Large Database Objects

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

Example Program:

```
DEFINE DATA LOCAL

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

END-DEFINE

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

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

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

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

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

(equivalent for Format B variables)

Parameter with LINDICATOR Clause in SQL Statements

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

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

Performance Aspects with Dynamic Variables

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

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

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

Example:

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

Rules:

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

Outputting Dynamic Variables

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

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

Dynamic X-Arrays

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

Example:

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

21 User-Defined Constants

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

Numeric Constants

The following topics are covered below:

Numeric Constants

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

Examples:

```
1234   +1234   -1234

12.34  +12.34  -12.34
```

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

Note: In general, numeric constants are represented internally as format I. In the following cases numeric constants are represented in packed form (format P):

- When decimal digits are specified.
- When the value is too large to fit into format I.
- When they are used as parameters within the statements CALLNAT or PERFORM, or as parameters for a helproutine. For more information, check *Statements* > *CALLNAT*, *PERFORM* or *Programming Guide* > *Passing Parameters to Helproutines*.

Example:

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

Validation of Numeric Constants

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

Alphanumeric Constants

The following topics are covered below:

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

Alphanumeric Constants

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

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

```
'text'
```

or quotation marks (")

```
"text"
```

Examples:

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

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

Apostrophes Within Alphanumeric Constants

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

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

Example:

If you want the following to be output:

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

you can use any of the following notations:

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

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

Concatenation of Alphanumeric Constants

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

Examples:

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

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

Unicode Constants

The following topics are covered below:

- Unicode Text Constants
- Apostrophes Within Unicode Text Constants
- Unicode Hexadecimal Constants
- Concatenation of Unicode Constants

Unicode Text Constants

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

```
U'text'
```

or quotation marks (")

```
U"text"
```

Example:

```
U'HELLO'
```

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

Apostrophes Within Unicode Text Constants

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

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

Example:

If you want the following to be output:

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

you can use any of the following notations:

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

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

Unicode Hexadecimal Constants

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

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

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

Example:

This example defines the string `45`.

```
UH'00340035'
```

Concatenation of Unicode Constants

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

Valid Example:

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

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

Invalid Example:

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

Further Valid Example:

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

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

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

Date and Time Constants

The following topics are covered below:

Date Constant

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

Date constants may have the following formats:

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

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

Example:

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

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

Time Constant

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

A time constant has the following format:

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

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

Example:

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

Extended Time Constant

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

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

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

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

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

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

Hexadecimal Constants

The following topics are covered below:

- Hexadecimal Constants
- Concatenation of Hexadecimal Constants

Hexadecimal Constants

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

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

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

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

ASCII examples:

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

EBCDIC examples:

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

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

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

Example:

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

Note: If a hexadecimal constant is output that contains any characters from the ranges `H'00'` to `H'1F'` or `H'80'` to `H'A0'`, these characters will not be output, as they would be interpreted as terminal control characters. As of Version 2.2 these hex constants are not suppressed.

Concatenation of Hexadecimal Constants

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

ASCII example:

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

EBCDIC example:

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

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

Logical Constants

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

Example:

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

Floating Point Constants

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

Example:

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

Attribute Constants

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

The following attributes may be used:

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

See also session parameters AD and CD.

Example:

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

Handle Constants

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

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

Defining Named Constants

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

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

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

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

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

Example:

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

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

22 Initial Values (and the RESET Statement)

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

Note: For example definitions of assigning initial values to arrays, see *Example 2 - DEFINE DATA (Array Definition/Initialization)* in the *Statements* documentation.

Default Initial Value of a User-Defined Variable/Array

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

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

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

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

- Assigning Characters as Initial Value for Alphanumeric/Unicode Variables

Assigning a Modifiable Initial Value

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

Example:

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

Assigning a Constant Initial Value

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

Example:

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

Assigning a Natural System Variable as Initial Value

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

Example:

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

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

Assigning Characters as Initial Value for Alphanumeric/Unicode Variables

As initial value, a variable can also be filled, entirely or partially, with a specific character string (only possible for variables of the Natural data format A or U).

- **Filling an entire field:**
  With the option FULL LENGTH <*character-string*>, the entire field is filled with the specified characters.

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

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

- **Filling the first*n* positions of a field:**
  With the option LENGTH *n* <*character-string*>, the first *n* positions of the field are filled with the specified characters.

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

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

Resetting a User-Defined Variable to its Initial Value

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

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

Notes:

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

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

Reset to Default Initial Value

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

Example:

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

Reset to Initial Value Defined in DEFINE DATA

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

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

Example:

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

23 Redefining Fields

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

Using the REDEFINE Option of DEFINE DATA

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

> **Important:** Dynamic variables are not allowed in a redefinition.

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

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

Example 1:

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

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

Example 2:

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

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

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

Example 3:

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

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

Example Program Illustrating the Use of a Redefinition

The following program illustrates the use of a redefinition:

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

Output of Program `DDATAX01`:

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

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

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


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

24 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.

Defining Arrays

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

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

To define a one-dimensional array

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

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

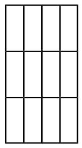
```
DEFINE DATA LOCAL
1 #ARRAY (A10/1:3)
END-DEFINE
...
```

To define a two-dimensional array

- Specify an index notation for both dimensions:

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

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



Initial Values for Arrays

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

Assigning Initial Values to One-Dimensional Arrays

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

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

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

  `A` is assigned to the second occurrence.

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

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

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

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

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

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

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

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

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

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

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

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

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

Alternatively, you could specify:

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

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

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

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

Assigning Initial Values to Two-Dimensional Arrays

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

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

Preliminary Information

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

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

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

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

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

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

Assigning the Same Value

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

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

		A	

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

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

		A	
		A	
		A	

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

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

A	A	A	A
A	A	A	A

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

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

A	A		
A	A		

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

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

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

Alternatively, you could specify:

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

Assigning Different Values

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

	A		
	B		
	C		

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

	A	A	
	B	B	
	C	C	

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

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

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

A	A	A	A
C	C	C	C

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

A	A	A	A
B	B	B	B

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

A		D	
B		E	
C		F	

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

A	B	C	D

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

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

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

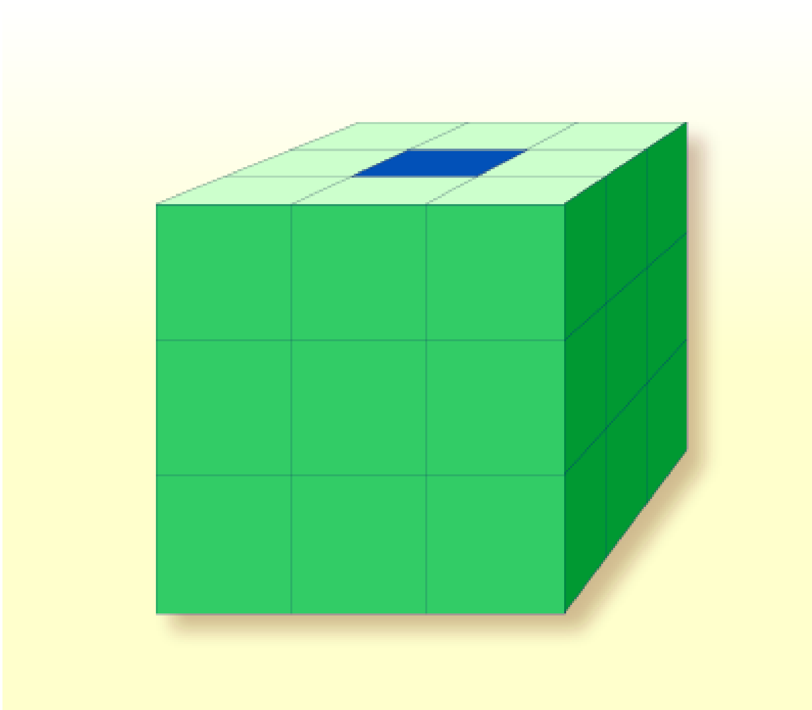
	B		
A	B		
	B	C	D

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

	B		
A	C		
	D	E	F

A Three-Dimensional Array

A three-dimensional array could be visualized as follows:



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

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

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

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

Arrays as Part of a Larger Data Structure

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

Example:

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

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

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

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

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

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

Database Arrays

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

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

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

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

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

Using Arithmetic Expressions in Index Notation

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

Examples:

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

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

Arithmetic Support for Arrays

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

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

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

Examples of Array Arithmetics

The following examples assume the following field definitions:

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

1. `ADD #A(*,*) TO #B(*,*)`

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

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

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

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

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

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

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

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

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

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

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

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

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

25 X-Arrays

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

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

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

Definition

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

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

Example:

```
DEFINE DATA LOCAL
1 #X-ARR1 (A5/1:*)         /* lower bound is fixed at 1, upper bound is variable
1 #X-ARR2 (A5/*)           /* shortcut for (A5/1:*)
1 #X-ARR3 (A5/*:100)       /* lower bound is variable, upper bound is fixed at 100
1 #X-ARR4 (A5/1:10,1:*)    /* 1st dimension has a fixed index range with (1:10)
END-DEFINE                 /* 2nd dimension has fixed lower bound 1 and variable ↩
upper bound
```

Storage Management of X-Arrays

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

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

Example:

```
DEFINE DATA LOCAL
1 #X-ARR(I4/10:*)
END-DEFINE
EXPAND ARRAY #X-ARR TO (10:10000)
/* #X-ARR(10) to #X-ARR(10000) are accessible
WRITE *LBOUND(#X-ARR)                  /* is 10
   *UBOUND(#X-ARR)                     /* is 10000
   *OCCURRENCE(#X-ARR)                 /* is 9991
#X-ARR(*) := 4711                      /* same as #X-ARR(10:10000) := 4711
/* resize array from current lower bound=10 to upper bound =1000
RESIZE ARRAY #X-ARR TO (*:1000)
/* #X-ARR(10) to #X-ARR(1000) are accessible
/* #X-ARR(1001) to #X-ARR(10000) are released
WRITE *LBOUND(#X-ARR)                  /* is 10
   *UBOUND(#X-ARR)                     /* is 1000
   *OCCURRENCE(#X-ARR)                 /* is 991
/* release all occurrences
REDUCE ARRAY #X-ARR TO  0
WRITE *OCCURRENCE(#X-ARR)              /* is 0
```

Storage Management of X-Group Arrays

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

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

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

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

Example - Independent/Dependent Dimensions:

```
DEFINE  DATA LOCAL
1 #X-GROUP-ARR1(1:*)                    /* (1:*)
  2 #X-ARR1    (I4)                     /* (1:*)
  2 #X-ARR2    (I4/2:*)                 /* (1:*,2:*)
  2 #X-GROUP-ARR2                       /* (1:*)
    3 #X-ARR3  (I4)                     /* (1:*)
    3 #X-ARR4  (I4/3:*)                 /* (1:*,3:*)
    3 #X-ARR5  (I4/4:*, 5:*)            /* (1:*,4:*,5:*)
END-DEFINE
```

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

Name	Dependent Dimension	Independent Dimension
#X-GROUP-ARR1		(1:*)
#X-ARR1	(1:*)	
#X-ARR2	(1:*)	(2:*)
#X-GROUP-ARR2	(1:*)	
#X-ARR3	(1:*)	
#X-ARR4	(1:*)	(3:*)
#X-ARR5	(1:*)	(4:*,5:*)

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

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

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

```
EXPAND ARRAY #X-GROUP-ARR1 TO (1:11)       /* #X-ARR1(1:11) are allocated
                                           /* #X-ARR3(1:11) are allocated
EXPAND ARRAY #X-ARR2 TO (*:*, 2:12)        /* #X-ARR2(1:11, 2:12) are allocated
EXPAND ARRAY #X-ARR2 TO (1:*, 2:12)        /* same as before
EXPAND ARRAY #X-ARR2 TO (*  , 2:12)        /* same as before
EXPAND ARRAY #X-ARR4 TO (*:*, 3:13)        /* #X-ARR4(1:11, 3:13) are allocated
EXPAND ARRAY #X-ARR5 TO (*:*, 4:14, 5:15)  /* #X-ARR5(1:11, 4:14, 5:15) are allocated
```

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

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

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

Referencing an X-Array

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

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

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

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

Example:

```
DEFINE DATA LOCAL
1 #X-ARR (A10/1:*) /* X-array only defined, but not allocated
END-DEFINE
RESET #X-ARR(*)    /* no error, because complete field referenced with (*)
RESET #X-ARR(1:3)  /* runtime error, because individual occurrences (1:3) are ↩
referenced
END  ↩
```

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

Parameter Transfer with X-Arrays

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

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

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

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

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

Example with Call By Value

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

Call By Reference/Call By Value Result

Caller	Parameter			

	Static	Variable (1:V)	X-Array with a fixed lower bound, e.g.
			`DEFINE DATA ↩`
			`PARAMETER`
			`1 #PX (A10/1:*)`
				X-Array with a fixed upper bound, e.g.
				`DEFINE DATA ↩`
				`PARAMETER`
				`1 #PX (A10/*:1)`

Static	no	no	no	no
X-array subrange, for example: `CALLNAT...#XA(1:5)`	no	no	no	no

X-Array with a fixed lower bound, complete range, for example:
`DEFINE DATA LOCAL`
`1 #XA(A10/1:*)`
`...`
`CALLNAT...#XA(*)`
	no	no	yes	no

X-Array with a fixed upper bound, complete range, for example:
`DEFINE DATA LOCAL`
`1 #XA(A10/*:1)`
`...`
`CALLNAT...#XA(*)`
	no	no	no	yes

Parameter Transfer with X-Group Arrays

The declaration of an X-group array implies that each element of the group will have the same values for upper boundary and lower boundary. Therefore, the number of occurrences of dependent dimensions of fields of an X-group array can only be changed when the group name of the X-group array is given with a `RESIZE`, `REDUCE` or `EXPAND` statement (see *Storage Management of X-Group Arrays* above).

Members of X-group arrays may be transferred as parameters to X-group arrays defined in a parameter data area. The group structures of the caller and the callee need not necessarily be identical. A RESIZE, REDUCE or EXPAND done by the callee is only possible as far as the X-group array of the caller stays consistent.

Example - Elements of X-Group Array Passed as Parameters:

Program:

```
DEFINE DATA LOCAL
1 #X-GROUP-ARR1(1:*)                    /* (1:*)
   2 #X-ARR1 (I4)                       /* (1:*)
   2 #X-ARR2 (I4)                       /* (1:*)
1 #X-GROUP-ARR2(1:*)                    /* (1:*)
   2 #X-ARR3 (I4)                       /* (1:*)
   2 #X-ARR4 (I4)                       /* (1:*)
END-DEFINE
...
CALLNAT ... #X-ARR1(*) #X-ARR4(*)
...
END
```

Subprogram:

```
DEFINE DATA PARAMETER
1 #X-GROUP-ARR(1:*)                     /* (1:*)
   2 #X-PAR1 (I4)                       /* (1:*)
   2 #X-PAR2 (I4)                       /* (1:*)
END-DEFINE
...
RESIZE ARRAY #X-GROUP-ARR to (1:5)
...
END
```

The RESIZE statement in the subprogram is not possible. It would result in an inconsistent number of occurrences of the fields defined in the X-group arrays of the program.

X-Array of Dynamic Variables

An X-array of dynamic variables may be allocated by first specifying the number of occurrences using the EXPAND statement and then assigning a value to the previously allocated array occurrences.

Example:

```
DEFINE DATA LOCAL
 1 #X-ARRAY(A/1:*)  DYNAMIC
END-DEFINE
EXPAND  ARRAY   #X-ARRAY TO (1:10)
  /* allocate #X-ARRAY(1) to #X-ARRAY(10) with zero length.
  /* *LENGTH(#X-ARRAY(1:10)) is zero
#X-ARRAY(*) := 'abc'
  /* #X-ARRAY(1:10) contains 'abc',
  /* *LENGTH(#X-ARRAY(1:10)) is 3
EXPAND  ARRAY   #X-ARRAY TO (1:20)
  /* allocate #X-ARRAY(11) to #X-ARRAY(20) with zero length
  /* *LENGTH(#X-ARRAY(11:20)) is zero
#X-ARRAY(11:20) := 'def'
  /* #X-ARRAY(11:20) contains 'def'
  /* *LENGTH(#X-ARRAY(11:20)) is 3
```

Lower and Upper Bound of an Array

The system variables `*LBOUND` and `*UBOUND` contain the current lower and upper bound of an array for the specified dimension(s): (1,2 or 3).

If no occurrences of an X-array have been allocated, the access to `*LBOUND` or `*UBOUND` is undefined for the variable index bounds, that is, for the boundaries that are represented by an asterisk (*) character in the index definition, and leads to a runtime error. In order to avoid a runtime error, the system variable `*OCCURRENCE` may be used to check against zero occurrences before `*LBOUND` or `*UBOUND` is evaluated:

Example:

```
IF *OCCURRENCE (#A) NE 0  AND  *UBOUND(#A) < 100 THEN ...
```

V Database Access

This part describes various aspects of accessing data in a database with Natural.

Note: On principle, the features and examples described for Adabas also apply to other database management systems. Differences, if any, are described in the relevant interface documentation, the *Statements* documentation or the *Parameter Reference*.

26 Natural and Database Access

This chapter gives an overview of the facilities that Natural provides for accessing different types of database management systems.

Database Management Systems Supported by Natural

Natural offers specific database interfaces for the following types of database management systems (DBMS):

- Nested-relational DBMS (Adabas)
- SQL-type DBMS (Oracle, Sybase, Informix, MS SQL Server)
- XML-type DBMS (Tamino)

The following topics are covered below:

Adabas

Via its integrated Adabas interface, Natural can access Adabas databases either on a local machine or on remote computers. For remote access, an additional routing and communication software such as Entire Net-Work is necessary. In any case, the type of host machine running the Adabas database is transparent for the Natural user.

Tamino

Natural for Tamino offers the possibility to access a Tamino database server on a local machine or on a remote host using a native HTTP protocol. The Tamino database can be accessed in the same manner as data access is done with Adabas or SQL databases.

SQL Databases

Natural accesses SQL database systems via Entire Access, a generic interface and routing software that supports various SQL database management systems such as Oracle, MS SQL Server or standardized ODBC connections. For a complete overview of the SQL database management systems and platforms supported, refer to the Entire Access documentation. Information on Natural configuration aspects is contained in the document Natural and Entire Access.

Profile Parameters Influencing Database Access

There are various Natural profile parameters to define how Natural handles the access to databases.

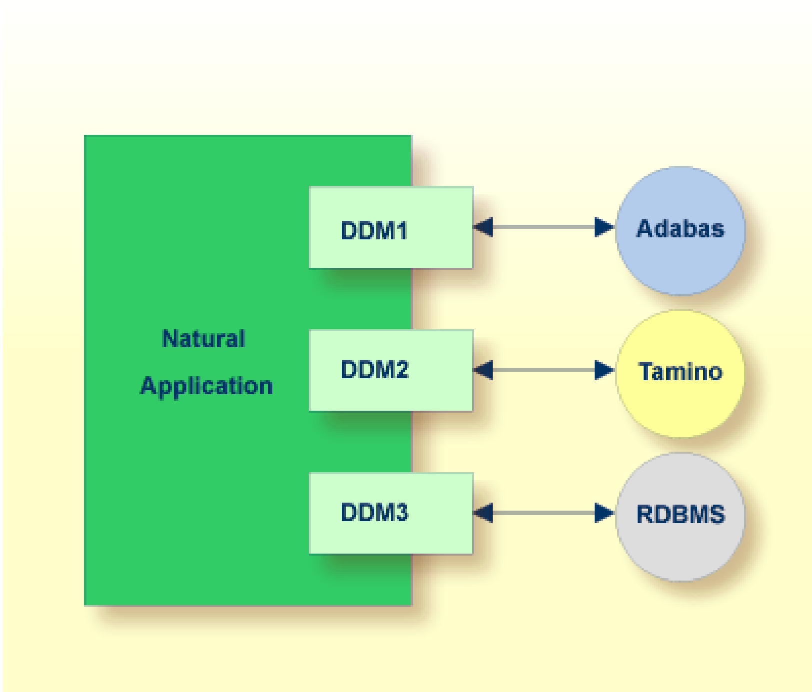
For an overview of these profile parameters, see the section *Database Management System Assignments* in *Overview of Configuration File Parameters* in the *Configuration Utility* documentation.

For a detailed parameter description, refer to the corresponding section in the *Parameter Reference*.

Access through Data Definition Modules

To enable convenient and transparent access to the different database management systems, a special object, the "data definition module" (DDM), is used in Natural. This DDM establishes the connection between the Natural data structures and the data structures in the database system to be used. Such a database structure might be a table in an SQL database, a file in an Adabas database or a doctype in a Tamino database. Hence, the DDM hides the real structure of the database accessed from the Natural application. DDMs are created using the Natural DDM editor.

Natural is capable of accessing multiple types of databases (Adabas, Tamino, RDBMS) from within a single application by using references to DDMs that represent the specific data structures in the specific database system. The diagram below shows an application that connects to different types of database.



Natural's Data Manipulation Language

Natural has a built-in data manipulation language (DML) that allows Natural applications to access all database systems supported by Natural using the same language statements such as `FIND`, `READ`, `STORE` or `DELETE`. These statements can be used in a Natural application without knowing the type of database that is going to be accessed.

Natural determines the real type of database system from its configuration files and translates the DML statements into database-specific commands; that is, it generates direct commands for Adabas, SQL statement strings and host variable structures for SQL databases and XQuery requests for a Tamino database.

Because some of the Natural DML statements provide functionality that cannot be supported for all database types, the use of this functionality is restricted to specific database systems. Please, note the corresponding database-specific considerations in the statements documentation.

Natural's Special SQL Statements

In addition to the "normal" Natural DML statements, Natural provides a set of SQL statements for a more specific use in conjunction with SQL database systems; see *SQL Statements Overview* (in the *Statements* documentation).

Flexible SQL and facilities for working with stored procedures complete the set of SQL commands. These statements can be used for SQL database access only and are not valid for Adabas or other non-SQL-databases.

27 Accessing Data in an Adabas Database

This chapter describes various aspects of accessing data in an Adabas database with Natural.

Adabas Database Management Interfaces ADA and ADA2

The keywords ADA and ADA2 are used synonymously for the same database interface. This single interface comprises the functional capabilities of the former distinct ADA and ADA2 interfaces.

The ADA/ADA2 interface can be used for Software AG products which have their own system files like Predict or Natural Security and for accessing large objects like LA fields at the same time. It is therefore possible, to use Natural Security or Predict as well as Adabas files with large data objects on the same database. In older Natural versions two distinct databases were required here.

For reasons of backward compatibility the keywords ADA and ADA2 do still exist.

Please note that Natural programs containing Adabas calls that are cataloged with Natural Version 9.1.3 or above cannot be executed with an older Natural version. An error message such as NAT6237: Attempt to execute database calls to undefined database type might occur. Possible work-arounds are: Catalog the application with the older Natural version or mark the affected DBIDs as ADA2 in the old environment.

Data Definition Modules - DDMs

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

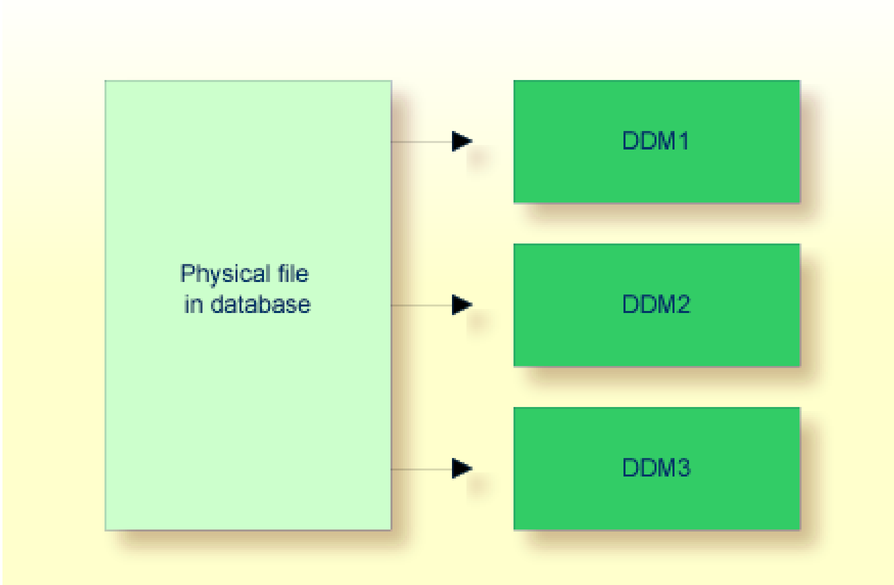
This section covers the following topics:

- Use of Data Definition Modules
- Maintaining DDMs
- Listing/Displaying DDMs

Use of Data Definition Modules

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

For each physical file of a database, one or more DDMs can be defined. And for each DDM one or more data views can be defined as described *View Definition* in the `DEFINE DATA` statement documentation and explained in the section ***Defining a Database View***.



DDMs are defined by the Natural administrator with Predict (or, if Predict is not available, with the corresponding Natural function).

Maintaining DDMs

Use the system command `SYSDDM` to invoke the `SYSDDM` utility. The `SYSDDM` utility is used to perform all functions needed for the creation and maintenance of Natural data definition modules.

For further information on the `SYSDDM` utility, see the section *DDM Services* in the *Editors* 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.).

For the field attributes defined in a DDM, refer to *Using the DDM Editor* in the section *DDM Services* of the *Editors* documentation.

Listing/Displaying DDMs

Database Arrays

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

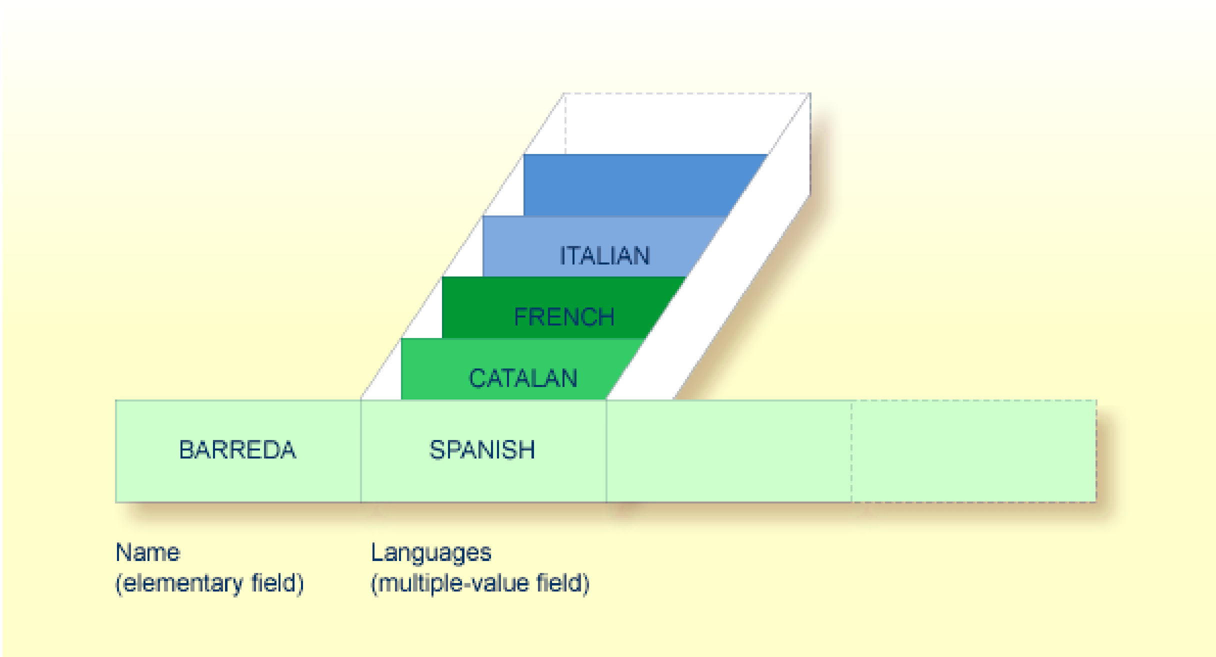
This section covers the following topics:

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

Multiple-Value Fields

A multiple-value field is a field which can have more than one value (up to 65534, depending on the Adabas version and definition of the FDT) within a given record.

Example:



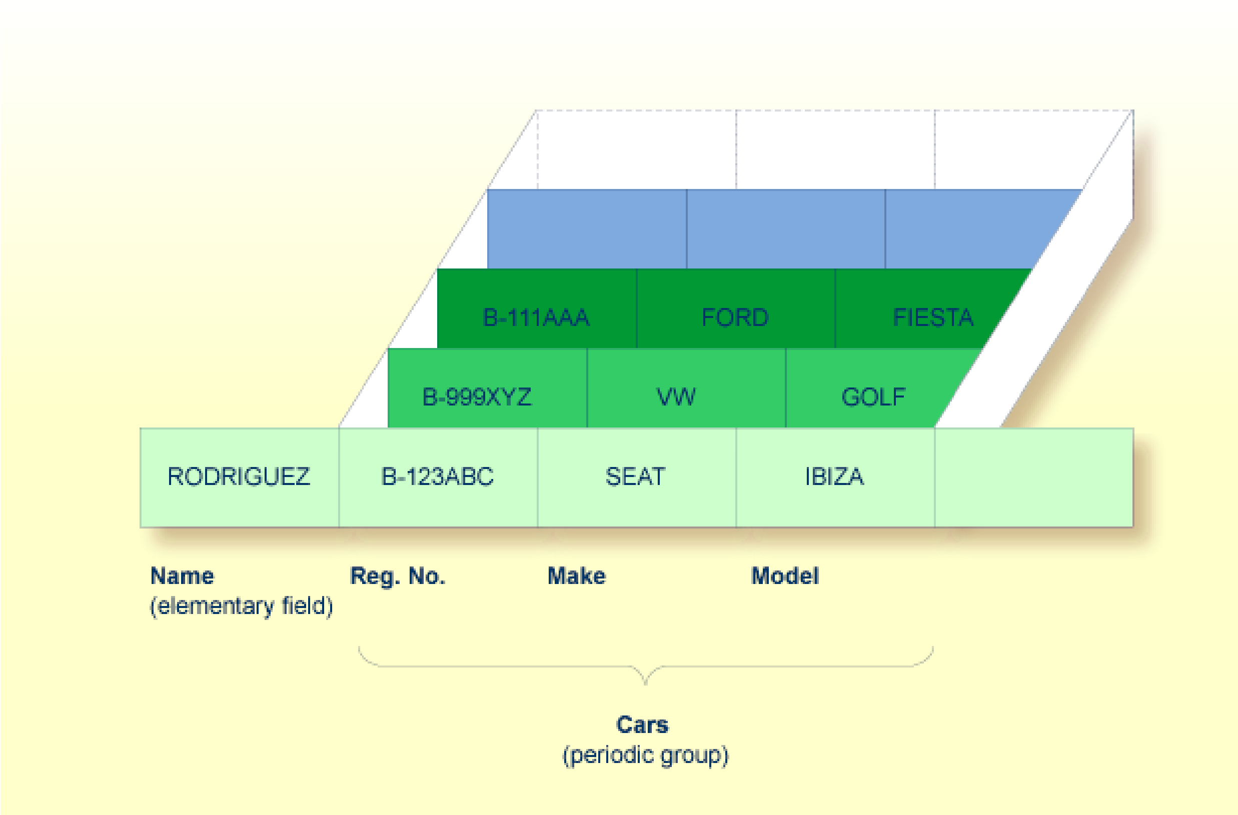
Assuming that the above is a record in an `EMPLOYEES` file, the first field (Name) is an elementary field, which can contain only one value, namely the name of the person; whereas the second field (Languages), which contains the languages spoken by the person, is a multiple-value field, as a person can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

Periodic Groups

A periodic group is a group of fields (which may be elementary fields and/or multiple-value fields) that may have more than one occurrence (up to 65534, depending on the Adabas version and definition of the field definition table (FDT)) within a given record.

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

Example:



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

Referencing Multiple-Value Fields and Periodic Groups

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

Examples:

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

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

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

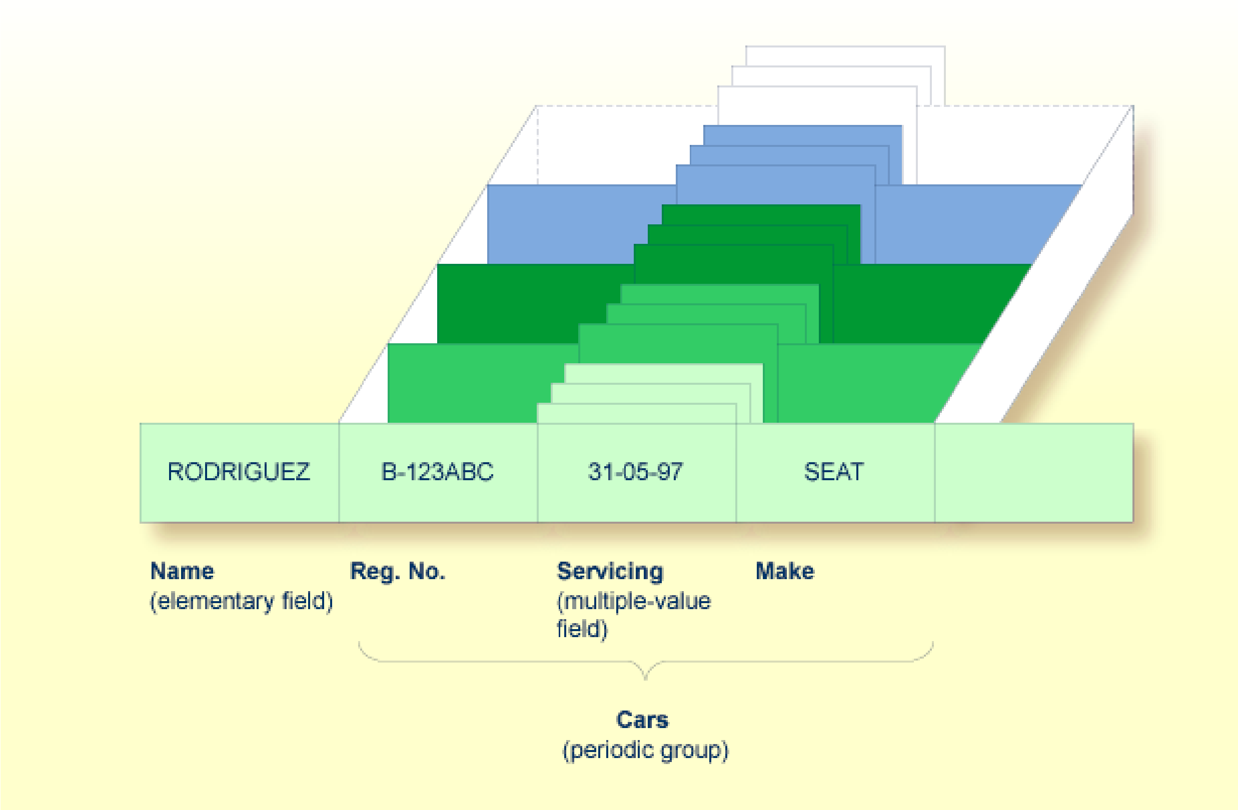
The various occurrences of the periodic group CARS can be referenced in the same manner:

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

Multiple-Value Fields within Periodic Groups

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

Example:



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

Referencing Multiple-Value Fields within Periodic Groups

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

Examples:

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

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

Referencing the Internal Count of a Database Array

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

The count is returned in format/length N3. See ***Referencing the Internal Count for a Database Array*** for further details.

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

Defining a Database View

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

In the view, you specify the name of the data definition module (see ***Data Definition Modules - DDMs***) from which the fields are to be taken, and the names of the database fields (see ***Field Definitions***) themselves (that is, their long names, not their database-internal short names).

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

As described in the section ***Statements for Database Access***, the view name is used in the statements READ, FIND, HISTOGRAM to determine which database is to be accessed.

For further information on the complete syntax of the view definition option or on the definition/re-definition of a group of fields, see *View Definition* in the description of the DEFINE DATA statement in the *Statements* documentation.

Basically, you have the following options to define a database view:

- **Inside the Program**
  You can define a database view inside the program, that is, directly within the `DEFINE DATA` statement of the program.
- **Outside the Program**
  You can define a database view outside the program, that is, in a separate object: either a local data area (LDA) or a global data area (GDA), with the `DEFINE DATA` statement of the program referencing that data area.

To define a database view inside the program

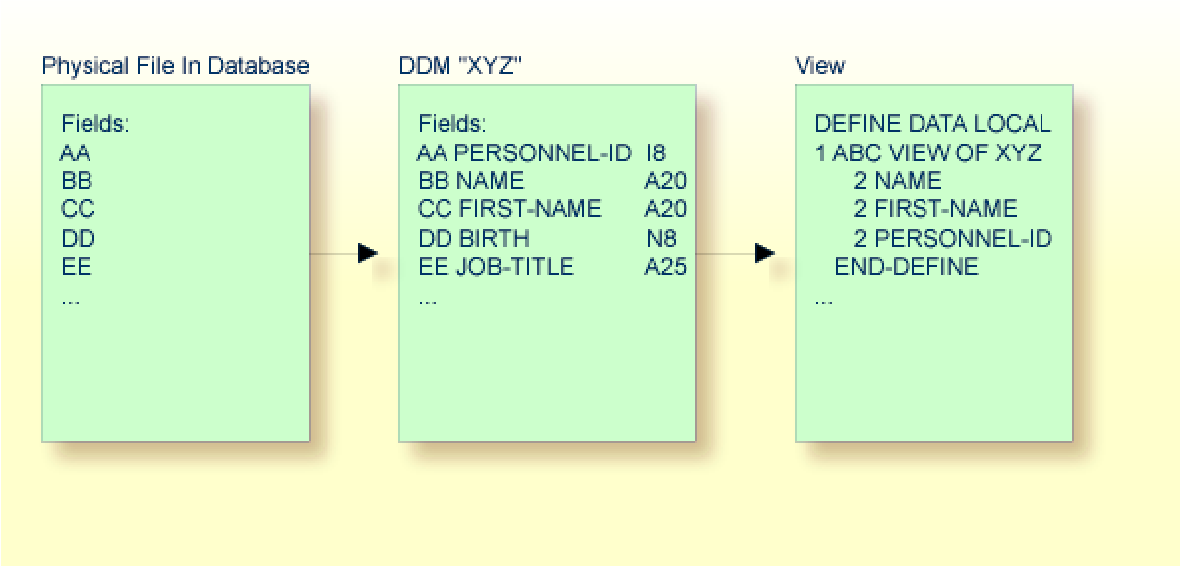
1 At Level 1, specify the view name as follows:

```
1 view-name VIEW OF ddm-name
```

where *`view-name`* is the name you choose for the view, *`ddm-name`* is the name of the DDM from which the fields specified in the view are taken.

2 At Level 2, specify the names of the database fields from the DDM.

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



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

Sample Program:

In this example, the *view-name* is VIEWEMP, and the *ddm-name* is EMPLOYEES, and the names of the database fields taken from the DDM are NAME, FIRST-NAME and PERSONNEL-ID.

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

To define a database view outside the program

1 In the program, specify:

```
DEFINE DATA LOCAL
       USING <data-area-name>
END-DEFINE
  ...
```

where *data-area-name* is the name you choose for the local or global data area, for example, LDA39.

2 In the data area to be referenced:

1. At Level 1 in the Name column, specify the name you choose for the view, and in the Miscellaneous column, the name of the DDM from which the fields specified in the view are taken.
2. At Level 2, specify the names of the database fields from the DDM.

   Example LDA39:

   In this example, the view name is VIEWEMP, the DDM name is EMPLOYEES, and the names of the database fields taken from the DDM are PERSONNEL-ID, FIRST-NAME and NAME.

```
I T L  Name                            F Length     Miscellaneous         ↩

All -- -------------------------------- - ---------- ------------------------->
  V  1 VIEWEMP                                      EMPLOYEES             ↩

     2 PERSONNEL-ID                     A         8                       ↩

     2 FIRST-NAME                       A        20                       ↩

     2 NAME                             A        20                       ↩
```

```
    1 #VARI-A                        A         20                              ↩
    1 #VARI-B                        N        3.2                              ↩
    1 #VARI-C                        I          4                              ↩
↩
```

Considerations for Large Data Fields

- If large alphanumeric (LA) or large object (LOB) fields (Adabas LA/LB option) are to be used, these fields can be specified within the view definition with both fixed format/length, for example, A20 or U20, and dynamic format/length, for example, (A)DYNAMIC or U(DYNAMIC).
- Length indicator fields L@... can also be specified within views if they are related to LA or LOB fields.

Statements for Database Access

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

Statement	Meaning
READ	Select a range of records from a database in a specified sequence.
FIND	Select from a database those records which meet a specified search criterion.
HISTOGRAM	Read only the values of one database field, or determine the number of records which meet a specified search criterion.

READ Statement

The following topics are covered:

- Further Example of READ Statement

Use of READ Statement

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

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

In this document, only `READ IN LOGICAL SEQUENCE` is discussed, as it is the most frequently used form of the `READ` statement.

For information on the other two options, please refer to the description of the `READ` statement in the *Statements* documentation.

Basic Syntax of READ Statement

The basic syntax of the `READ` statement is:

```
READ view IN LOGICAL SEQUENCE BY descriptor
```

or shorter:

```
READ view LOGICAL BY descriptor
```

- where

view	is the name of a view defined in the `DEFINE DATA` statement and as explained in ***Defining a Database View***.
descriptor	is the name of a database field defined in that view. The values of this field determine the order in which the records are read from the database.

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

```
READ view BY descriptor
```

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

```
READ view LOGICAL
```

Example of READ Statement

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

Output of Program `READX01`:

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

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

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

        NAME         PERSONNEL          CURRENT
                        ID             POSITION
-------------------- --------- ------------------------

ABELLAN              60008339  MAQUINISTA
ACHIESON             30000231  DATA BASE ADMINISTRATOR
ADAM                 50005800  CHEF DE SERVICE
ADKINSON             20008800  PROGRAMMER
ADKINSON             20009800  DBA
ADKINSON             2001100
```

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

```
READ MYVIEW BY BIRTH
```

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

Limiting the Number of Records to be Read

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

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

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

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

STARTING/ENDING Clauses

The `READ` statement also allows you to qualify the selection of records based on the *value* of a descriptor field. With an `EQUAL TO/STARTING FROM` option in the `BY` clause, you can specify the value at which reading should begin. (Instead of using the keyword `BY`, you may specify the keyword `WITH`, which would have the same effect). By adding a `THRU/ENDING AT` option, you can also specify the value in the logical sequence at which reading should end.

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

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

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

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

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

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

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

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

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

WHERE Clause

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

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

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

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

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

The `WHERE` clause differs from the `BY` clause in two respects:

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

The following logical operators are possible in a `WHERE` clause:

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

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

```
** Example 'READX02': READ  (with STARTING, ENDING and WHERE clause)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 INCOME     (1:2)
    3 CURR-CODE
    3 SALARY
    3 BONUS    (1:1)
END-DEFINE
*
READ (3) MYVIEW WITH  JOB-TITLE
STARTING FROM 'TRAINEE' ENDING AT 'TRAINEE'
                WHERE CURR-CODE (*) = 'USD'
  DISPLAY NOTITLE NAME / JOB-TITLE 5X INCOME (1:2)
  SKIP 1
END-READ
END
```

Output of Program READX02:

```
         NAME                               INCOME
         CURRENT
        POSITION                CURRENCY   ANNUAL     BONUS
                                  CODE     SALARY
-------------------------     -------- ---------- ----------

SENKO                          USD            23000          0
TRAINEE                        USD            21800          0

BANGART                        USD            25000          0
TRAINEE                        USD            23000          0

LINCOLN                        USD            24000          0
TRAINEE                        USD            22000          0
```

Further Example of READ Statement

See the following example program:

READX03 - READ statement

FIND Statement

The following topics are covered:

Use of FIND Statement

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

Basic Syntax of FIND Statement

The basic syntax of the `FIND` statement is:

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

or shorter:

```
FIND view WITH field = value
```

- where

view	is the name of a view as defined in the `DEFINE DATA` statement and as explained in ***Defining a Database View***.
field	is the name of a database field as defined in that view.

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

For the complete syntax, refer to the `FIND` statement documentation.

Limiting the Number of Records to be Processed

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

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

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

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

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

WHERE Clause

With the WHERE clause of the FIND statement, you can specify an additional selection criterion which is evaluated *after* a record (selected with the WITH clause) has been read and *before* any processing is performed on the record.

Example of FIND Statement with WHERE Clause

```
** Example 'FINDX01': FIND (with WHERE)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 CITY
END-DEFINE
*
FIND MYVIEW WITH CITY = 'PARIS'
            WHERE JOB-TITLE = 'INGENIEUR COMMERCIAL'
  DISPLAY NOTITLE CITY JOB-TITLE PERSONNEL-ID NAME
END-FIND
END
```

Note: In this example only those records which meet the criteria of the WITH clause *and* the WHERE clause are processed in the DISPLAY statement.

Output of Program FINDX01:

```
       CITY                  CURRENT          PERSONNEL         NAME
                             POSITION            ID
-------------------- ------------------------ --------- --------------------

PARIS                INGENIEUR COMMERCIAL      50007300  CAHN
PARIS                INGENIEUR COMMERCIAL      50006500  MAZUY
PARIS                INGENIEUR COMMERCIAL      50004700  FAURIE
PARIS                INGENIEUR COMMERCIAL      50004400  VALLY
PARIS                INGENIEUR COMMERCIAL      50002800  BRETON
PARIS                INGENIEUR COMMERCIAL      50001000  GIGLEUX
PARIS                INGENIEUR COMMERCIAL      50000400  KORAB-BRZOZOWSKI
```

IF NO RECORDS FOUND Condition

If no records are found that meet the search criteria specified in the `WITH` and `WHERE` clauses, the statements within the `FIND` processing loop are not executed (for the previous example, this would mean that the `DISPLAY` statement would not be executed and consequently no employee data would be displayed).

However, the `FIND` statement also provides an `IF NO RECORDS FOUND` clause, which allows you to specify processing you wish to be performed in the case that no records meet the search criteria.

Example:

```
** Example 'FINDX02': FIND (with IF NO RECORDS FOUND)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
END-DEFINE
*
FIND MYVIEW WITH NAME = 'BLACKSMITH'
  IF NO RECORDS FOUND
    WRITE 'NO PERSON FOUND.'
  END-NOREC
  DISPLAY NAME FIRST-NAME
END-FIND
END
```

The above program selects all records in which the field `NAME` contains the value `BLACKSMITH`. For each selected record, the name and first name are displayed. If no record with `NAME = 'BLACKSMITH'` is found on the file, the `WRITE` statement within the `IF NO RECORDS FOUND` clause is executed.

Output of Program `FINDX02`:

```
Page      1                                              04-11-11  14:15:54

        NAME                 FIRST-NAME
-------------------- --------------------

NO PERSON FOUND.
```

Further Examples of FIND Statement

See the following example programs:

- *FINDX07 - FIND (with several clauses)*
- *FINDX08 - FIND (with LIMIT)*
- *FINDX09 - FIND (using *NUMBER, *COUNTER, *ISN)*
- *FINDX10 - FIND (combined with READ)*
- *FINDX11 - FIND NUMBER (with *NUMBER)*

HISTOGRAM Statement

The following topics are covered:

Use of HISTOGRAM Statement

The `HISTOGRAM` statement is used to either read only the values of one database field, or determine the number of records which meet a specified search criterion.

The `HISTOGRAM` statement does not provide access to any database fields other than the one specified in the `HISTOGRAM` statement.

Syntax of HISTOGRAM Statement

The basic syntax of the `HISTOGRAM` statement is:

```
HISTOGRAM VALUE IN view FOR field
```

or shorter:

```
HISTOGRAM view FOR field
```

- where

view	is the name of a view as defined in the `DEFINE DATA` statement and as explained in ***Defining a Database View***.
field	is the name of a database field as defined in that view.

For the complete syntax, refer to the `HISTOGRAM` statement documentation.

Limiting the Number of Values to be Read

In the same way as with the `READ` statement, you can limit the number of values to be read by specifying a number in parentheses after the keyword `HISTOGRAM`:

```
HISTOGRAM (6) MYVIEW FOR NAME
```

In the above example, only the first 6 values of the field `NAME` would be read.

Without the limit notation, all values would be read.

STARTING/ENDING Clauses

Like the `READ` statement, the `HISTOGRAM` statement also provides a `STARTING FROM` clause and an `ENDING AT (or THRU)` clause to narrow down the range of values to be read by specifying a starting value and ending value.

Examples:

```
HISTOGRAM MYVIEW FOR NAME STARTING from 'BOUCHARD'
HISTOGRAM MYVIEW FOR NAME STARTING from 'BOUCHARD' ENDING AT 'LANIER'
HISTOGRAM MYVIEW FOR NAME from 'BLOOM' THRU 'ROESER'
```

WHERE Clause

The HISTOGRAM statement also provides a WHERE clause which may be used to specify an additional selection criterion that is evaluated *after* a value has been read and *before* any processing is performed on the value. The field specified in the WHERE clause must be the same as in the main clause of the HISTOGRAM statement.

Example of HISTOGRAM Statement

```
** Example 'HISTOX01': HISTOGRAM
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
END-DEFINE
*
LIMIT 8
HISTOGRAM MYVIEW CITY STARTING FROM 'M'
  DISPLAY NOTITLE CITY 'NUMBER OF/PERSONS' *NUMBER *COUNTER
END-HISTOGRAM
END
```

In this program, the system variables *NUMBER and *COUNTER are also evaluated by the HISTOGRAM statement, and output with the DISPLAY statement. *NUMBER contains the number of database records that contain the last value read; *COUNTER contains the total number of values which have been read.

Output of Program HISTOX01:

```
        CITY          NUMBER OF     CNT
                       PERSONS
-------------------- ----------- -----------

MADISON                        3           1
MADRID                        41           2
MAILLY LE CAMP                 1           3
MAMERS                         1           4
MANSFIELD                      4           5
MARSEILLE                      2           6
MATLOCK                        1           7
MELBOURNE                      2           8
```

MULTI-FETCH Clause

The `MULTI-FETCH` clause supports the multi-fetch record retrieval functionality for Adabas databases.

The multi-fetch functionality described in this section is supported for databases of type `ADA`/`ADA2`, which can be defined in the DBMS Assignments table in the Configuration Utility; see *Database Management System Assignments* in the *Configuration Utility* documentation.

The multi-fetch clause is not supported

- when Adabas LA or large objects fields are used or
- when view sizes greater than 64MB are defined.

The following topics are covered:

Purpose of Multi-Fetch Feature

In standard mode, Natural does not read multiple records with a single database call; it always operates in a one-record-per-fetch mode. This kind of operation is solid and stable, but can take some time if a large number of database records are being processed. To improve the performance of those programs, you can use multi-fetch processing.

By default, Natural uses single-fetch to retrieve data from Adabas databases. This default can be configured using the Natural profile parameter `MFSET`.

Values `ON` (multi-fetch) and `OFF` (single-fetch) define the default behavior. If `MFSET` is set to `NEVER`, Natural always uses single-fetch mode and ignores any settings at statement level.

The default processing mode can also be overridden at statement level.

Supported Statements

The multi-fetch processing of database records is supported for the following statements that do not modify the database:

- `FIND`
- `READ`
- `HISTOGRAM`

For more information about the syntax of the `MULTI-FETCH` clause in the supported statements, see `FIND`, `READ` or `HISTOGRAM`.

You can use the `MULTI-FETCH` clause to improve the performance of the supported statements by defining the number of records read per database access as a numeric value called the *multi-fetch factor*.

```
{ FIND      } [              { ON                      } ]
{ READ      } [ MULTI-FETCH  { OFF                     } ]
{ HISTOGRAM } [              { [OF] multi-fetch-factor } ]
```

Valid values of the *multi-fetch-factor* are:

- a numeric constant in the value range (0-2147483647).
- a variable in integer (I1, I2, I4), binary (B1-B4), or packed/numeric with only integer digits format.

Based on the value of the *multi-fetch-factor* specified in the database statement, the database call is processed as follows:

Value	Database call process description
Negative	A negative value is out of range and results in a run-time error.
0 or 1	A value of 0 or 1 indicates to process one record per database access (which is the standard processing mode).
2 or greater	A value of 2 or greater indicates that the database call is dynamically prepared to read multiple records with a single database access and store them in the multi-fetch buffer. If successful, the first record is transferred to the underlying data view. In the next loop, the data view is filled directly from the multi-fetch buffer, without accessing the database. After fetching all records stored in the multi-fetch buffer, the database call reads the next set of records from the database. If the database loop is terminated by an action, such as end-of-records, `ESCAPE`, or `STOP`, the content of the multi-fetch buffer is released.

Adapting the Multi-Fetch Parameters

There are three Natural profile parameters that influence the behavior of the multi-fetch feature:

Name	Description
MFBS	Sets the maximum buffer size used for fetching records in multi-fetch-mode.
MFMR	Sets the maximum number of records being fetched with one multi-fetch call.
MFSET	Switches between multi-fetch and single-fetch mode.

Note: The actual number of records being fetched with one multi-fetch call is calculated during runtime, so that both limits in MFBS and MFMR will be respected.

If your application is strongly based on reading large amounts of data from an Adabas database, then using multi-fetch mode can fairly increase the overall performance. But keep in mind that in multi-fetch mode the data in a READ loop, for example, is in most cases provided from Natural's internal multi-fetch buffer and not directly from the database anymore. Direct Adabas calls get more seldom the higher you set the MFBS and MFMR parameters. So in single-fetch mode you work with he most current state of data in the database, while in multi-fetch mode you always work with a slightly older state of the data.

Consequently, consider using multi-fetch mode in batch oriented applications where performance is a main factor. In interactive applications where the data retrieval performance might be secondary - go with single-fetch mode.

Considerations for Multi-Fetch Usage

If nested database loops that refer to the same Adabas file contain UPDATE statements in one of the inner loops, Natural continues processing the outer loops with the updated values. This implies in multi-fetch mode, that an outer logical READ loop has to be repositioned if an inner database loop updates the value of the descriptor that is used for sequence control in the outer loop. If this attempt leads to a conflict for the current descriptor, an error is returned. To avoid this situation, we recommend that you disable multi-fetch in the outer database loops.

In general, multi-fetch mode improves performance when accessing Adabas databases. In some cases, however, it might be advantageous to use single-fetch to enhance performance, especially if database modifications are involved.

Database Processing Loops

This section discusses processing loops required to process data that have been selected from a database as a result of a FIND, READ or HISTOGRAM statement.

The following topics are covered:

Creation of Database Processing Loops

Natural automatically creates the necessary processing loops which are required to process data that have been selected from a database as a result of a FIND, READ or HISTOGRAM statement.

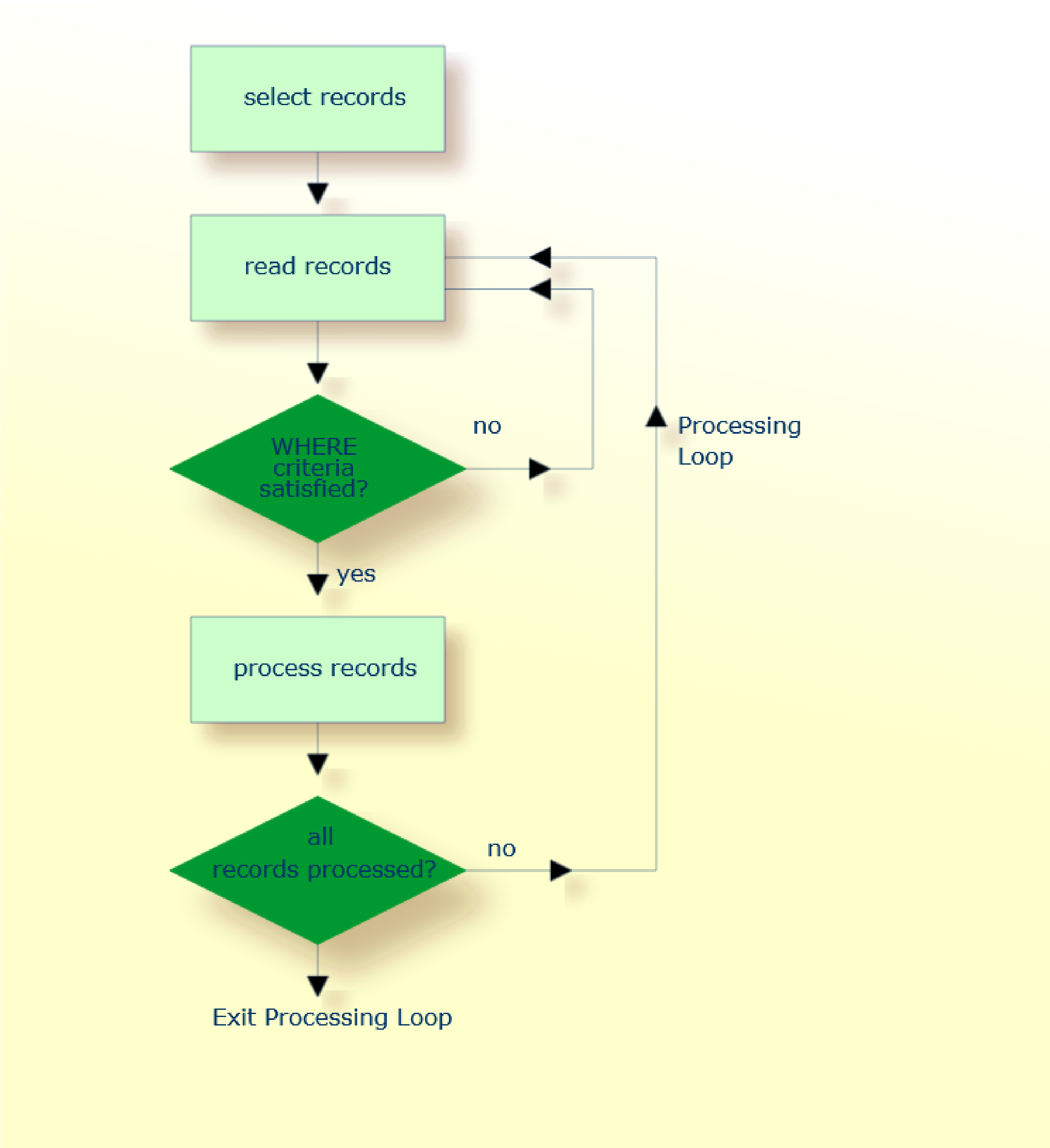
Example:

In the following example, the FIND loop selects all records from the EMPLOYEES file in which the field NAME contains the value ADKINSON and processes the selected records. In this example, the processing consists of displaying certain fields from each record selected.

```
** Example 'FINDX03': FIND
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
END-DEFINE
*
FIND MYVIEW WITH NAME = 'ADKINSON'
  DISPLAY NAME FIRST-NAME CITY
END-FIND
END
```

If the FIND statement contained a WHERE clause in addition to the WITH clause, only those records that were selected as a result of the WITH clause *and* met the WHERE criteria would be processed.

The following diagram illustrates the flow logic of a database processing loop:


select records
read records
WHERE criteria satisfied?
no
yes
Processing Loop
process records
all records processed?
no
Exit Processing Loop

Hierarchies of Processing Loops

The use of multiple FIND and/or READ statements creates a hierarchy of processing loops, as shown in the following example:

Example of Processing Loop Hierarchy

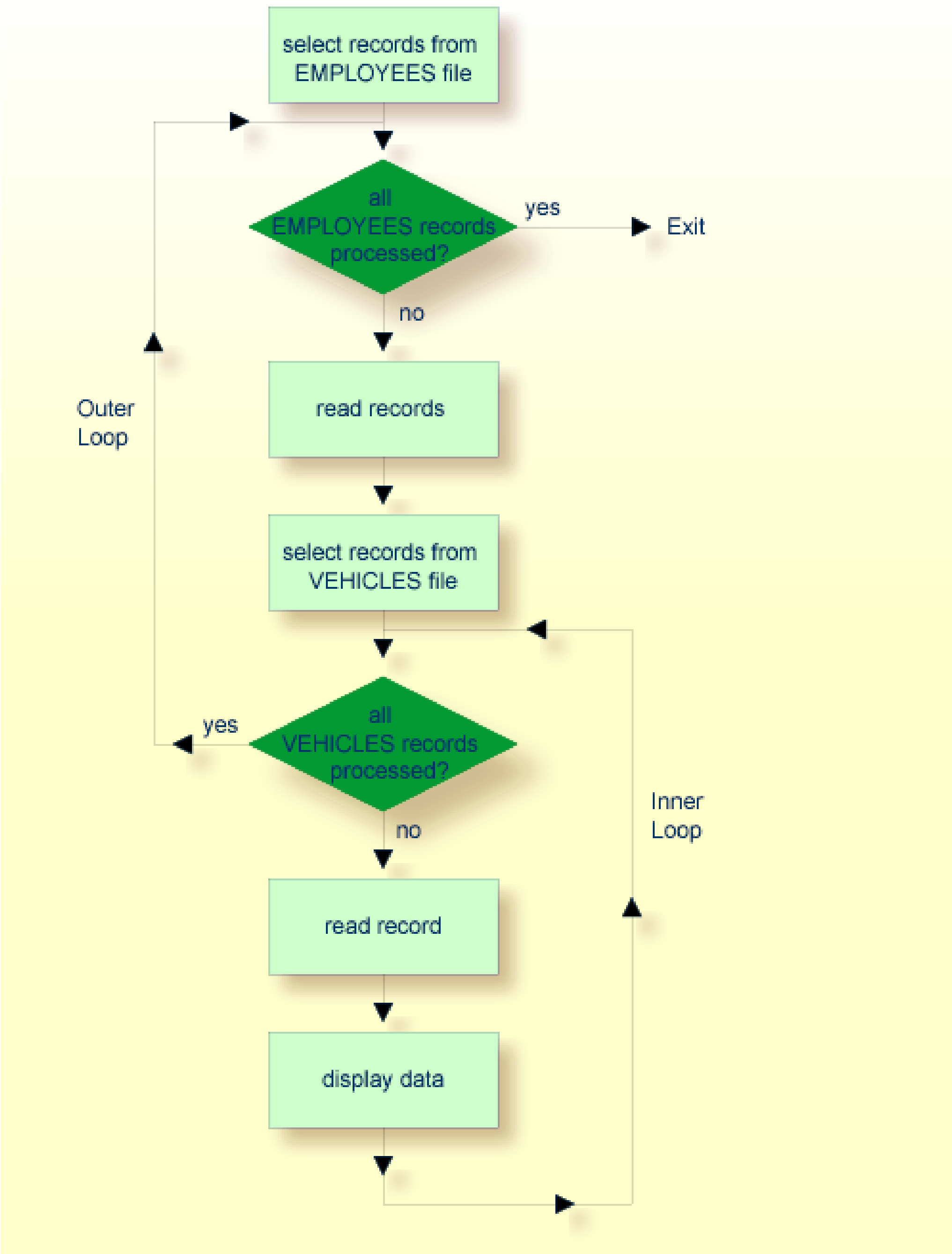
```
** Example 'FINDX04': FIND  (two FIND statements nested)
************************************************************************
DEFINE DATA LOCAL
1 PERSONVIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
1 AUTOVIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
  2 MODEL
END-DEFINE
*
EMP. FIND PERSONVIEW WITH NAME = 'ADKINSON'
  VEH. FIND AUTOVIEW WITH PERSONNEL-ID = PERSONNEL-ID (EMP.)
    DISPLAY NAME MAKE MODEL
  END-FIND
END-FIND
END
```

The above program selects from the EMPLOYEES file all people with the name ADKINSON. Each record (person) selected is then processed as follows:

1. The second FIND statement is executed to select the automobiles from the VEHICLES file, using as selection criterion the PERSONNEL-IDs from the records selected from the EMPLOYEES file with the first FIND statement.
2. The NAME of each person selected is displayed; this information is obtained from the EMPLOYEES file. The MAKE and MODEL of each automobile owned by that person is also displayed; this information is obtained from the VEHICLES file.

The second FIND statement creates an inner processing loop within the outer processing loop of the first FIND statement, as shown in the following diagram.

The diagram illustrates the flow logic of the hierarchy of processing loops in the previous example program:



select records from EMPLOYEES file
all EMPLOYEES records processed?
yes
Exit
no
read records
select records from VEHICLES file
all VEHICLES records processed?
yes
no
read record
display data
Outer Loop
Inner Loop

Example of Nested FIND Loops Accessing the Same File

It is also possible to construct a processing loop hierarchy in which the same file is used at both levels of the hierarchy:

```
** Example 'FINDX05': FIND (two FIND statements on same file nested)
************************************************************************
DEFINE DATA LOCAL
1 PERSONVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
1 #NAME (A40)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED
  'PEOPLE IN SAME CITY AS:' #NAME / 'CITY:' CITY SKIP 1
*
FIND PERSONVIEW WITH NAME = 'JONES'
                WHERE FIRST-NAME = 'LAUREL'
  COMPRESS NAME FIRST-NAME INTO #NAME
  /*
  FIND PERSONVIEW WITH CITY = CITY
    DISPLAY NAME FIRST-NAME CITY
  END-FIND
END-FIND
END
```

The above program first selects all people with name `JONES` and first name `LAUREL` from the `EMPLOYEES` file. Then all who live in the same city are selected from the `EMPLOYEES` file and a list of these people is created. All field values displayed by the `DISPLAY` statement are taken from the second `FIND` statement.

Output of Program `FINDX05`:

```
PEOPLE IN SAME CITY AS: JONES LAUREL
CITY: BALTIMORE

        NAME              FIRST-NAME              CITY
-------------------- -------------------- --------------------

JENSON               MARTHA               BALTIMORE
LAWLER               EDDIE                BALTIMORE
FORREST              CLARA                BALTIMORE
ALEXANDER            GIL                  BALTIMORE
NEEDHAM              SUNNY                BALTIMORE
ZINN                 CARLOS               BALTIMORE
JONES                LAUREL               BALTIMORE
```

Further Examples of Nested READ and FIND Statements

See the following example programs:

- *READX04 - READ statement (in combination with FIND and the system variables *NUMBER and *COUNTER)*
- *LIMITX01 - LIMIT statement (for READ, FIND loop processing)*

Database Update - Transaction Processing

This section describes how Natural performs database updating operations based on transactions.

The following topics are covered:

Logical Transaction

Natural performs database updating operations based on transactions, which means that all database update requests are processed in logical transaction units. A logical transaction is the smallest unit of work (as defined by you) which must be performed in its entirety to ensure that the information contained in the database is logically consistent.

A logical transaction may consist of one or more update statements (`DELETE`, `STORE`, `UPDATE`) involving one or more database files. A logical transaction may also span multiple Natural programs.

A logical transaction begins when a record is put on “hold”; Natural does this automatically when the record is read for updating, for example, if a `FIND` loop contains an `UPDATE` or `DELETE` statement.

The end of a logical transaction is determined by an `END TRANSACTION` statement in the program. This statement ensures that all updates within the transaction have been successfully applied, and releases all records that were put on “hold” during the transaction.

Example:

```
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
END-DEFINE
FIND MYVIEW WITH NAME = 'SMITH'
   DELETE
   END TRANSACTION
END-FIND
END
```

Each record selected would be put on “hold”, deleted, and then - when the END TRANSACTION statement is executed - released from “hold”.

Note: The Natural profile parameter ETEOP, as set by the Natural administrator, determines whether or not Natural will generate an END TRANSACTION statement at the end of each Natural program. Ask your Natural administrator for details.

Example of STORE Statement:

The following example program adds new records to the EMPLOYEES file.

```
** Example 'STOREX01': STORE  (Add new records to EMPLOYEES file)
*
** CAUTION: Executing this example will modify the database records!
************************************************************************
DEFINE DATA LOCAL
1 EMPLOYEE-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID(A8)
  2 NAME        (A20)
  2 FIRST-NAME  (A20)
  2 MIDDLE-I    (A1)
  2 SALARY      (P9/2)
  2 MAR-STAT    (A1)
  2 BIRTH       (D)
  2 CITY        (A20)
  2 COUNTRY     (A3)
*
1 #PERSONNEL-ID (A8)
1 #NAME         (A20)
1 #FIRST-NAME   (A20)
1 #INITIAL      (A1)
1 #MAR-STAT     (A1)
1 #SALARY       (N9)
1 #BIRTH        (A8)
1 #CITY         (A20)
1 #COUNTRY      (A3)
1 #CONF         (A1)   INIT <'Y'>
END-DEFINE
```

```
*
REPEAT
  INPUT 'ENTER A PERSONNEL ID AND NAME (OR ''END'' TO END)' //
        'PERSONNEL-ID : ' #PERSONNEL-ID    //
        'NAME         : ' #NAME  /
        'FIRST-NAME   : ' #FIRST-NAME
  /*********************************************************************
  /*  validate entered data
  /*********************************************************************
  IF #PERSONNEL-ID = 'END' OR #NAME = 'END'
    STOP
  END-IF
  IF #NAME = ' '
    REINPUT WITH TEXT 'ENTER A LAST-NAME'
            MARK 2 AND SOUND ALARM
  END-IF
  IF #FIRST-NAME = ' '
    REINPUT WITH TEXT 'ENTER A FIRST-NAME'
            MARK 3 AND SOUND ALARM
  END-IF
  /*********************************************************************
  /*  ensure person is not already on file
  /*********************************************************************
  FIP2. FIND NUMBER EMPLOYEE-VIEW WITH PERSONNEL-ID =  #PERSONNEL-ID
  /*
  IF *NUMBER (FIP2.) > 0
    REINPUT 'PERSON WITH SAME PERSONNEL-ID ALREADY EXISTS'
             MARK 1 AND SOUND ALARM
  END-IF
  /*********************************************************************
  /*  get further information
  /*********************************************************************
  INPUT
    'ENTER EMPLOYEE DATA'                            ////
    'PERSONNEL-ID             :' #PERSONNEL-ID (AD=IO) /
    'NAME                     :' #NAME         (AD=IO) /
    'FIRST-NAME               :' #FIRST-NAME   (AD=IO) ///
    'INITIAL                  :' #INITIAL              /
    'ANNUAL SALARY            :' #SALARY               /
    'MARITAL STATUS           :' #MAR-STAT             /
    'DATE OF BIRTH (YYYYMMDD) :' #BIRTH                /
    'CITY                     :' #CITY                 /
    'COUNTRY (3 CHARS)        :' #COUNTRY              //
    'ADD THIS RECORD (Y/N)    :' #CONF         (AD=M)
  /*********************************************************************
  /*  ENSURE REQUIRED FIELDS CONTAIN VALID DATA
  /*********************************************************************
  IF #SALARY < 10000
    REINPUT TEXT 'ENTER A PROPER ANNUAL SALARY' MARK 2
  END-IF
  IF NOT (#MAR-STAT = 'S'  OR = 'M' OR = 'D' OR = 'W')
    REINPUT TEXT 'ENTER VALID MARITAL STATUS  S=SINGLE ' -
```

```
                 'M=MARRIED D=DIVORCED W=WIDOWED' MARK 3
  END-IF
  IF NOT(#BIRTH = MASK(YYYYMMDD) AND #BIRTH = MASK(1582-2699))
    REINPUT TEXT 'ENTER CORRECT DATE' MARK 4
  END-IF
  IF #CITY  = ' '
    REINPUT TEXT 'ENTER A CITY NAME' MARK 5
  END-IF
  IF #COUNTRY = ' '
    REINPUT TEXT 'ENTER A COUNTRY CODE' MARK 6
  END-IF
  IF NOT (#CONF = 'N' OR= 'Y')
    REINPUT TEXT 'ENTER Y (YES) OR N (NO)' MARK 7
  END-IF
  IF #CONF = 'N'
    ESCAPE TOP
  END-IF
  /*********************************************************************
  /*  add the record with STORE
  /*********************************************************************
  MOVE #PERSONNEL-ID TO EMPLOYEE-VIEW.PERSONNEL-ID
  MOVE #NAME         TO EMPLOYEE-VIEW.NAME
  MOVE #FIRST-NAME   TO EMPLOYEE-VIEW.FIRST-NAME
  MOVE #INITIAL      TO EMPLOYEE-VIEW.MIDDLE-I
  MOVE #SALARY       TO EMPLOYEE-VIEW.SALARY (1)
  MOVE #MAR-STAT     TO EMPLOYEE-VIEW.MAR-STAT
  MOVE EDITED #BIRTH TO EMPLOYEE-VIEW.BIRTH (EM=YYYYMMDD)
  MOVE #CITY         TO EMPLOYEE-VIEW.CITY
  MOVE #COUNTRY      TO EMPLOYEE-VIEW.COUNTRY
  /*
  STP3. STORE RECORD IN FILE EMPLOYEE-VIEW
  /*
  /*********************************************************************
  /*  mark end of logical transaction
  /*********************************************************************
  END OF TRANSACTION
  RESET INITIAL #CONF
END-REPEAT
END
```

Output of Program `STOREX01`:

```
ENTER A PERSONNEL ID AND NAME (OR 'END' TO END)

PERSONNEL ID :

NAME         :
FIRST NAME   :
```

Record Hold Logic

If Natural is used with Adabas, any record which is to be updated will be placed in “hold” status until an END TRANSACTION or BACKOUT TRANSACTION statement is issued or the transaction time limit is exceeded.

When a record is placed in “hold” status for one user, the record is not available for update by another user. Another user who wishes to update the same record will be placed in “wait” status until the record is released from “hold” when the first user ends or backs out his/her transaction.

To prevent users from being placed in wait status, the session parameter WH (Wait for Record in Hold Status) can be used (see the *Parameter Reference*).

When you use update logic in a program, you should consider the following:

- The maximum time that a record can be in hold status is determined by the Adabas transaction time limit (Adabas parameter TT). If this time limit is exceeded, you will receive an error message and all database modifications done since the last END TRANSACTION will be made undone.
- The number of records on hold and the transaction time limit are affected by the size of a transaction, that is, by the placement of the END TRANSACTION statement in the program. Restart facilities should be considered when deciding where to issue an END TRANSACTION. For example, if a majority of records being processed are *not* to be updated, the GET statement is an efficient way of controlling the “holding” of records. This avoids issuing multiple END TRANSACTION statements and reduces the number of ISNs on hold. When you process large files, you should bear in mind that the GET statement requires an additional Adabas call. An example of a GET statement is shown below.

Example of Hold Logic:

```
** Example 'GETX01': GET (put single record in hold with UPDATE stmt)
**
** CAUTION: Executing this example will modify the database records!
***********************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 SALARY (1)
END-DEFINE
*
RD. READ EMPLOY-VIEW BY NAME
  DISPLAY EMPLOY-VIEW
  IF SALARY (1) > 1500000
    /*
    GE. GET EMPLOY-VIEW *ISN (RD.)
    /*
    WRITE '=' (50) 'RECORD IN HOLD:' *ISN(RD.)
    COMPUTE SALARY (1) = SALARY (1) * 1.15
    UPDATE (GE.)
```

```
    END TRANSACTION
  END-IF
END-READ
END
```

Backing Out a Transaction

During an active logical transaction, that is, before the `END TRANSACTION` statement is issued, you can cancel the transaction by using a `BACKOUT TRANSACTION` statement. The execution of this statement removes all updates that have been applied (including all records that have been added or deleted) and releases all records held by the transaction.

Restarting a Transaction

With the `END TRANSACTION` statement, you can also store transaction-related information. If processing of the transaction terminates abnormally, you can read this information with a `GET TRANSACTION DATA` statement to ascertain where to resume processing when you restart the transaction.

Example of Using Transaction Data to Restart a Transaction

The following program updates the `EMPLOYEES` and `VEHICLES` files. After a restart operation, the user is informed of the last `EMPLOYEES` record successfully processed. The user can resume processing from that `EMPLOYEES` record. It would also be possible to set up the restart transaction message to include the last `VEHICLES` record successfully updated before the restart operation.

```
** Example 'GETTRX01': GET TRANSACTION
*
** CAUTION: Executing this example will modify the database records!
************************************************************************
DEFINE DATA LOCAL
01 PERSON VIEW OF EMPLOYEES
  02 PERSONNEL-ID      (A8)
  02 NAME              (A20)
  02 FIRST-NAME        (A20)
  02 MIDDLE-I          (A1)
  02 CITY              (A20)
01 AUTO VIEW OF VEHICLES
  02 PERSONNEL-ID      (A8)
  02 MAKE              (A20)
  02 MODEL             (A20)
*
01 ET-DATA
  02 #APPL-ID          (A8) INIT <' '>
  02 #USER-ID          (A8)
  02 #PROGRAM          (A8)
  02 #DATE             (A10)
  02 #TIME             (A8)
```

```
  02 #PERSONNEL-NUMBER (A8)
END-DEFINE
*
GET TRANSACTION DATA #APPL-ID #USER-ID #PROGRAM
                     #DATE    #TIME    #PERSONNEL-NUMBER
*
IF  #APPL-ID NOT = 'NORMAL'     /* if last execution ended abnormally
AND #APPL-ID NOT = ' '
  INPUT (AD=OIL)
    //  20T '*** LAST SUCCESSFUL TRANSACTION ***' (I)
    /   20T '***********************************'
    /// 25T      'APPLICATION:' #APPL-ID
    /   32T             'USER:' #USER-ID
    /   29T          'PROGRAM:' #PROGRAM
    /   24T     'COMPLETED ON:' #DATE 'AT' #TIME
    /   20T 'PERSONNEL NUMBER:' #PERSONNEL-NUMBER
END-IF
*
REPEAT
  /*
  INPUT (AD=MIL) // 20T 'ENTER PERSONNEL NUMBER:' #PERSONNEL-NUMBER
  /*
  IF #PERSONNEL-NUMBER = '99999999'
    ESCAPE BOTTOM
  END-IF
  /*
  FIND1. FIND PERSON WITH PERSONNEL-ID = #PERSONNEL-NUMBER
    IF NO RECORDS FOUND
      REINPUT 'SPECIFIED NUMBER DOES NOT EXIST; ENTER ANOTHER ONE.'
    END-NOREC
    FIND2. FIND AUTO WITH PERSONNEL-ID = #PERSONNEL-NUMBER
      IF NO RECORDS FOUND
        WRITE 'PERSON DOES NOT OWN ANY CARS'
        ESCAPE BOTTOM
      END-NOREC
      IF *COUNTER (FIND2.) = 1     /* first pass through the loop
        INPUT (AD=M)
          /   20T 'EMPLOYEES/AUTOMOBILE DETAILS' (I)
          /   20T '----------------------------'
          /// 20T 'NUMBER:' PERSONNEL-ID (AD=O)
          /   22T   'NAME:' NAME ' ' FIRST-NAME ' ' MIDDLE-I
          /   22T   'CITY:' CITY
          /   22T   'MAKE:' MAKE
          /   21T  'MODEL:' MODEL
        UPDATE (FIND1.)            /* update the EMPLOYEES file
      ELSE                         /* subsequent passes through the loop
        INPUT NO ERASE (AD=M IP=OFF) //////// 28T MAKE / 28T MODEL
      END-IF
      /*
      UPDATE (FIND2.)              /* update the VEHICLES file
      /*
      MOVE *APPLIC-ID TO #APPL-ID
```

```
      MOVE *INIT-USER TO #USER-ID
      MOVE *PROGRAM   TO #PROGRAM
      MOVE *DAT4E     TO #DATE
      MOVE *TIME      TO #TIME
      /*
      END TRANSACTION #APPL-ID #USER-ID #PROGRAM
                     #DATE    #TIME    #PERSONNEL-NUMBER
      /*
    END-FIND                  /* for VEHICLES    (FIND2.)
  END-FIND                    /* for EMPLOYEES   (FIND1.)
END-REPEAT                    /* for REPEAT
*
STOP                          /* Simulate abnormal transaction end
END TRANSACTION 'NORMAL  '
END
```

Selecting Records Using ACCEPT/REJECT

This section discusses the statements ACCEPT and REJECT which are used to select records based on user-specified logical criteria.

The following topics are covered:

Statements Usable with ACCEPT and REJECT

The statements ACCEPT and REJECT can be used in conjunction with the database access statements:

- READ
- FIND
- HISTOGRAM

Example of ACCEPT Statement

```
** Example 'ACCEPX01': ACCEPT IF
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 CURR-CODE (1:1)
  2 SALARY    (1:1)
END-DEFINE
*
READ (20) MYVIEW BY NAME WHERE CURR-CODE (1) = 'USD'
ACCEPT IF SALARY (1) >= 40000
  DISPLAY NAME JOB-TITLE SALARY (1)
END-READ
END
```

Output of Program ACCEPX01:

```
  Page      1                                                    04-11-11  11:11:11

        NAME                 CURRENT            ANNUAL
                            POSITION            SALARY
  -------------------- ------------------------- ----------


  ADKINSON             DBA                            46700
  ADKINSON             MANAGER                        47000
  ADKINSON             MANAGER                        47000
  AFANASSIEV           DBA                            42800
  ALEXANDER            DIRECTOR                       48000
  ANDERSON             MANAGER                        50000
  ATHERTON             ANALYST                        43000
  ATHERTON             MANAGER                        40000
```

Logical Condition Criteria in ACCEPT/REJECT Statements

The statements ACCEPT and REJECT allow you to specify logical conditions in addition to those that were specified in WITH and WHERE clauses of the READ statement.

The logical condition criteria in the IF clause of an ACCEPT / REJECT statement are evaluated *after* the record has been selected and read.

Logical condition operators include the following (see ***Logical Condition Criteria*** for more detailed information):

EQUAL	EQ	:=
NOT EQUAL TO	NE	¬=
LESS THAN	LT	<
LESS EQUAL	LE	<=
GREATER THAN	GT	>
GREATER EQUAL	GE	>=

Logical condition criteria in ACCEPT / REJECT statements may also be connected with the Boolean operators AND, OR, and NOT. Moreover, parentheses may be used to indicate logical grouping; see the following examples.

Example of ACCEPT Statement with AND Operator

The following program illustrates the use of the Boolean operator AND in an ACCEPT statement.

```
** Example 'ACCEPX02': ACCEPT IF ... AND ...
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 CURR-CODE (1:1)
  2 SALARY    (1:1)
END-DEFINE
*
READ (20) MYVIEW BY NAME WHERE CURR-CODE (1) = 'USD'
ACCEPT IF  SALARY (1) >= 40000
         AND SALARY (1) <= 45000
  DISPLAY NAME JOB-TITLE SALARY (1)
END-READ
END
```

Output of Program ACCEPX02:

```
Page      1                                             04-12-14  12:22:01

        NAME                 CURRENT            ANNUAL
                             POSITION           SALARY
-------------------- ------------------------- ----------

AFANASSIEV           DBA                            42800
ATHERTON             ANALYST                        43000
ATHERTON             MANAGER                        40000                  ↩
```

Example of REJECT Statement with OR Operator

The following program, which uses the Boolean operator `OR` in a `REJECT` statement, produces the same output as the `ACCEPT` statement in the example above, as the logical operators are reversed.

```
** Example 'ACCEPX03': REJECT IF ... OR ...
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 CURR-CODE (1:1)
  2 SALARY    (1:1)
END-DEFINE
*
READ (20) MYVIEW BY NAME WHERE CURR-CODE (1) = 'USD'
REJECT IF SALARY (1) < 40000
         OR SALARY (1) > 45000
  DISPLAY NAME JOB-TITLE SALARY (1)
END-READ
END
```

Output of Program `ACCEPX03`:

```
Page      1                                                  04-12-14  12:26:27

        NAME                  CURRENT            ANNUAL
                             POSITION            SALARY
-------------------- ------------------------- ----------

AFANASSIEV           DBA                            42800
ATHERTON             ANALYST                        43000
ATHERTON             MANAGER                        40000                     ↩
```

Further Examples of ACCEPT and REJECT Statements

See the following example programs:

- *ACCEPX04 - ACCEPT IF ... LESS THAN ...*
- *ACCEPX05 - ACCEPT IF ... AND ...*
- *ACCEPX06 - REJECT IF ... OR ...*

AT START/END OF DATA Statements

This section discusses the use of the statements `AT START OF DATA` and `AT END OF DATA`.

The following topics are covered:

AT START OF DATA Statement

The `AT START OF DATA` statement is used to specify any processing that is to be performed after the first of a set of records has been read in a database processing loop.

The `AT START OF DATA` statement must be placed within the processing loop.

If the `AT START OF DATA` processing produces any output, this will be output *before the first field value*. By default, this output is displayed left-justified on the page.

AT END OF DATA Statement

The `AT END OF DATA` statement is used to specify processing that is to be performed after all records for a database processing loop have been processed.

The `AT END OF DATA` statement must be placed within the processing loop.

If the `AT END OF DATA` processing produces any output, this will be output *after the last field value*. By default, this output is displayed left-justified on the page.

Example of AT START OF DATA and AT END OF DATA Statements

The following example program illustrates the use of the statements `AT START OF DATA` and `AT END OF DATA`.

The Natural system variable `*TIME` has been incorporated into the `AT START OF DATA` statement to display the time of day.

The Natural system function `OLD` has been incorporated into the `AT END OF DATA` statement to display the name of the last person selected.

```
** Example 'ATSTAX01': AT START OF DATA
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
  2 INCOME  (1:1)
    3 CURR-CODE
    3 SALARY
    3 BONUS (1:1)
END-DEFINE
*
WRITE TITLE 'XYZ EMPLOYEE ANNUAL SALARY AND BONUS REPORT' /
READ (3) MYVIEW BY CITY STARTING FROM 'E'
  DISPLAY GIVE SYSTEM FUNCTIONS
          NAME (AL=15) JOB-TITLE (AL=15) INCOME (1)
  /*
 AT START OF DATA
    WRITE 'RUN TIME:' *TIME /
  END-START
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD (NAME) /
  END-ENDDATA
END-READ
*
AT END OF PAGE
  WRITE / 'AVERAGE SALARY:' AVER (SALARY(1))
END-ENDPAGE
END
```

The program produces the following output:

```
                  XYZ EMPLOYEE ANNUAL SALARY AND BONUS REPORT

     NAME           CURRENT                 INCOME
                   POSITION
                                CURRENCY   ANNUAL     BONUS
                                  CODE     SALARY
--------------- --------------- -------- ---------- ----------

RUN TIME: 12:43:19.1

DUYVERMAN       PROGRAMMER      USD           34000          0
PRATT           SALES PERSON    USD           38000       9000
MARKUSH         TRAINEE         USD           22000          0

LAST PERSON SELECTED: MARKUSH


AVERAGE SALARY:      31333                                  ↩
```

Further Examples of AT START OF DATA and AT END OF DATA

See the following example programs:

- *ATENDX01 - AT END OF DATA*
- *ATSTAX02 - AT START OF DATA*
- *WRITEX09 - WRITE (in combination with AT END OF DATA)*

Unicode Data

Natural enables users to access wide-character fields (format W) in an Adabas database.

The following topics are covered:

Data Definition Module

Adabas wide-character fields (W) are mapped to Natural format U (Unicode).

The length definition for a Natural field of format U corresponds to half the size of the Adabas field of format W. An Adabas wide-character field of length `200` is, for example, mapped to `(U100)` in Natural.

Access Configuration

Natural receives data from Adabas and sends data to Adabas using UTF-16 as common encoding.

This encoding is specified with the `OPRB` parameter and sent to Adabas with the open request. It is used for wide-character fields and applies to the entire Adabas user session.

Restrictions

Wide-character fields (W) of variable length are not supported.

Collating descriptors are not supported.

For further information on Adabas and Unicode support refer to the specific Adabas product documentation.

28 Accessing Data in an SQL Database

This chapter describes how to use Natural with SQL databases via Entire Access. For information about installation and configuration, see *Natural and Entire Access* in the *Database Management System Interfaces* documentation and the separate Entire Access documentation.

Note: On principle, the features and examples contained in the document ***Accessing Data in an Adabas Database*** also apply to the SQL databases supported by Natural. Differences, if any, are described in the documents for the individual database access statements (see the *Statements* documentation) in paragraphs named *Database-Specific Considerations* or in the documents for the individual Natural parameters (see the *Parameter Reference*). In addition, Natural offers a specific set of statements to access SQL databases.

Generating Natural DDMs

Entire Access is an application programming interface (API) that supports Natural SQL statements and most Natural DML statements (see the *Statements* documentation).

Natural DML and SQL statements can be used in the same Natural program. At compilation, if a DML statement references a DDM for a data source defined in *NATCONF.CFG* with DBMS type `SQL`, Natural translates the DML statement into an SQL statement.

Natural converts DML and SQL statements into calls to Entire Access. Entire Access converts the requests to the data formats and SQL dialect required by the target RDBMS and passes the requests to the database driver.

Setting Natural Profile Parameters

ETEOP Parameter

This parameter can be set only by Natural administrators.

The Natural profile parameter `ETEOP` controls transaction processing during a Natural session. It is required, for example, if a single logical transaction is to span two or more Natural programs. In this case, Natural must not issue an `END TRANSACTION` command (that is, not “commit”) at the termination of a Natural program.

If the `ETEOP` parameter is set to:

ON	Natural issues an END TRANSACTION statement (that is, automatically "commits") at the end of a Natural program if the Natural session is not at ET status.
OFF	Natural does not issue an END TRANSACTION command (that is, does not "commit") at the end of a Natural program. This setting thus enables a single logical transaction to span more than one Natural program. This is the default.

Note: The ETEOP parameter applies to Natural Version 6.1 and above. With previous Natural versions, the Natural profile parameter OPRB has to be used instead of ETEOP (ETEOP=ON corresponds to OPRB=OFF, ETEOP=OFF corresponds to ORPB=NOOPEN).

Natural DML Statements

The following table shows how Natural translates DML statements into SQL statements:

DML Statement	SQL Statement
BACKOUT TRANSACTION	ROLLBACK
DELETE	DELETE WHERE CURRENT OF *cursor-name*
END TRANSACTION	COMMIT
EQUAL ... OR	IN (...)
EQUAL ... THRU ...	BETWEEN ... AND ...
FIND ALL	SELECT
FIND NUMBER	SELECT COUNT (*)
HISTOGRAM	SELECT COUNT (*)
READ LOGICAL	SELECT ... ORDER BY
READ PHYSICAL	SELECT ... ORDER BY
SORTED BY ... [DESCENDING]	ORDER BY ... [DESCENDING]
STORE	INSERT
UPDATE	UPDATE WHERE CURRENT of *cursor-name*
WITH	WHERE

Note: Boolean and relational operators function the same way in DML and SQL statements.

Entire Access does not support the following DML statements and options:

- CIPHER
- COUPLED
- FIND FIRST, FIND UNIQUE, FIND ... RETAIN AS

- GET, GET SAME, GET TRANSACTION DATA, GET RECORD
- PASSWORD
- READ BY ISN
- STORE USING/GIVING NUMBER

BACKOUT TRANSACTION

Natural translates a BACKOUT TRANSACTION statement into an SQL ROLLBACK command. This statement reverses all database modifications made after the completion of the last recovery unit. A recovery unit may start at the beginning of a session or after the last END TRANSACTION (COMMIT) or BACKOUT TRANSACTION (ROLLBACK) statement.

Note: Because all cursors are closed when a logical unit of work ends, do not place a BACKOUT TRANSACTION statement within a database loop; place it outside the loop or after the outermost loop of nested loops.

DELETE

The DELETE statement deletes a row from a database table that has been read with a preceding FIND, READ, or SELECT statement. It corresponds to the SQL statement DELETE WHERE CURRENT OF *cursor-name*, which means that only the last row that was read can be deleted.

Example:

```
FIND EMPLOYEES WITH NAME = 'SMITH'
     AND FIRST_NAME = 'ROGER'
DELETE
```

Natural translates the Natural statements above into the following SQL statements and assigns a cursor name (for example, CURSOR1). The SELECT statement and the DELETE statement refer to the same cursor.

```
SELECT FROM EMPLOYEES
 WHERE NAME = 'SMITH' AND FIRST_NAME = 'ROGER'
DELETE FROM EMPLOYEES
 WHERE CURRENT OF CURSOR1
```

Natural translates a DELETE statement into an SQL DELETE statement the way it translates a FIND statement into an SQL SELECT statement. For details, see the FIND statement description **below**.

Note: You cannot delete a row read with a FIND SORTED BY or READ LOGICAL statement. For an explanation, see the FIND and READ statement descriptions below.

END TRANSACTION

Natural translates an END TRANSACTION statement into an SQL COMMIT command. The END TRANSACTION statement indicates the end of a logical transaction, commits all modifications to the database, and releases data locked during the transaction.

Notes:

1. Because all cursors are closed when a logical unit of work ends, do not place an END TRANSACTION statement within a database loop; place it outside the loop or after the outermost loop of nested loops.
2. The END TRANSACTION statement cannot be used to store transaction (ET) data when used with Entire Access.
3. Entire Access does not issue a COMMIT automatically when the Natural program terminates.

FIND

Natural translates a FIND statement into an SQL SELECT statement. The SELECT statement is executed by an OPEN CURSOR command followed by a FETCH command. The FETCH command is executed repeatedly until all records have been read or the program exits the FIND processing loop. A CLOSE CURSOR command ends the SELECT processing.

Example:

Natural statements:

```
FIND EMPLOYEES WITH NAME = 'BLACKMORE'
   AND AGE EQ 20 THRU 40
OBTAIN PERSONNEL_ID NAME AGE
```

Equivalent SQL statement:

```
SELECT PERSONNEL_ID, NAME, AGE
  FROM EMPLOYEES
    WHERE NAME = 'BLACKMORE'
      AND AGE BETWEEN 20 AND 40
```

You can use any table column (field) designated as a descriptor to construct search criteria.

Natural translates the WITH clause of a FIND statement into the WHERE clause of an SQL SELECT statement. Natural evaluates the WHERE clause of the FIND statement after the rows have been selected using the WITH clause. View fields may be used in a WITH clause only if they are designated as descriptors.

Natural translates a FIND NUMBER statement into an SQL SELECT statement containing a COUNT(*) clause. When you want to determine whether a record exists for a specific search condition, the FIND NUMBER statement provides better performance than the IF NO RECORDS FOUND clause.

Note: A row read with a FIND statement containing a SORTED BY clause cannot be updated or deleted. Natural translates the SORTED BY clause of a FIND statement into the ORDER BY clause of an SQL SELECT statement, which produces a read-only result table.

HISTOGRAM

Natural translates the HISTOGRAM statement into an SQL SELECT statement. The HISTOGRAM statement returns the number of rows in a table that have the same value in a specific column. The number of rows is returned in the Natural system variable *NUMBER.

Example:

Natural statements:

```
HISTOGRAM EMPLOYEES FOR AGE
OBTAIN AGE
```

Equivalent SQL statements:

```
SELECT AGE, COUNT(*) FROM EMPLOYEES
  GROUP BY AGE
  ORDER BY AGE
```

READ

Natural translates a READ statement into an SQL SELECT statement. Both READ PHYSICAL and READ LOGICAL statements can be used.

A row read with a READ LOGICAL statement (Example 1) cannot be updated or deleted. Natural translates a READ LOGICAL statement into the ORDER BY clause of an SQL SELECT statement, which produces a read-only result table.

A READ PHYSICAL statement (Example 2) can be updated or deleted. Natural translates it into a SELECT statement without an ORDER BY clause.

Example 1:

Natural statements:

```
READ PERSONNEL BY NAME
OBTAIN NAME FIRSTNAME DATEOFBIRTH
```

Equivalent SQL statement:

```
SELECT NAME, FIRSTNAME, DATEOFBIRTH FROM PERSONNEL
  WHERE NAME >= ' '
    ORDER BY NAME
```

Example 2:

Natural statements:

```
READ PERSONNEL PHYSICAL
OBTAIN NAME
```

Equivalent SQL statement:

```
SELECT NAME FROM PERSONNEL
```

When a `READ` statement contains a `WHERE` clause, Natural evaluates the `WHERE` clause after the rows have been selected according to the search criterion.

STORE

The `STORE` statement adds a row to a database table. It corresponds to the SQL `INSERT` statement.

Example:

Natural statement:

```
STORE RECORD IN EMPLOYEES
  WITH PERSONNEL_ID = '2112'
       NAME         = 'LIFESON'
       FIRST_NAME   = 'ALEX'
```

Equivalent SQL statement:

```
INSERT INTO EMPLOYEES (PERSONNEL_ID, NAME, FIRST_NAME)
  VALUES ('2112', 'LIFESON', 'ALEX')
```

UPDATE

The DML `UPDATE` statement updates a table row that has been read with a preceding `FIND`, `READ`, or `SELECT` statement. Natural translates the DML `UPDATE` statement into the SQL statement `UPDATE WHERE CURRENT OF` *`cursor-name`* (a positioned `UPDATE` statement), which means that only the last row that was read can be updated. In the case of nested loops, the last row in each nested loop can be updated.

UPDATE with FIND/READ

When a DML `UPDATE` statement is used after a Natural `FIND` statement, Natural translates the `FIND` statement into an SQL `SELECT` statement with a `FOR UPDATE OF` clause, and translates the DML `UPDATE` statement into an `UPDATE WHERE CURRENT OF` *`cursor-name`* statement.

Example:

```
FIND EMPLOYEES WITH SALARY < 5000
  ASSIGN SALARY = 6000
  UPDATE
```

Natural translates the Natural statements above into the following SQL statements and assigns a cursor name (for example, `CURSOR1`). The `SELECT` and `UPDATE` statements refer to the same cursor.

```
SELECT SALARY FROM EMPLOYEES WHERE SALARY < 5000
  FOR UPDATE OF SALARY
UPDATE EMPLOYEES SET SALARY = 6000
  WHERE CURRENT OF CURSOR1
```

You cannot update a row read with a `FIND SORTED BY` or `READ LOGICAL` statement. For an explanation, see the `FIND` and `READ` statement descriptions above.

An `END TRANSACTION` or `BACKOUT TRANSACTION` statement releases data locked by an `UPDATE` statement.

UPDATE with SELECT

The DML `UPDATE` statement can be used after a `SELECT` statement only in the following case:

```
SELECT *
  INTO VIEW view-name
```

Natural rejects any other form of the `SELECT` statement used with the DML `UPDATE` statement. Natural translates the DML `UPDATE` statement into a non-cursor or “searched” SQL `UPDATE` statement, which means than only an entire Natural view can be updated; individual columns cannot be updated.

In addition, the DML `UPDATE` statement can be used after a `SELECT` statement only in Natural structured mode, which has the following syntax:

```
UPDATE [RECORD] [IN] [STATEMENT] [(r)]
```

Example:

```
DEFINE DATA LOCAL
01 PERS VIEW OF SQL-PERSONNEL
  02 NAME
  02 AGE
END-DEFINE
SELECT *
  INTO VIEW PERS
  FROM SQL-PERSONNEL
  WHERE NAME LIKE 'S%'
  OBTAIN NAME
    IF NAME = 'SMITH'
      ADD 1 TO AGE
    UPDATE
    END-IF
END-SELECT
```

In other respects, the DML `UPDATE` statement works with the `SELECT` statement the way it works with the Natural `FIND` statement (see ***UPDATE with FIND/READ*** above).

Natural SQL Statements

The SQL statements available within the Natural programming language comprise two different sets of statements: the common set and the extended set. On this platform, only the extended set is supported by Natural.

The common set can be handled by each SQL-eligible database system supported by Natural. It basically corresponds to the standard SQL syntax definitions. For a detailed description of the common set of Natural SQL statements, see *Common Set and Extended Set* (in the *Statements* documentation).

This section describes considerations and restrictions when using the common set of Natural SQL statements with Entire Access.

- DELETE
- INSERT
- PROCESS SQL
- SELECT

- UPDATE

DELETE

The Natural SQL `DELETE` statement deletes rows in a table without using a cursor.

Whereas Natural translates the DML `DELETE` statement into a positioned `DELETE` statement (that is, an SQL `DELETE WHERE CURRENT OF` *`cursor-name`* statement), the Natural SQL `DELETE` statement is a non-cursor or searched `DELETE` statement. A searched `DELETE` statement is a stand-alone statement unrelated to any `SELECT` statement.

INSERT

The `INSERT` statement adds rows to a table; it corresponds to the Natural `STORE` statement.

PROCESS SQL

The `PROCESS SQL` statement issues SQL statements in a *`statement-string`* to the database identified by a *`ddm-name`*.

Note: It is not possible to run database loops using the `PROCESS SQL` statement.

Parameters

Natural supports the `INDICATOR` and `LINDICATOR` clauses. As an alternative, the *`statement-string`* may include parameters. The syntax item *`parameter`* is syntactically defined as follows:

```
[ :U ]
[    ] :host-variable
[ :G ]
```

A *`host-variable`* is a Natural program variable referenced in an SQL statement.

SET SQLOPTION option=value

With Entire Access, you can also specify `SET SQLOPTION` *`option`*`=`*`value`* as *`statement-string`*. This can be used to specify various options for accessing SQL databases. The options apply only to the database referenced by the `PROCESS SQL` statement.

Supported options are:

- `DATEFORMAT`
- `DBPROCESS` (for Sybase only)
- `TIMEOUT` (for Sybase only)
- `TRANSACTION` (for Sybase only)

DATEFORMAT

This option specifies the format used to retrieve SQL Date and Datetime information into Natural fields of type A. The option is obsolete if Natural fields of type D or T are used. A subset of the Natural date and time edit masks can be used:

YYYY	Year (4 digits)
YY	Year (2 digits)
MM	Month
DD	Day
HH	Hour
II	Minute
SS	Second

If the date format contains blanks, it must be enclosed in apostrophes.

Examples:

To use ISO date format, specify

```
PROCESS SQL sql-ddm << SET SQLOPTION DATEFORMAT = YYYY-MM-DD >>
```

To obtain date and time components in ISO format, specify

```
PROCESS SQL sql-ddm << SET SQLOPTION DATEFORMAT = 'YYYY-MM-DD HH:II:SS'    >>
```

The DATEFORMAT is evaluated only if data are retrieved from the database. If data are passed to the database, the conversion is done by the database system. Therefore, the format specified with DATEFORMAT should be a valid date format of the underlying database.

If no DATEFORMAT is specified for Natural fields,

- the default date format DD-MON-YY is used (where MON is a 3-letter abbreviation of the English month name) and
- the following default datetime formats are used:

Adabas D	YYYYMMDDHHIISS
Db2	YYYY-MM-DD-HH.II.SS
INFORMIX	YYYY-MM-DD HH:II:SS
ODBC	YYYY-MM-DD HH:II:SS
ORACLE	YYYYMMDDHHIISS
SYBASE DBLIB	YYYYMMDD HH:II:SS
SYBASE CTLIB	YYYYMMDD HH:II:SS

Microsoft SQL Server	`YYYYMMDD HH:II:SS`
other	`DD-MON-YY`

DBPROCESS

This option is valid for Sybase and Microsoft SQL Server databases only.

This option is used to influence the allocation of SQL statements to Sybase and Microsoft SQL Server `DBPROCESS`es. `DBPROCESS`es are used by Entire Access to emulate database cursors, which are not provided by the Sybase and Microsoft SQL Server DBlib interface.

Two values are possible:

`MULTIPLE`	With `DBPROCESS` set to `MULTIPLE`, each `SELECT` statement uses its own secondary `DBPROCESS`, whereas all other SQL statements are executed within the primary `DBPROCESS`. The value `MULTIPLE` therefore enables your application to execute further SQL statements, even if a database loop is open. It also allows nested database loops.
`SINGLE`	With `DBPROCESS` set to `SINGLE`, all SQL statements use the same (that is, the primary) `DBPROCESS`. It is therefore not possible to execute a new database statement while a database loop is active, because one `DBPROCESS` can only execute one SQL batch at a time. Since all statements are executed in the same (primary) `DBPROCESS`, however, this setting enables `SELECT`ions from non-shared temporary tables.

Notes:

1. The specified value can only be changed if no database loop is active.
2. As the `DBPROCESS` option only applies to the Sybase and Microsoft SQL Server DBlib interface, your application should use a central `CALLNAT` statement to change the value (at least for `SINGLE`), so that you can easily remove these calls once Sybase client libraries are supported. Your application should also use a central error handling that establishes the default setting (`MULTIPLE`).

TIMEOUT

This option is valid for Sybase and Microsoft SQL Server databases only.

With Sybase and Microsoft SQL Server, Entire Access uses a timeout technique to detect database-access deadlocks. The default timeout period is 8 seconds. With this option, you can change the duration of the timeout period (in seconds).

For example, to set the timeout period to 30 seconds, specify

```
PROCESS SQL sql-ddm << SET SQLOPTION TIMEOUT = 30 >>
```

TRANSACTION

This option is valid for Sybase and Microsoft SQL Server databases only.

This option is used to enable or disable transaction mode. It becomes effective after the next `END TRANSACTION` or `BACKOUT TRANSACTION` statement.

If transaction mode is enabled (this is the default), Natural automatically issues all required statements to begin a transaction.

Examples:

To disable transaction mode, specify

```
PROCESS SQL sql-ddm << SET SQLOPTION TRANSACTION = NO >>
...
END TRANSACTION
```

To enable transaction mode, specify

```
PROCESS SQL sql-ddm << SET SQLOPTION TRANSACTION = YES >>
...
END TRANSACTION
```

SQLDISCONNECT

With Entire Access, you can also specify `SQLDISCONNECT` as the *`statement-string`*. In combination with the `SQLCONNECT` statement (see **below**), this statement can be used to access different databases by one application within the same session, by simply connecting and disconnecting as required.

A successfully performed `SQLDISCONNECT` statement clears the information previously provided by the `SQLCONNECT` statement; that is, it disconnects your application from the currently connected SQL database determined by the DBID of the DDM used in the `PROCESS SQL` statement. If no connection is established, the `SQLDISCONNECT` statement is ignored. It will fail if a transaction is open.

Note: If Natural reports an error in the `SQLDISCONNECT` statement, the connection status does not change. If the database reports an error, the connection status is undefined.

SQLCONNECT option=value

With Entire Access, you can also specify `SQLCONNECT` *option=value* as the *statement-string*. This statement can be used to establish a connection to an SQL database according to the DBID specified in the DDM addressed by the `PROCESS SQL` statement. The `SQLCONNECT` statement will fail if the specified connection is already established.

Supported options are:

- USERID
- PASSWORD
- OS_PASSWORD
- OS_USERID
- DBMS_PARAMETER

Notes:

1. If the `SQLCONNECT` statement fails, the connection status does not change.
2. If several options are specified, they must be separated by a comma.
3. The specified value can be either a character literal or a Natural variable of format A.
4. If Natural performs an implicit reconnect, because the connection to the database was lost, the values provided by the `SQLCONNECT` statement are used.

The options are evaluated as described below.

USERID and PASSWORD

Specifying `USERID` and `PASSWORD` for the database logon suppresses the default logon window and the evaluation of the environment variables `SQL_DATABASE_USER` and `SQL_DATABASE_PASSWORD`.

If only `USERID` is specified, `PASSWORD` is assumed to be blank, and vice versa.

If neither `USERID` nor `PASSWORD` is specified, default logon processing applies.

Note: With database systems that do not require user ID and password, a blank user ID and password can be specified to suppress the default logon processing.

OS_USERID and OS_PASSWORD

Specifying OS_PASSWORD and OS_USERID for the operating system logon suppresses the logon window and the evaluation of the environment variables SQL_OS_USER and SQL_OS_PASSWORD.

If only OS_USERID is specified, OS_PASSWORD is assumed to be blank, and vice versa.

If neither OS_USERID nor OS_PASSWORD is specified, default logon processing applies.

Note: With operating systems that do not require user ID and password, a blank user ID and password can be specified to suppress the default logon processing.

DBMS_PARAMETER

Specifying DBMS_PARAMETER dynamically overwrites the DBMS assignment in the Natural global configuration file.

Examples:

```
PROCESS SQL sql-ddm << SQLCONNECT USERID = 'DBA', PASSWORD = 'SECRET'    >>
```

This example connects to the database specified in the Natural global configuration file with user ID DBA and password SECRET.

```
DEFINE DATA LOCAL
1 #UID (A20)
1 #PWD (A20)
END-DEFINE
INPUT 'Please enter ADABAS D user ID and password' / #UID / #PWD
PROCESS SQL sql-ddm << SQLCONNECT USERID = : #UID,
                       PASSWORD        = : #PWD,
                       DBMS_PARAMETER    = 'ADABASD:mydb'
                    >>
```

This example connects to the Adabas D database mydb with the user ID and password taken from the INPUT statement.

```
PROCESS SQL sql-ddm << SQLCONNECT USERID = ' ', PASSWORD = ' ',
                       DBMS_PARAMETER    = 'DB2:EXAMPLE' >>
```

This example connects to the Db2 database EXAMPLE without specifying user ID and password (since these are not required by Db2 which uses the operating system user ID).

SELECT

The `INTO` clause and scalar operators for the `SELECT` statement either are RDBMS-specific and do not conform to the standard SQL syntax definitions (the Natural common set), or impose restrictions when used with Entire Access.

Entire Access does not support the `INDICATOR` and `LINDICATOR` clauses in the `INTO` clause. Thus, Entire Access requires the following syntax for the `INTO` clause:

```
INTO [ parameter, ...        ]
     [ VIEW {view-name},...  ]
```

Note: The concatenation operator (`||`) does not belong to the common set and is therefore not supported by Entire Access.

SELECT SINGLE

The `SELECT SINGLE` statement provides the functionality of a non-cursor `SELECT` operation (singleton `SELECT`); that is, a `SELECT` statement that retrieves a maximum of one row without using a cursor.

This statement is similar to the Natural `FIND UNIQUE` statement. However, Natural automatically checks the number of rows returned. If more than one row is selected, Natural returns an error message.

If your RDBMS does not support dynamic execution of a non-cursor `SELECT` operation, the Natural `SELECT SINGLE` statement is executed like a set-level `SELECT` statement, which results in a cursor operation. However, Natural still checks the number of returned rows and issues an error message if more than one row is selected.

UPDATE

The Natural SQL `UPDATE` statement updates rows in a table without using a cursor.

Whereas Natural translates the DML `UPDATE` statement into a positioned `UPDATE` statement (that is, the SQL `DELETE WHERE CURRENT OF` *`cursor-name`* statement), the Natural SQL `UPDATE` statement is a non-cursor or searched `UPDATE` statement. A searched `UPDATE` statement is a stand-alone statement unrelated to any `SELECT` statement.

Flexible SQL

Flexible SQL allows you to use arbitrary RDBMS-specific SQL syntax extensions. Flexible SQL can be used as a replacement for any of the following syntactical SQL items:

- atom
- column reference
- scalar expression
- condition

The Natural compiler does not recognize the SQL text used in flexible SQL; it simply copies the SQL text (after substituting values for the host variables, which are Natural program variables referenced in an SQL statement) into the SQL string that it passes to the RDBMS. Syntax errors in flexible SQL text are detected at runtime when the RDBMS executes the string.

Note the following characteristics of flexible SQL:

- It is enclosed in << and >> characters and can include arbitrary SQL text and host variables.
- Host variables must be prefixed by a colon (:).
- The SQL string can cover several statement lines; comments are permitted.

Flexible SQL can also be used between the clauses of a select expression:

```
SELECT selection
  << ... >>
  INTO ...
  FROM ...
  << ... >>
  WHERE ...
  << ... >>
  GROUP BY ...
  << ... >>
  HAVING ...
  << ... >>
  ORDER BY ...
  << ... >>
```

Examples:

```
SELECT NAME
FROM EMPLOYEES
WHERE << MONTH (BIRTH) >> = << MONTH (CURRENT_DATE) >>

SELECT NAME
FROM EMPLOYEES
WHERE << MONTH (BIRTH) = MONTH (CURRENT_DATE) >>

SELECT NAME
FROM EMPLOYEES
WHERE SALARY > 50000
<< INTERSECT
   SELECT NAME
   FROM EMPLOYEES
   WHERE DEPT = 'DEPT10'
>>
```

RDBMS-Specific Requirements and Restrictions

This section discusses restrictions and special requirements for Natural and some RDBMSs used with Entire Access.

The following topics are covered:

- Case-Sensitive Database Systems
- SYBASE and Microsoft SQL Server

Case-Sensitive Database Systems

In case-sensitive database systems, use lower-case characters for table and column names, as all names specified in a Natural program are automatically converted to lower-case.

Note: This restriction does not apply when you use flexible SQL.

SYBASE and Microsoft SQL Server

To execute SQL statements against SYBASE and Microsoft SQL Server, you must use one or more `DBPROCESS` structures. A `DBPROCESS` can execute SQL command batches.

A command batch is a sequence of SQL statements. Statements must be executed in the sequence in which they are defined in the command batch. If a statement (for example, a `SELECT` statement) returns a result, you must execute the statement first and then fetch the rows one by one. Once you execute the next statement from the command batch, you can no longer fetch rows from the previous query.

With SYBASE and Microsoft SQL Server, an application can use more than one `DBPROCESS` structure; therefore, it is possible to have nested queries if you use a separate `DBPROCESS` for each query. Because SYBASE and Microsoft SQL Server lock data for each `DBPROCESS`, however, an application that uses more than one `DBPROCESS` can deadlock itself. Natural times out in case of a deadlock.

The following topics are covered below:

How Natural Statements are Converted to Database Calls

Natural uses one `DBPROCESS` for each open query and another `DBPROCESS` for all other SQL statements (`UPDATE`, `DELETE`, `INSERT`, ...).

If a query is referenced by a positioned `UPDATE` or `DELETE` statement, Natural automatically appends the `FOR BROWSE` clause to the generated `SELECT` statement to allow `UPDATE`s while rows are being read.

For a positioned `UPDATE` or `DELETE` statement, the SYBASE `dbqual` function is used to generate the following search condition:

```
WHERE unique-index = value AND tsequal (timestamp,old-timestamp)
```

This search condition can be used to reselect the current row from the query. The `tsequal` function checks whether the row has been updated by another user.

Natural Restrictions with SYBASE and Microsoft SQL Server

The following restrictions apply when using Natural with SYBASE and Microsoft SQL Server.

Case-Sensitivity

SYBASE and Microsoft SQL Server are case-sensitive, and Natural passes parameters in lowercase. Thus, if your SYBASE and Microsoft SQL Server tables or fields are defined in uppercase or mixed case, you must use database `SYNONYM`s or Natural flexible SQL.

Positioned UPDATE and DELETE Statements

To support positioned `UPDATE` and `DELETE` statements, the table to be accessed must have a unique index and a timestamp column. In addition, the timestamp column must not be included in the select list of the query.

Querying Rows

SYBASE and Microsoft SQL Server lock pages, and locked pages are owned by `DBPROCESS` structures.

Pages locked by an active `DBPROCESS` cannot subsequently be read (by the same or another `DBPROCESS`) until the lock is released by an `END TRANSACTION` or `BACKOUT TRANSACTION` statement.

Therefore, if you have updated, inserted, or deleted a row in a table:

- Do not start a new `SELECT` (`FIND`, `READ`, ...) loop against the same table.
- Do not fetch additional rows from a query that references the same table if the `SELECT` statement has no `FOR BROWSE` clause.

Natural automatically appends the `FOR BROWSE` clause if the query is referenced by a positioned `UPDATE` or `DELETE` statement.

Transaction/Non-Transaction Mode

SYBASE and Microsoft SQL Server differentiate between transaction and non-transaction mode. In transaction mode, Natural connects to the database allowing `INSERT`s, `UPDATE`s and `DELETE`s to be issued; thus, commands that run in non-transaction mode, for example, `CREATE TABLE`, cannot be issued.

Stored Procedures

It is possible to use stored procedures in SYBASE and Microsoft SQL Server using the `PROCESS SQL` statement. However, the stored procedures must not contain

- commands that work only in non-transaction mode; or
- return values.

Data-Type Conversion

When a Natural program accesses data in a relational database, Entire Access converts RDBMS-specific data types to Natural data formats, and vice versa. The RDBMS data types and their corresponding Natural data formats are described in the *Editors* documentation under *Data Conversion for RDBMS* (in the section *DDM Services*.

The date/time or datetime format specific to a particular database can be converted into the Natural formats D and T; see below.

Date/Time Conversion

The RDBMS-specific date/time or datetime format can be converted into the Natural formats D and T.

To use this conversion, you first have to edit the Natural DDM to change the date or time field formats from A(lphanumeric) to D(ate) or T(ime). The `SQLOPTION DATEFORMAT` is obsolete for fields with format D or T.

Note: Date or time fields converted to Natural D(ate)/T(ime) format may not be mixed with those converted to Natural A(lphanumeric) format.

- For update commands, Natural converts the Natural Date and Time format to the database-dependent representation of `DATE`/`TIME`/`DATETIME` to a precision level of seconds.
- For retrieval commands, Natural converts the returned database-dependent character representation to the internal Natural Date or Time format; see conversion tables below. The date component of Natural Time is not ignored and is initialized to `0000-01-02` (`YYYY-MM-DD`) if the RDBMS`s time format does not contain a date component.
- For Natural Date variables, the time portion is ignored and initialized to zero.
- For Natural Time variables, tenth of seconds are ignored and initialized to zero.

Conversion Tables

Adabas D

RDBMS Formats	Natural Date	Natural Time
DATE	YYYYMMDD	
TIME		00HHIISS

Db2

RDBMS Formats	Natural Date	Natural Time
DATE	YYYY-MM-DD	
TIME		HH.II.SS

INFORMIX

RDBMS Formats	Natural Date	Natural Time
DATETIME, year to day	YYYY-MM-DD	
DATETIME, year to second (other formats are not supported)		YYYY-MM-DD-HH:II:SS*

ODBC

RDBMS Formats	Natural Date	Natural Time
DATE	YYYY-MM-DD	
TIME		HH:II:SS

ORACLE

RDBMS Formats	Natural Date	Natural Time
DATE (ORACLE session parameter NLS_DATE_FORMAT is set to YYYYMMDDHH24MISS)	YYYYMMDD000000 (ORACLE time component is set to null for update commands and ignored for retrieval commands.)	YYYYMMDDHHIISS *

SYBASE

RDBMS Formats	Natural Date	Natural Time
DATETIME	YYYYMMDD	YYYYMMDD HH:II:SS *

* When comparing two time values, remember that the date components may have different values.

Microsoft SQL Server

RDBMS Formats	Natural Date	Natural Time
DATETIME	YYYYMMDD	YYYYMMDD HH:II:SS*

Obtaining Diagnostic Information about Database Errors

If the database returns an error while being accessed, you can call the non-Natural program CMOSQERR to obtain diagnostic information about the error, using the following syntax:

```
CALL 'CMOSQERR' parm1 parm2
```

The parameters are:

Parameter	Format/Length	Description
parm1	I4	The number of the error returned by the database.
parm2	A70	The text of the error returned by the database.

SQL Authorization

The Natural Configuration Utility allows you to add DBID specific settings of user IDs and passwords for automatic login to SQL databases. It distinguishes between operating system authentication and database authentication, depending on the current database system. If the **Auto login** flag in the **SQL Authorization** table is set for an SQL DBID then no interactive login prompt will pop up. The login values will be taken from this table row.

Please refer to *SQL Assignments* in the *Configuration Utility* documentation for a more detailed description of the SQL Authorization table.

29 Accessing Data in a Tamino Database

This chapter describes the different aspects of accessing a Tamino database with the Natural data manipulation language (DML).

For information about how to configure Natural to work with Tamino, see *Natural for Tamino* in the *Database Management System Interfaces* documentation.

Prerequisite

Tamino stores structured data-oriented XML documents in containers called doctypes. The doctypes are grouped logically together in so-called collections. Collections are stored in a Tamino database, which is the physical container of data.

The kind of data that can be stored in Tamino and that is to be accessed by Natural for Tamino must be defined in a Tamino XML Schema.

DDM and View Definitions with Natural for Tamino

This section describes the basic concepts of the Tamino XML schema language, Natural DDMs and view definitions and how they interact with Natural for Tamino.

The following topics are covered:

Introducing Tamino XML Schema Language

The Tamino XML schema language is used to define a data type-oriented description of the structure of XML documents. In Tamino, a doctype represents a container for XML documents with the same root element and the same structure within a collection.

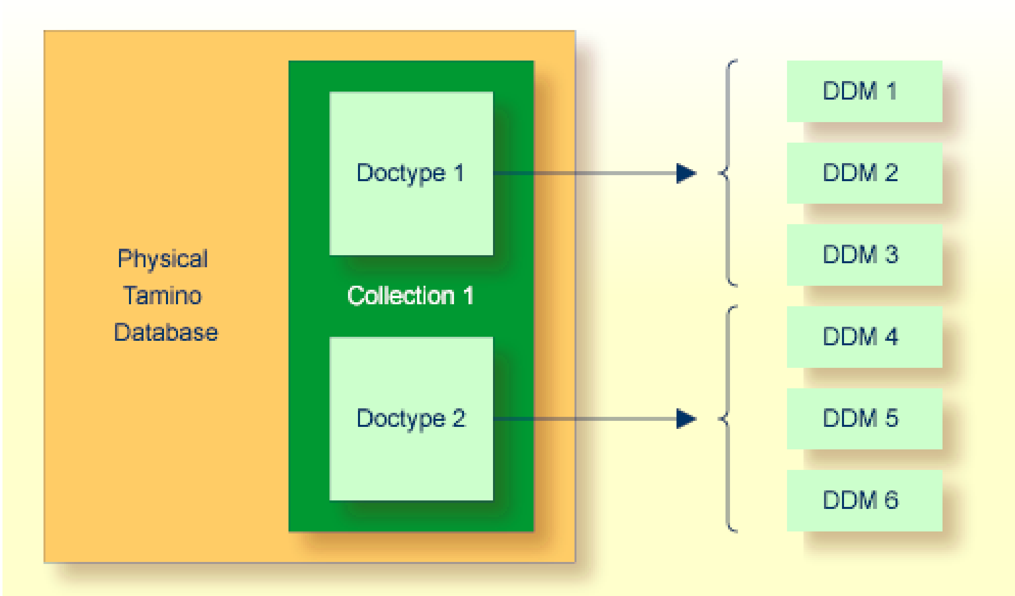
In Tamino, a collection is a container for a set of varying doctypes, so that a collection can be seen as the logical grouping of doctypes that belong together.

In a Tamino XML schema definition, a doctype is defined together with the collection in which it is contained. One Tamino XML schema can define more than one doctype and it can also define doctypes for more than one collection.

For more information on the Tamino XML schema language, refer to the Tamino documentation.

DDMs from Tamino

For Natural to be able to access a Tamino database, a logical connection between a Tamino doctype and the Natural data structures must be provided. Such a logical connection is called a DDM (data definition module).



A Natural DDM generated from a Tamino database is a representation of one doctype defined in one schema. The DDM contains information about the type of each data field and all the necessary structural information as defined in the corresponding Tamino XML schema. To generate a new DDM, the doctype must be selected from a list of all doctypes available in a given collection. Since one collection is bound to one Natural database ID (DBID), it is necessary to use a second DBID if a doctype from another collection is to be accessed.

A Tamino XML schema describes data and data structures in a very different way than with Natural data definitions. Therefore, specific mappings are introduced to derive a Natural data format from a Tamino XML schema data type.

You define DDMs with Natural DDM Services. For more information about Tamino XML schema mapping, refer to *Data Conversion for Tamino* in the *DDM Services* section of the *Editors* documentation.

For the field attributes defined in a DDM, refer to the DDM editor*DDM Editor*, *Using the DDM EditorDDM Services*, *Using the DDM Editor* section in the *Editors* documentation.

Arrays in DDMs from Tamino

If you define an XML element with a `maxOccurs` value greater than one in the Tamino XML Schema, then this element can occur as often as this value indicates. Such a construction is mapped either on a Natural static array definition or on a Natural X-Array definition. Depending on the type of the XML element you are dealing with, the following situations may occur:

- If the XML element is a `complexType` with `complexContent` (i.e. it is an element containing other elements) then the generated corresponding Natural group will be an indexed group.
- If the XML element is a `simpleType` (i.e. the element is holding data only) or a `complexType` with `simpleContent` (i.e. the element has only data and attributes but no other elements) then the generated Natural data field will be an array.

For further information about mapping `maxOccurs` definitions onto Natural arrays, see *Data Conversion for Tamino* in the *DDM Services* section of the *Editors* documentation. The array boundaries or the kind of the array (static array or X-Array) can be adapted in a corresponding view definition as usual.

Example of a DDM

This is an example of an `EMPLOYEES` DDM generated from a Tamino XML Schema definition.

The schema can, for example, be defined with the Natural demo application `SYSEXDB`:

```
DB: 00250 FILE: 00001  - EMPLOYEES-XML
TYPE: XML
COLLECTION: NATDemoData
SCHEMA: Employee
DOCTYPE: Employee
NAMESPACE-PREFIX: xs
NAMESPACE-URI: http://www.w3.org/2001/XMLSchema
T L  Name                             F Leng       D Remark
- -- -------------------------------- - ---------- - -----------
G  1 EMPLOYEE
     FLAGS=MULT_REQUIRED,MULT_ONCE
     TAG=Employee
     XPATH=/Employee
G  2 GROUP$1
     FLAGS=GROUP_ATTRIBUTES
   3 PERSONNEL-ID                     A          8 D xs:string
     FLAGS=ATTR_REQUIRED
     TAG=@Personnel-ID
     XPATH=/Employee/@Personnel-ID
G  2 GROUP$2
     FLAGS=GROUP_SEQUENCE,MULT_REQUIRED,MULT_ONCE
G  3 FULL-NAME
     FLAGS=MULT_OPTIONAL
     TAG=Full-Name
     XPATH=/Employee/Full-Name
```

```
G  4 GROUP$3
     FLAGS=GROUP_SEQUENCE,MULT_REQUIRED,MULT_ONCE
   5 FIRST-NAME                        A         20 D xs:string
     FLAGS=MULT_OPTIONAL
     TAG=First-Name
     XPATH=/Employee/Full-Name/First-Name
   5 MIDDLE-NAME                       A         20 D xs:string
     FLAGS=MULT_OPTIONAL
     TAG=Middle-Name
     XPATH=/Employee/Full-Name/Middle-Name
   5 MIDDLE-I                          A         20 D xs:string
     FLAGS=MULT_OPTIONAL
     TAG=Middle-I
     XPATH=/Employee/Full-Name/Middle-I
   5 NAME                              A         20 D xs:string
     FLAGS=MULT_OPTIONAL
     TAG=Name
     XPATH=/Employee/Full-Name/Name
       . . .
   3 LANG                              A          3   xs:string
     FLAGS=ARRAY,MULT_OPTIONAL
     OCC=1:4
     TAG=Lang
     XPATH=/Employee/Lang   ↩
```

Definition of Views

In order to work with Tamino database fields in a Natural program, you must specify the required fields of the DDM in a Natural `view-definition` (see the `DEFINE DATA` statement). Normally, a view is a special subset of the complete data structure as defined in the DDM.

Tamino XML Schema->Natural for Tamino DDM->Natural `view-definition`

If the view is used to store XML objects, it has to contain all fields that are required to a generate documents that are valid according to the corresponding Tamino XML schema definition.

A view for the `EMPLOYEES-XML` DDM, where one of the view fields is a static array, might look like this:

```
DEFINE DATA LOCAL
01 VW VIEW OF EMPLOYEES-XML
02 NAME
02 CITY
02 LANG (1:4)
END-DEFINE
```

Natural Statements for Tamino Database Access

The Natural DML statements which are provided for Tamino access can be subdivided into two categories:

- pure retrieval statements;
- database modification statements.

The Natural system variable `*ISN` is mapped on the Tamino `ino:id`.

Natural for Tamino Retrieval Statements

The following Natural statements can be used for database retrieval:

- `FIND`

  This statement is used to select those records from a database which meet a specified search criterion.
- `GET`

  This statement is used to select one special record with its unique id from the database.
- `READ`

  This statement is used to select a range of records from a database in a specified sequence.

Not all of the possible options and all of the possible clauses of the retrieval statements can be used for Tamino access. Please read the appropriate section in the *Statements* documentation for a detailed description.

All statements are internally realized with the Tamino `_xquery` command verb. Statement clauses are mapped to corresponding Tamino XQuery expressions, e.g. search criteria are mapped to Tamino XQuery comparison expressions, sequence specifications are mapped to Tamino XQuery ordering expressions with sort direction.

The result set for the `FIND` and `READ` statements is determined at start of the loop and remains unchanged throughout the loop.

The following is an example of reading a set of employee records from a Tamino database where one view field is an array:

```
* READ 5 RECORDS DESCENDING CONTAINING A
* STATIC ARRAY IN THE VIEW DEFINE DATA LOCAL
01 VW VIEW OF EMPLOYEES-TAMINO
02 NAME
02 CITY
02 LANG (1:4)
END-DEFINE
*
READ(5) VW DESCENDING BY NAME = 'MAYER'
  DISPLAY NAME CITY LANG(*)
END-READ
*
END
```

Natural for Tamino Database Modification Statements

The following database modification statements are provided for use with Natural for Tamino:

- `STORE`

  This statement is used for inserting a new XML document into the database.
- `DELETE`

  This statement is used for deleting a document from the database. The `DELETE` statement implements a positioned delete.

For a detailed description of the statements, see the appropriate sections of the *Statements* documentation.

The `DELETE` statement is internally realized with the Tamino `_delete` command verb using the current `ino:id`, and the `STORE` statement is implemented with the `_process` command verb.

Example:

The following example program stores a new employee record with some data in the database:

```
* STORE NEW EMPLOYEE
DEFINE DATA LOCAL
01 VW VIEW OF EMPLOYEES-TAMINO
02 PERSONNEL-ID
02 NAME
02 CITY
02 LANG (1:3)
END-DEFINE
*
* FILL VIEW
PERSONNEL-ID := '1230815'
NAME         := 'KENT'
CITY         := 'ROME'
LANG(1)      := 'ENG'
```

```
LANG(2)      := 'GER'
LANG(3)      := 'SPA'
*
* STORE VIEW
STORE RECORD IN VW
*
COMMIT
*
END
```

If the Tamino XML Schema defines data structures for a doctype as being mandatory, then these data structures must also be filled in the view before a `STORE` statement is issued, otherwise this will result in a Tamino error.

Natural for Tamino Logical Transaction Handling

Natural performs database modification operations based on transactions, which means that all database modification requests are processed in logical transaction units. A logical transaction is the smallest unit of work (as defined by you) which must be performed in its entirety to ensure that the information contained in the database is logically consistent.

A logical transaction may consist of one or more modification statements (`DELETE`, `STORE`) involving one or more doctypes in the database. A logical transaction may also span multiple Natural programs.

A logical transaction begins when a database modification statement is issued. Natural does this automatically. For example, if a `FIND` loop contains a `DELETE` statement. The end of a logical transaction is determined by an `END TRANSACTION` statement in the program. This statement ensures that all modifications within the transaction have been successfully applied.

Natural for Tamino Error Handling

In addition to Natural's standard error messages there are two special error codes which provide additional information via a sub-error code.

Error Message NAT8400

```
NAT8400 Tamino error ... occurred
```

For this special error an additional sub-code number is shown. This number refers to a Tamino error message. Please see the Tamino *Messages and Codes* documentation. The user exit USR6007 in library SYSEXT is provided for obtaining diagnostic information in case a NAT8400 error occurs.

Here is an example of usage:

```
DEFINE DATA LOCAL
 01 VW VIEW OF EMPLOYEES-TAMINO
  02 NAME
  02 CITY
 01 TAMINO_PARMS
  02 TAMINO_ERROR_NUM        (I4)   /* Error number of Tamino error
  02 TAMINO_ERROR_TEXT       (A70)  /* Tamino error text
  02 TAMINO_ERROR_LINE       (A253) /* Tamino error message line
END-DEFINE
*
NAME := 'MEYER'
CITY := 'BOSTON'
STORE VW
*
ON ERROR
 IF *ERROR EQ 8400     /* in case of error 8400 obtain diagnostic information
  CALLNAT 'USR6007N' TAMINO_PARMS
  PRINT 'Error 8400 occurred:'
  PRINT 'Error Number:' TAMINO_ERROR_NUM
  PRINT 'Error Text  :' TAMINO_ERROR_TEXT
  PRINT 'Error Line  :' TAMINO_ERROR_LINE
 END-IF
END-ERROR
*
END
```

Error Message NAT8411

```
NAT8411 HTTP request failed with response code...
```

The error code from the HTTP server is delivered as additional information. See also the `REQUEST DOCUMENT` statement, *HTTP Responses Redirected and Denied*.

Example of Natural for Tamino Interacting with a SQL Database

This is a more sophisticated example of Natural for Tamino interacting with an SQL database; it retrieves data from a Tamino database and inserts or updates the corresponding row in an appropriate table in a SQL database.

```
*
* TAMINO DB --> SQL RDBMS EXAMPLE
*
DEFINE DATA LOCAL
* DEFINE VIEW FOR TAMINO
01 VW-TAMINO VIEW OF EMPLOYEES-TAMINO
02 PERSONNEL-ID
02 NAME
02 CITY
* DEFINE VIEW FOR SQL DATABASE
01 VW-SQL VIEW OF EMPLOYEES-SQL
```

```
02 PERSONNEL_ID
02 NAME
02 CITY
END-DEFINE
*
* OPEN A TAMINO LOGICAL READ LOOP
*
TAMINO. READ VW-TAMINO BY NAME
*
* SEARCH RECORD IN SQL DATABASE AND
* INSERT A NEW RECORD IF NOT FOUND OR
* UPDATE THE EXISTING ONE WITH THE DATA
* FROM TAMINO DB
SQL. FIND(1) VW-SQL WITH PERSONNEL_ID = PERSONNEL-ID (TAMINO.)
       IF NO RECORDS FOUND
         PERSONNEL_ID := PERSONNEL-ID (TAMINO.)
         NAME         := NAME         (TAMINO.)
         CITY         := CITY         (TAMINO.)
         STORE VW-SQL
         ESCAPE BOTTOM
       END-NOREC
       PERSONNEL_ID := PERSONNEL-ID (TAMINO.)
       NAME         := NAME         (TAMINO.)
       CITY         := CITY         (TAMINO.)
       UPDATE
     END-FIND
*
END-READ
*
END TRANSACTION
*
END
```

Natural for Tamino Restrictions

There are restrictions concerning the scope of the Tamino XML Schema language that can be used for creating schemas for Natural for Tamino DDM generation:

- Only Tamino XML Schema language constructors and attributes (as mentioned in *Tamino XML Schema Constructors* in the *DDM Services* section of the *Editors* documentation) are supported by Natural for Tamino. Other constructors such as `xs:any`, `xs:anyAttribute` cannot be applied in Tamino XML Schemas if you wish to use them together with Natural for Tamino.
- The functionality of `xs:import` is not supported by Natural for Tamino. This means that external schema components must not be referenced in a Tamino XML Schema suitable for usage together with Natural. In other words, a doctype definition in a Tamino XML Schema must resolve all references within this Tamino XML Schema itself if you are planning to use it together with Natural for Tamino.

- The attribute `mixed` of the constructor `xs:complexType` is only supported with its default value `false`. Natural for Tamino does not support mixed-content document definitions (as set with the specification `mixed="true"`). Using `mixed="true"` will result in an error during DDM generation.
- The level of nested structures in a Natural for Tamino DDM is limited to 99. A new DDM level is generated whenever one of the following constructors occurs in the Tamino XML Schema:

  ```
  xs:element
  xs:attribute
  xs:choice
  xs:all
  xs:sequence
  ```

- Recursively defined structures in a Tamino XML Schema cannot be used together with Natural for Tamino.
- The Tamino XML Schema language constructor `xs:choice` is mapped on a Natural group containing all alternatives of the choice. To restrict processing to one particular choice, an appropriate view with the required choice has to be created.
- Natural for Tamino only supports “closed content validation mode”. Tamino XML Schemas with “open content validation mode” cannot be used together with Natural for Tamino.
- For the Tamino XML Schema language constructors `xs:choice`, `xs:sequence` and `xs:all`, a value greater than 1 of the attribute `maxOccurs` cannot be handled in the Natural data structures. Hence a value greater than 1 will always lead to an error during DDM generation.
- Natural for Tamino can handle only Tamino objects that are defined with a Tamino XML Schema as a subset of the W3C schema. Especially Natural for Tamino does not support non-XML (`tsd:nonXML`) data or instances without a defined schema (`ino:etc`).

VI Report Format and Control

This part describes how to proceed if a Natural program is to produce multiple reports. Furthermore, it discusses various aspects of how you can control the format of an output report created with Natural, that is, the way in which the data are displayed.

30 Report Specification - (rep) Notation

`(rep)` is the output report identifier for which a statement is applicable.

Use of Report Specifications

If a Natural program is to produce multiple reports, the notation `(rep)` must be specified with each output statement (see *Statements Concerned*, below) which is to be used to create output for any report other than the first report (Report 0).

A value of `0 - 31` may be specified.

The value for `(rep)` may also be a logical name which has been assigned using the `DEFINE PRINTER` statement, see *Example 2* below.

Statements Concerned

The notation `(rep)` can be used with the following output statements:

`AT END OF PAGE` | `AT TOP OF PAGE` | `DISPLAY` | `EJECT` | `FORMAT` | `NEWPAGE` | `PRINT` | `SKIP` | `SUSPEND IDENTICAL SUPPRESS` | `WRITE` | `WRITE TITLE` | `WRITE TRAILER`

Examples of Report Specification

Example 1 - Multiple Reports

```
DISPLAY (1) NAME ...
WRITE (4) NAME ...
```

Example 2 - Using Logical Names

```
DEFINE PRINTER (LIST=5) OUTPUT 'LPT1'
WRITE (LIST) NAME ...
```

31 Layout of an Output Page

This chapter gives an overview of the statements that may be used to define a specific layout for a report.

Statements Influencing a Report Layout

The following statements have an impact on the layout of the report:

Statement	Function
`WRITE TITLE`	With this statement, you can specify a page title, that is, text to be output at the top of a page. By default, page titles are centered and not underlined.
`WRITE TRAILER`	With this statement, you can specify a page trailer, that is, text to be output at the bottom of a page. By default, the trailer lines are centered and not underlined.
`AT TOP OF PAGE`	With this statement, you can specify any processing that is to be performed whenever a new page of the report is started. Any output from this processing will be output below the page title.
`AT END OF PAGE`	With this statement, you can specify any processing that is to be performed whenever an end-of-page condition occurs. Any output from this processing will be output below any page trailer (as specified with the `WRITE TRAILER` statement).
`AT START OF DATA`	With this statement, you specify processing that is to be performed after the first record has been read in a database processing loop. Any output from this processing will be output before the first field value. See note below.
`AT END OF DATA`	With this statement, you specify processing that is to be performed after all records for a processing loop have been processed. Any output from this processing will be output immediately after the last field value. See note below.
`DISPLAY` / `WRITE`	With these statements, you control the format in which the field values that have been read are to be output. See section ***Statements DISPLAY and WRITE***.

Note: The relevance of the statements `AT START OF DATA` and `AT END OF DATA` for the output of data is described under *Database Access*, ***AT START/END OF DATA Statements***. The other statements listed above are discussed in other sections of the part ***Report Format and Control***.

General Layout Example

The following example program illustrates the general layout of an output page:

```
** Example 'OUTPUX01': Several sections of output
************************************************************************
DEFINE DATA LOCAL
1 EMP-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BIRTH
END-DEFINE
*
WRITE TITLE   '********** Page Title **********'
WRITE TRAILER '********** Page Trailer **********'
*
AT TOP OF PAGE
  WRITE '===== Top of Page ====='
END-TOPPAGE
AT END OF PAGE
  WRITE '===== End of Page ====='
END-ENDPAGE
*
READ (10) EMP-VIEW BY NAME
  /*
  DISPLAY NAME FIRST-NAME BIRTH (EM=YYYY-MM-DD)
  /*
  AT START OF DATA
    WRITE '>>>>> Start of Data >>>>>'
  END-START
  AT END OF DATA
    WRITE '<<<<< End of Data <<<<<'
  END-ENDDATA
END-READ
END
```

Output of Program `OUTPUX01`:

```
                    ********** Page Title **********
===== Top of Page =====
        NAME              FIRST-NAME         DATE
                                              OF
                                             BIRTH
-------------------- -------------------- ----------

>>>>> Start of Data >>>>>
ABELLAN              KEPA                 1961-04-08
ACHIESON             ROBERT               1963-12-24
ADAM                 SIMONE               1952-01-30
ADKINSON             JEFF                 1951-06-15
ADKINSON             PHYLLIS              1956-09-17
ADKINSON             HAZEL                1954-03-19
ADKINSON             DAVID                1946-10-12
ADKINSON             CHARLIE              1950-03-02
ADKINSON             MARTHA               1970-01-01
ADKINSON             TIMMIE               1970-03-03
```

```
<<<<< End of Data <<<<<
                  ********** Page Trailer **********
===== End of Page =====
```

32 Statements DISPLAY and WRITE

This chapter describes how to use the statements DISPLAY and WRITE to output data and control the format in which information is output.

DISPLAY Statement

The DISPLAY statement produces output in column format; that is, the values for one field are output in a column underneath one another. If multiple fields are output, that is, if multiple columns are produced, these columns are output next to one another horizontally.

The order in which fields are displayed is determined by the sequence in which you specify the field names in the DISPLAY statement.

The DISPLAY statement in the following program displays for each person first the personnel number, then the name and then the job title:

```
** Example 'DISPLX01': DISPLAY
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
*
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID NAME JOB-TITLE
END-READ
END
```

Output of Program DISPLX01:

```
Page      1                                                04-11-11  14:15:54

PERSONNEL         NAME                  CURRENT
   ID                                  POSITION
--------- -------------------- -------------------------


30020013  GARRET               TYPIST
30016112  TAILOR               WAREHOUSEMAN
20017600  PIETSCH              SECRETARY
```

To change the order of the columns that appear in the output report, simply reorder the field names in the DISPLAY statement. For example, if you prefer to list employee names first, then job titles followed by personnel numbers, the appropriate DISPLAY statement would be:

```
** Example 'DISPLX02': DISPLAY
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
*
READ (3) VIEWEMP BY BIRTH
  DISPLAY NAME JOB-TITLE PERSONNEL-ID
END-READ
END
```

Output of Program DISPLX02:

```
Page      1                                                04-11-11  14:15:54

        NAME                  CURRENT          PERSONNEL
                             POSITION             ID
-------------------- ------------------------- ---------


GARRET               TYPIST                    30020013
TAILOR               WAREHOUSEMAN              30016112
PIETSCH              SECRETARY                 20017600
```

A header is output above each column. Various ways to influence this header are described in the document ***Column Headers***.

WRITE Statement

The WRITE statement is used to produce output in free format (that is, not in columns). In contrast to the DISPLAY statement, the following applies to the WRITE statement:

- If necessary, it automatically creates a line advance; that is, a field or text element that does not fit onto the current output line, is automatically output in the next line.
- It does not produce any headers.
- The values of a multiple-value field are output next to one another horizontally, and not underneath one another.

The two example programs shown below illustrate the basic differences between the DISPLAY statement and the WRITE statement.

You can also use the two statements in combination with one another, as described later in the document *Vertical Displays*, ***Combining DISPLAY and WRITE*** .

Example of DISPLAY Statement

```
** Example 'DISPLX03': DISPLAY
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:3)
END-DEFINE
*
READ (2) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME FIRST-NAME SALARY (1:3)
END-READ
END
```

Output of Program DISPLX03:

```
Page      1                                                  04-11-11  14:15:54

        NAME                FIRST-NAME         ANNUAL
                                               SALARY
-------------------- -------------------- ----------

JONES                VIRGINIA                  46000
                                               42300
                                               39300
JONES                MARSHA                    50000
                                               46000
                                               42700
```

Example of WRITE Statement

```
** Example 'WRITEX01': WRITE
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:3)
END-DEFINE
*
READ (2) VIEWEMP BY NAME STARTING FROM 'JONES'
  WRITE NAME FIRST-NAME SALARY (1:3)
END-READ
END
```

Output of Program WRITEX01:

```
Page      1                                                   04-11-11  14:15:55

JONES                VIRGINIA                  46000      42300      39300
JONES                MARSHA                    50000      46000      42700
```

Column Spacing - SF Parameter and nX Notation

By default, the columns output with a DISPLAY statement are separated from one another by *one* space.

With the session parameter SF, you can specify the default number of spaces to be inserted between columns output with a DISPLAY statement. You can set the number of spaces to any value from 1 to 30.

The parameter can be specified with a FORMAT statement to apply to the whole report, or with a DISPLAY statement at statement level, but not at element level.

With the *n*X notation in the DISPLAY statement, you can specify the number of spaces (*n*) to be inserted between two columns. An *n*X notation overrides the specification made with the SF parameter.

```
** Example 'DISPLX04': DISPLAY (with nX)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
*
FORMAT SF=3
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID NAME 5X JOB-TITLE
END-READ
END
```

Output of Program DISPLX04:

The above example program produces the following output, where the first two columns are separated by 3 spaces due to the SF parameter in the FORMAT statement, while the second and third columns are separated by 5 spaces due to the notation 5X in the DISPLAY statement:

```
Page      1                                                04-11-11  14:15:54

PERSONNEL          NAME                        CURRENT
   ID                                          POSITION
---------   --------------------     -------------------------

30020013    GARRET                   TYPIST
30016112    TAILOR                   WAREHOUSEMAN
20017600    PIETSCH                  SECRETARY
```

The *n*X notation is also available with the `WRITE` statement to insert spaces between individual output elements:

```
WRITE PERSONNEL-ID 5X NAME 3X JOB-TITLE
```

With the above statement, 5 spaces will be inserted between the fields `PERSONNEL-ID` and `NAME`, and 3 spaces between `NAME` and `JOB-TITLE`.

Tab Setting - nT Notation

With the *n*T notation, which is available with the `DISPLAY` and the `WRITE` statement, you can specify the position where an output element is to be output.

```
** Example 'DISPLX05': DISPLAY (with nT)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY 5T NAME 30T FIRST-NAME
END-READ
END
```

Output of Program `DISPLX05`:

The above program produces the following output, where the field `NAME` is output starting in the 5th position (counted from the left margin of the page), and the field `FIRST-NAME` starting in the 30th position:

```
Page      1                                                     04-11-11  14:15:54

            NAME                 FIRST-NAME
    --------------------     --------------------

    JONES                    VIRGINIA
    JONES                    MARSHA
    JONES                    ROBERT
```

Line Advance - Slash Notation

With a slash (/) in a `DISPLAY` or `WRITE` statement, you cause a line advance.

- In a `DISPLAY` statement, a slash causes a line advance *between fields* and *within text*.
- In a `WRITE` statement, a slash causes a line advance only when placed *between fields*; within text, it is treated like an ordinary text character.

When placed between fields, the slash must have a blank on either side.

For multiple line advances, you specify multiple slashes.

Example 1 - Line Advance in DISPLAY Statement:

```
** Example 'DISPLX06': DISPLAY (with slash '/')
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 DEPARTMENT
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME / FIRST-NAME 'DEPART-/MENT' DEPARTMENT
END-READ
END
```

Output of Program `DISPLX06`:

The above `DISPLAY` statement produces a line advance after each value of the field `NAME` and within the text `DEPART-MENT`:

```
Page      1                                                   04-11-11  14:15:54

        NAME         DEPART-
     FIRST-NAME       MENT
-------------------- -------

JONES                SALE
VIRGINIA
JONES                MGMT
MARSHA
JONES                TECH
ROBERT
```

Example 2 - Line Advance in WRITE Statement:

```
** Example 'WRITEX02': WRITE (with line advance)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 DEPARTMENT
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  WRITE NAME / FIRST-NAME 'DEPART-/MENT' DEPARTMENT //
END-READ
END
```

Output of Program `WRITEX02`:

The above `WRITE` statement produces a line advance after each value of the field `NAME`, and a double line advance after each value of the field `DEPARTMENT`, but none within the text `DEPART-/MENT`:

```
Page      1                                                   04-11-11  14:15:55

JONES
VIRGINIA             DEPART-/MENT SALE


JONES
MARSHA               DEPART-/MENT MGMT


JONES
ROBERT               DEPART-/MENT TECH
```

Example 3 - Line Advance in DISPLAY and WRITE Statements:

```
** Example 'DISPLX21': DISPLAY (usage of slash '/' in DISPLAY and WRITE)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
  2 ADDRESS-LINE (1)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED
       *TIME
   5X  'PEOPLE LIVING IN SALT LAKE CITY'
   21X 'PAGE:' *PAGE-NUMBER /
   15X 'AS OF' *DAT4E //
*
WRITE TRAILER UNDERLINED 'REGISTER OF' / 'SALT LAKE CITY'
*
READ (2) EMPLOY-VIEW WITH CITY = 'SALT LAKE CITY'
  DISPLAY  NAME /
           FIRST-NAME
           'HOME/CITY' CITY
           'STREET/OR BOX NO.' ADDRESS-LINE (1)
  SKIP 1
END-READ
END
```

Output of Program DISPLX21:

```
14:15:54.6     PEOPLE LIVING IN SALT LAKE CITY                     PAGE:      1
               AS OF 11/11/2004



-------------------------------------------------------------------------------
        NAME                 HOME                STREET
     FIRST-NAME              CITY              OR BOX NO.
-------------------- -------------------- --------------------

ANDERSON             SALT LAKE CITY       3701 S. GEORGE MASON
JENNY

SAMUELSON            SALT LAKE CITY       7610 W. 86TH STREET
MARTIN

                                  REGISTER OF
                                SALT LAKE CITY
-------------------------------------------------------------------------------
```

Further Examples of DISPLAY and WRITE Statements

See the following example programs:

- *DISPLX13 - DISPLAY (compare with WRITEX08 using WRITE)*
- *WRITEX08 - WRITE (compare with DISPLX13 using DISPLAY)*
- *DISPLX14 - DISPLAY (with AL, SF and nX)*
- *WRITEX09 - WRITE (in combination with AT END OF DATA)*

33 Index Notation for Multiple-Value Fields and Periodic Groups

This chapter describes how you can use the index notation (*n*:*n*) to specify how many values of a multiple-value field or how many occurrences of a periodic group are to be output.

Use of Index Notation

With the index notation (*n*:*n*) you can specify how many values of a multiple-value field or how many occurrences of a periodic group are to be output.

For example, the field INCOME in the DDM EMPLOYEES is a periodic group which keeps a record of the annual incomes of an employee for each year he/she has been with the company.

These annual incomes are maintained in chronological order. The income of the most recent year is in occurrence 1.

If you wanted to have the annual incomes of an employee for the last three years displayed - that is, occurrences 1 to 3 - you would specify the notation (1:3) after the field name in a DISPLAY or WRITE statement (as shown in the following example program).

Example of Index Notation in DISPLAY Statement

```
** Example 'DISPLX07': DISPLAY (with index notation)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 INCOME  (1:3)
    3 CURR-CODE
    3 SALARY
    3 BONUS (1:1)
END-DEFINE
*
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID NAME INCOME (1:3)
  SKIP 1
END-READ
END
```

Output of Program DISPLX07:

Note that a DISPLAY statement outputs multiple values of a multiple-value field underneath one another:

```
Page      1                                                 04-11-11  14:15:54

PERSONNEL         NAME                     INCOME
   ID
                               CURRENCY   ANNUAL     BONUS
                                 CODE     SALARY
--------- -------------------- -------- ---------- ----------

30020013  GARRET               UKL            4200          0
                               UKL            4150          0
                                                 0          0

30016112  TAILOR               UKL            7450          0
                               UKL            7350          0
                               UKL            6700          0

20017600  PIETSCH              USD           22000          0
                               USD           20200          0
                               USD           18700          0
```

As a `WRITE` statement displays multiple values horizontally instead of vertically, this may cause a line overflow and a - possibly undesired - line advance.

If you use only a single field within a periodic group (for example, `SALARY`) instead of the entire periodic group, and if you also insert a slash (/) to cause a line advance (as shown in the following example between `NAME` and `JOB-TITLE`), the report format becomes manageable.

Example of Index Notation in WRITE Statement

```
** Example 'WRITEX03': WRITE (with index notation)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
  2 SALARY (1:3)
END-DEFINE
*
READ (3) VIEWEMP BY BIRTH
  WRITE PERSONNEL-ID NAME / JOB-TITLE SALARY (1:3)
  SKIP 1
END-READ
END
```

Output of Program `WRITEX03`:

```
Page      1                                                04-11-11  14:15:55

30020013 GARRET
TYPIST                          4200       4150          0

30016112 TAILOR
WAREHOUSEMAN                    7450       7350       6700

20017600 PIETSCH
SECRETARY                      22000      20200      18700
```

34 Page Titles, Page Breaks, Blank Lines

This chapter describes various ways of controlling page breaks in a report, the output of page titles at the top of each report page and the generation of empty lines in an output report.

Default Page Title

For each page output via a `DISPLAY` or `WRITE` statement, Natural automatically generates a single default title line. This title line contains the page number, the date and the time of day.

Example:

```
WRITE 'HELLO'
END
```

The above program produces the following output with default page title:

```
Page      1                                                   04-12-14  13:19:33

HELLO                                                                          ↩
```

Suppress Page Title - NOTITLE Option

If you wish your report to be output without page titles, you add the keyword `NOTITLE` to the statement `DISPLAY` or `WRITE`.

Example - DISPLAY with NOTITLE:

```
** Example 'DISPLX20': DISPLAY (with NOTITLE)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
END-DEFINE
*
READ (5) EMPLOY-VIEW BY CITY FROM 'BOSTON'
  DISPLAY NOTITLE NAME FIRST-NAME CITY
END-READ
END
```

Output of Program `DISPLX20`:

```
        NAME                 FIRST-NAME              CITY
-------------------- -------------------- --------------------


SHAW                 LESLIE               BOSTON
STANWOOD             VERNON               BOSTON
CREMER               WALT                 BOSTON
PERREAULT            BRENDA               BOSTON
COHEN                JOHN                 BOSTON
```

Example - WRITE with NOTITLE:

```
WRITE NOTITLE 'HELLO'
END
```

The above program produces the following output without page title:

```
HELLO                                                                                        ↩
```

Define Your Own Page Title - WRITE TITLE Statement

If you wish a page title of your own to be output instead of the Natural default page title, you use the statement `WRITE TITLE`.

The following topics are covered below:

- Specifying Text for Your Title
- Specifying Empty Lines after the Title
- Title Justification and/or Underlining
- Title with Page Number

Specifying Text for Your Title

With the statement `WRITE TITLE`, you specify the text for your title (in apostrophes).

```
WRITE TITLE 'THIS IS MY PAGE TITLE'
WRITE 'HELLO'
END
```

The above program produces the following output:

```
                              THIS IS MY PAGE TITLE
HELLO                                                                         ↩
```

Specifying Empty Lines after the Title

With the `SKIP` option of the `WRITE TITLE` statement, you can specify the number of empty lines to be output immediately below the title line. After the keyword `SKIP`, you specify the number of empty lines to be inserted.

```
WRITE TITLE 'THIS IS MY PAGE TITLE' SKIP 2
WRITE 'HELLO'
END
```

The above program produces the following output:

```
                              THIS IS MY PAGE TITLE


HELLO                                                                         ↩
```

`SKIP` is not only available as part of the `WRITE TITLE` statement, but also as a stand-alone statement.

Title Justification and/or Underlining

By default, the page title is centered on the page and not underlined.

The `WRITE TITLE` statement provides the following options which can be used independent of each other:

Option	Effect
`LEFT JUSTIFIED`	Causes the page trailer to be displayed left-justified.
`UNDERLINED`	Causes the title to be displayed underlined. The underlining runs the width of the line size (see also Natural profile and session parameter `LS`). By default, titles are underlined with a hyphen (-). However, with the `UC` session parameter you can specify another character to be used as underlining character (see ***Underlining Character for Titles and Headers***).

The following example shows the effect of the `LEFT JUSTIFIED` and `UNDERLINED` options:

```
WRITE TITLE LEFT JUSTIFIED UNDERLINED 'THIS IS MY PAGE TITLE'
SKIP 2
WRITE 'HELLO'
END
```

The above program produces the following output:

```
THIS IS MY PAGE TITLE
-------------------------------------------------------------------------------


HELLO                                                                         ↩
```

The `WRITE TITLE` statement is executed whenever a new page is initiated for the report.

Title with Page Number

In the following examples, the system variable `*PAGE-NUMBER` is used in conjunction with the `WRITE TITLE` statement to output the page number in the title line.

```
** Example 'WTITLX01': WRITE TITLE (with *PAGE-NUMBER)
************************************************************************
DEFINE DATA LOCAL
1 VEHIC-VIEW VIEW OF VEHICLES
  2 MAKE
  2 YEAR
  2 MAINT-COST (1)
END-DEFINE
*
LIMIT 5
*
READ VEHIC-VIEW
END-ALL
SORT BY YEAR USING MAKE MAINT-COST (1)
  DISPLAY NOTITLE YEAR MAKE MAINT-COST (1)
  AT BREAK OF YEAR
    MOVE 1 TO *PAGE-NUMBER
    NEWPAGE
  END-BREAK
  /*
  WRITE TITLE LEFT JUSTIFIED
        'YEAR:' YEAR 15X 'PAGE' *PAGE-NUMBER
END-SORT
END
```

Output of Program `WTITLX01`:

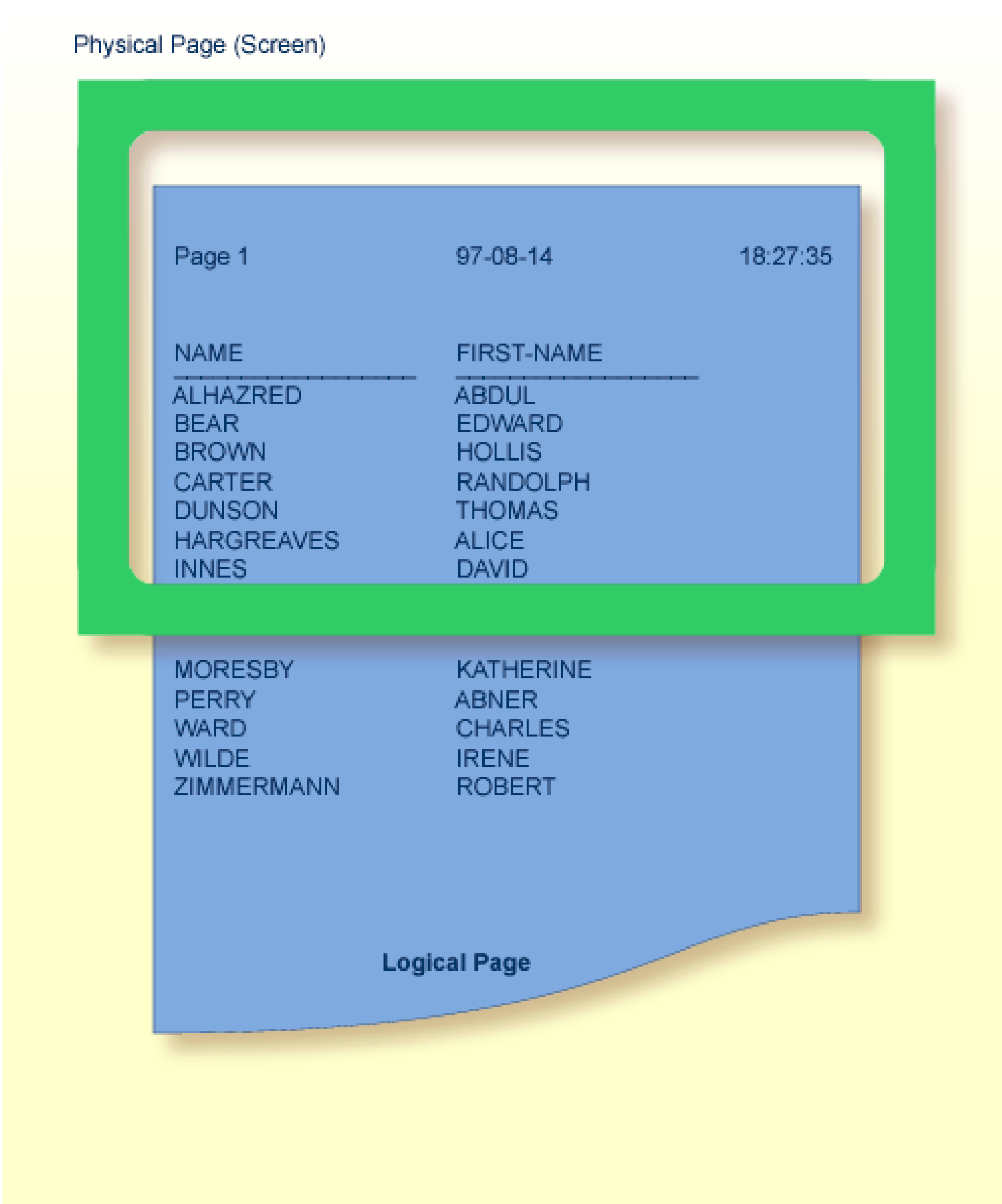
```
YEAR:  1980               PAGE      1
YEAR         MAKE         MAINT-COST
----- -------------------- ----------

 1980 RENAULT                 20000
 1980 RENAULT                 20000
 1980 PEUGEOT                 20000
```

Logical Page and Physical Page

A *logical page* is the output produced by a Natural program. A *physical page* is your terminal screen on which the output is displayed; or it may be the piece of paper on which the output is printed.

The size of the logical page is determined by the number of lines output by the Natural program.

If more lines are output than fit onto one screen, the logical page will exceed the physical screen, and the remaining lines will be displayed on the next screen.



Note: If information you wish to appear at the bottom of the screen (for example, output created by a `WRITE TRAILER` or `AT END OF PAGE` statement) is output on the next screen instead, reduce the logical page size accordingly (with the session parameter `PS`, which is discussed below).

Page Size - PS Parameter

With the parameter `PS` (Page Size for Natural Reports), you determine the maximum number of lines per (logical) page for a report.

When the number of lines specified with the `PS` parameter is reached, a page advance occurs (unless page advance is controlled with a `NEWPAGE` or `EJECT` statement; see ***Page Advance Controlled by EJ Parameter*** below).

The `PS` parameter can be set either at session level with the system command `GLOBALS`, or within a program with the following statements:

At report level:

- `FORMAT PS=`*nn*

At statement level:

- `DISPLAY (PS=`*nn*`)`
- `WRITE (PS=`*nn*`)`
- `WRITE TITLE (PS=`*nn*`)`
- `WRITE TRAILER (PS=`*nn*`)`
- `INPUT (PS=`*nn*`)`

Page Advance

A page advance can be triggered by one of the following methods:

These methods are discussed below.

Page Advance Controlled by EJ Parameter

With the session parameter `EJ` (Page Eject), you determine whether page ejects are to be performed or not. By default, `EJ=ON` applies, which means that page ejects will be performed as specified.

If you specify `EJ=OFF`, page break information will be ignored. This may be useful to save paper during test runs where page ejects are not needed.

The `EJ` parameter can be set at session level with the system command `GLOBALS`; for example:

```
GLOBALS EJ=OFF
```

The `EJ` parameter setting is overridden by the `EJECT` statement.

Page Advance Controlled by EJECT or NEWPAGE Statements

The following topics are covered below:

- Page Advance without Title/Header on Next Page
- Page Advance with End/Top-of-Page Processing

Page Advance without Title/Header on Next Page

The `EJECT` statement causes a page advance *without* a title or header line being generated on the next page. A new physical page is started *without* any top-of-page or end-of-page processing being performed (for example, no `WRITE TRAILER` or `AT END OF PAGE`, `WRITE TITLE`, `AT TOP OF PAGE` or `*PAGE-NUMBER` processing).

The `EJECT` statement overrides the `EJ` parameter setting.

Page Advance with End/Top-of-Page Processing

The `NEWPAGE` statement causes a page advance *with* associated end-of-page and top-of-page processing. A trailer line will be displayed, if specified. A title line, either default or user-specified, will be displayed on the new page, unless the `NOTITLE` option has been specified in a `DISPLAY` or `WRITE` statement (as described **above**).

If the `NEWPAGE` statement is not used, page advance is automatically controlled by the setting of the `PS` parameter; see ***Page Size - PS Parameter*** above).

Eject/New Page when less than n Lines Left

Both the NEWPAGE statement and the EJECT statement provide a WHEN LESS THAN *n* LINES LEFT option. With this option, you specify a number of lines (*n*). The NEWPAGE/EJECT statement will then be executed if - at the time the statement is processed - less than *n* lines are available on the current page.

Example 1:

```
FORMAT PS=55
...
NEWPAGE WHEN LESS THAN 7 LINES LEFT
...
```

In this example, the page size is set to 55 lines.

If only 6 or less lines are left on the current page at the time when the NEWPAGE statement is processed, the NEWPAGE statement is executed and a page advance occurs. If 7 or more lines are left, the NEWPAGE statement is not executed and no page advance occurs; the page advance then occurs depending on the session parameter PS (Page Size for Natural Reports), that is, after 55 lines.

Example 2:

```
** Example 'NEWPAX02': NEWPAGE (in combination with EJECT and
**                     parameter PS)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
END-DEFINE
*
FORMAT PS=15
*
READ (9) EMPLOY-VIEW BY CITY STARTING FROM 'BOSTON'
  AT START OF DATA
    EJECT
    WRITE /// 20T '%' (29) /
              20T '%%'                          47T '%%' /
              20T '%%' 3X 'REPORT OF EMPLOYEES' 47T '%%' /
              20T '%%' 3X '  SORTED BY CITY   ' 47T '%%' /
              20T '%%'                          47T '%%' /
              20T '%' (29) /
    NEWPAGE
  END-START
  AT BREAK OF CITY
    NEWPAGE WHEN LESS 3 LINES LEFT
  END-BREAK
  DISPLAY CITY (IS=ON) NAME JOB-TITLE
```

```
END-READ
END
```

New Page with Title

The NEWPAGE statement also provides a WITH TITLE option. If this option is not used, a default title will appear at the top of the new page or a WRITE TITLE statement or NOTITLE clause will be executed.

The WITH TITLE option of the NEWPAGE statement allows you to override these with a title of your own choice. The syntax of the WITH TITLE option is the same as for the WRITE TITLE statement.

Example:

```
NEWPAGE WITH TITLE LEFT JUSTIFIED 'PEOPLE LIVING IN BOSTON:'
```

The following program illustrates the use of the session parameter PS (Page Size for Natural Reports) and the NEWPAGE statement. Moreover, the system variable *PAGE-NUMBER is used to display the current page number.

```
** Example 'NEWPAX01': NEWPAGE
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 DEPT
END-DEFINE
*
FORMAT PS=20
READ (5) VIEWEMP BY CITY STARTING FROM 'M'
  DISPLAY NAME 'DEPT' DEPT 'LOCATION' CITY
  AT BREAK OF CITY
    NEWPAGE WITH TITLE LEFT JUSTIFIED
           'EMPLOYEES BY CITY - PAGE:' *PAGE-NUMBER
  END-BREAK
END-READ
END
```

Output of Program NEWPAX01:

Note the position of the page breaks and the title line:

```
Page      1                                                    04-11-11  14:15:54

        NAME          DEPT        LOCATION
-------------------- ------ --------------------


FICKEN               TECH10 MADISON
KELLOGG              TECH10 MADISON
ALEXANDER            SALE20 MADISON
```

Page 2:

```
EMPLOYEES BY CITY - PAGE:      2
        NAME          DEPT        LOCATION
-------------------- ------ --------------------


DE JUAN              SALE03 MADRID
DE LA MADRID         PROD01 MADRID
```

Page 3:

```
EMPLOYEES BY CITY - PAGE: 3
```

Page Trailer - WRITE TRAILER Statement

The following topics are covered below:

- Specifying a Page Trailer
- Considering Logical Page Size
- Page Trailer Justification and/or Underlining

Specifying a Page Trailer

The `WRITE TRAILER` statement is used to output text (in apostrophes) at the bottom of a page.

```
WRITE TRAILER 'THIS IS THE END OF THE PAGE'
```

The statement is executed when an end-of-page condition is detected, or as a result of a `SKIP` or `NEWPAGE` statement.

Considering Logical Page Size

As the end-of-page condition is checked only *after* an entire DISPLAY or WRITE statement has been processed, it may occur that the logical page size (that is, the number of lines output by a DISPLAY or WRITE statement) causes the physical size of the output page to be exceeded before the WRITE TRAILER statement is executed.

To ensure that a page trailer actually appears at the bottom of a physical page, you should set the logical page size (with the PS session parameter) to a value less than the physical page size.

Page Trailer Justification and/or Underlining

By default, the page trailer is displayed centered on the page and not underlined.

The WRITE TRAILER statement provides the following options which can be used independent of each other:

Option	Effect
LEFT JUSTIFIED	Causes the page trailer to be displayed left justified.
UNDERLINED	The underlining runs the width of the line size (see also Natural profile and session parameter LS). By default, titles are underlined with a hyphen (-). However, with the UC session parameter you can specify another character to be used as underlining character (see *Underlining Character for Titles and Headers*).

The following examples show the use of the LEFT JUSTIFIED and UNDERLINED options of the WRITE TRAILER statement:

Example 1:

```
WRITE TRAILER LEFT JUSTIFIED UNDERLINED 'THIS IS THE END OF THE PAGE'
```

Example 2:

```
** Example 'WTITLX02': WRITE TITLE AND WRITE TRAILER
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
  2 ADDRESS-LINE (1)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED
       *TIME
   5X  'PEOPLE LIVING IN SALT LAKE CITY'
   21X 'PAGE:' *PAGE-NUMBER /
```

```
   15X 'AS OF' *DAT4E //
*
WRITE TRAILER UNDERLINED 'REGISTER OF' / 'SALT LAKE CITY'
*
READ (2) EMPLOY-VIEW WITH CITY = 'SALT LAKE CITY'
  DISPLAY  NAME /
           FIRST-NAME
           'HOME/CITY' CITY
           'STREET/OR BOX NO.' ADDRESS-LINE (1)
  SKIP 1
END-READ
END
```

Generating Blank Lines - SKIP Statement

The `SKIP` statement is used to generate one or more blank lines in an output report.

Example 1 - SKIP in conjunction with WRITE and DISPLAY:

```
** Example 'SKIPX01': SKIP (in conjunction with WRITE and DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
  2 ADDRESS-LINE (1)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED
     'PEOPLE LIVING IN SALT LAKE CITY AS OF' *DAT4E 7X
     'PAGE:' *PAGE-NUMBER
SKIP 3
*
READ (2) EMPLOY-VIEW WITH CITY = 'SALT LAKE CITY'
  DISPLAY NAME / FIRST-NAME CITY ADDRESS-LINE (1)
  SKIP 1
END-READ
END
```

Example 2 - SKIP in conjunction with DISPLAY VERT:

```
** Example 'SKIPX02': SKIP (in conjunction with DISPLAY VERT)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 JOB-TITLE
END-DEFINE
*
READ (2) EMPLOY-VIEW WITH JOB-TITLE = 'SECRETARY'
  DISPLAY NOTITLE VERT
          NAME FIRST-NAME / CITY
  SKIP 3
END-READ
*
NEWPAGE
*
READ (2) EMPLOY-VIEW WITH JOB-TITLE = 'SECRETARY'
  DISPLAY NOTITLE
          NAME FIRST-NAME / CITY
  SKIP 3
END-READ
END
```

AT TOP OF PAGE Statement

The AT TOP OF PAGE statement is used to specify any processing that is to be performed whenever a new page of the report is started.

If the AT TOP OF PAGE processing produces any output, this will be output below the page title (with a skipped line in between).

By default, this output is displayed left-justified on the page.

Example:

```
** Example 'ATTOPX01': AT TOP OF PAGE
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 MAR-STAT
  2 BIRTH
  2 CITY
```

```
  2 JOB-TITLE
  2 DEPT
END-DEFINE
*
LIMIT 10
READ EMPLOY-VIEW BY PERSONNEL-ID FROM '20017000'
  DISPLAY NOTITLE (AL=10)
          NAME DEPT JOB-TITLE CITY 5X
          MAR-STAT 'DATE OF/BIRTH' BIRTH (EM=YY-MM-DD)
  /*
  AT TOP OF PAGE
    WRITE /   '-BUSINESS INFORMATION-'
          26X '-PRIVATE INFORMATION-'
  END-TOPPAGE
END-READ
END
```

Output of Program ATTOPX01:

```
-BUSINESS INFORMATION-                          -PRIVATE INFORMATION-
   NAME    DEPARTMENT  CURRENT      CITY         MARITAL   DATE OF
              CODE     POSITION                   STATUS    BIRTH
---------- ---------- ---------- ----------     ---------- --------

CREMER     TECH10     ANALYST    GREENVILLE     S          70-01-01
MARKUSH    SALE00     TRAINEE    LOS ANGELE     D          79-03-14
GEE        TECH05     MANAGER    CHAPEL HIL     M          41-02-04
KUNEY      TECH10     DBA        DETROIT        S          40-02-13
NEEDHAM    TECH10     PROGRAMMER CHATTANOOG     S          55-08-05
JACKSON    TECH10     PROGRAMMER ST LOUIS       D          70-01-01
PIETSCH    MGMT10     SECRETARY  VISTA          M          40-01-09
PAUL       MGMT10     SECRETARY  NORFOLK        S          43-07-07
HERZOG     TECH05     MANAGER    CHATTANOOG     S          52-09-16
DEKKER     TECH10     DBA        MOBILE         W          40-03-03
```

AT END OF PAGE Statement

The AT END OF PAGE statement is used to specify any processing that is to be performed whenever an end-of-page condition occurs.

If the AT END OF PAGE processing produces any output, this will be output after any **page trailer** (as specified with the WRITE TRAILER statement).

By default, this output is displayed left-justified on the page.

The same considerations **described above** for page trailers regarding physical and logical page sizes and the number of lines output by a `DISPLAY` or `WRITE` statement also apply to `AT END OF PAGE` output.

Example:

```
** Example 'ATENPX01': AT END OF PAGE (with system function available
**                     via GIVE SYSTEM FUNCTIONS in DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 SALARY (1)
END-DEFINE
*
READ (10) EMPLOY-VIEW BY PERSONNEL-ID = '20017000'
  DISPLAY NOTITLE GIVE SYSTEM FUNCTIONS
          NAME JOB-TITLE 'SALARY' SALARY(1)
  /*
  AT END OF PAGE
    WRITE / 24T 'AVERAGE SALARY: ...' AVER(SALARY(1))
  END-ENDPAGE
END-READ
END
```

Output of Program `ATENPX01`:

```
        NAME                      CURRENT            SALARY
                                  POSITION
-------------------- ------------------------- ----------

CREMER               ANALYST                        34000
MARKUSH              TRAINEE                        22000
GEE                  MANAGER                        39500
KUNEY                DBA                            40200
NEEDHAM              PROGRAMMER                     32500
JACKSON              PROGRAMMER                     33000
PIETSCH              SECRETARY                      22000
PAUL                 SECRETARY                      23000
HERZOG               MANAGER                        48500
DEKKER               DBA                            48000


AVERAGE SALARY: ...      34270
            ↩
```

Further Example

See the following example program:

- *DISPLX21 - DISPLAY (with slash '/' and compare with WRITE)*

35 Column Headers

This chapter describes various ways of controlling the display of column headers produced by a DISPLAY statement.

Default Column Headers

By default, each database field output with a DISPLAY statement is displayed with a default column header (which is defined for the field in the DDM).

```
** Example 'DISPLX01': DISPLAY
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
*
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID NAME JOB-TITLE
END-READ
END
```

Output of Program DISPLX01:

The above example program uses default headers and produces the following output.

```
Page      1                                                04-11-11  14:15:54

PERSONNEL         NAME                 CURRENT
   ID                                 POSITION
--------- -------------------- -------------------------

30020013  GARRET               TYPIST
30016112  TAILOR               WAREHOUSEMAN
20017600  PIETSCH              SECRETARY
```

Suppress Default Column Headers - NOHDR Option

If you wish your report to be output without column headers, add the keyword `NOHDR` to the `DISPLAY` statement.

```
DISPLAY NOHDR PERSONNEL-ID NAME JOB-TITLE
```

Define Your Own Column Headers

If you wish column headers of your own to be output instead of the default headers, you specify '*text*' (in apostrophes) immediately before a field, *text* being the header to be used for the field.

```
** Example 'DISPLX08': DISPLAY  (with column title in 'text')
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
*
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID
         'EMPLOYEE' NAME
         'POSITION' JOB-TITLE
END-READ
END
```

Output of Program `DISPLX08`:

The above program contains the header `EMPLOYEE` for the field `NAME`, and the header `POSITION` for the field `JOB-TITLE`; for the field `PERSONNEL-ID`, the default header is used. The program produces the following output:

```
Page      1                                                  04-11-11  14:15:54

PERSONNEL       EMPLOYEE               POSITION
   ID
--------- -------------------- -------------------------


30020013  GARRET               TYPIST
30016112  TAILOR               WAREHOUSEMAN
20017600  PIETSCH              SECRETARY
```

Combining NOTITLE and NOHDR

To create a report that has neither page title nor column headers, you specify the NOTITLE and NOHDR options together in the following order:

```
DISPLAY NOTITLE NOHDR PERSONNEL-ID NAME JOB-TITLE
```

Centering of Column Headers - HC Parameter

By default, column headers are centered above the columns. With the HC parameter, you can influence the placement of column headers.

If you specify

HC=L	headers will be left-justified.
HC=R	headers will be right-justified.
HC=C	headers will be centered.

The HC parameter can be used in a FORMAT statement to apply to the whole report, or it can be used in a DISPLAY statement at both statement level and element level, for example:

```
DISPLAY (HC=L) PERSONNEL-ID NAME JOB-TITLE
```

Width of Column Headers - HW Parameter

With the HW parameter, you determine the width of a column output with a DISPLAY statement.

If you specify

HW=ON	the width of a DISPLAY column is determined by either the length of the header text or the length of the field, whichever is longer. This also applies by default.
HW=OFF	the width of a DISPLAY column is determined only by the length of the field. However, HW=OFF only applies to DISPLAY statements which do *not* create headers; that is, either a first DISPLAY statement with NOHDR option or a subsequent DISPLAY statement.

The HW parameter can be used in a FORMAT statement to apply to the entire report, or it can be used in a DISPLAY statement at both statement level and element (field) level.

Filler Characters for Headers - Parameters FC and GC

With the `FC` parameter, you specify the *filler character* which will appear on either side of a *header* produced by a `DISPLAY` statement across the full column width if the column width is determined by the field length and not by the header (see `HW` parameter **above**); otherwise `FC` will be ignored.

When a group of fields or a periodic group is output via a `DISPLAY` statement, a *group header* is displayed across all field columns that belong to that group above the headers for the individual fields within the group. With the `GC` parameter, you can specify the *filler character* which will appear on either side of such a group header.

While the `FC` parameter applies to the headers of individual fields, the `GC` parameter applies to the headers for groups of fields.

The parameters `FC` and `GC` can be specified in a `FORMAT` statement to apply to the whole report, or they can be specified in a `DISPLAY` statement at both statement level and element (field) level.

```
** Example 'FORMAX01': FORMAT (with parameters FC, GC)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 INCOME (1:1)
    3 CURR-CODE
    3 SALARY
    3 BONUS (1:1)
END-DEFINE
*
FORMAT FC=* GC=$
*
READ (3) VIEWEMP BY NAME
  DISPLAY NAME (FC==) INCOME (1)
END-READ
END
```

Output of Program `FORMAX01`:

```
Page      1                                                  04-11-11  14:15:54

========NAME======== $$$$$$$$$$$$INCOME$$$$$$$$$$$$

                     CURRENCY **ANNUAL** **BONUS***
                         CODE     SALARY
-------------------- -------- ---------- ----------

ABELLAN              PTA         1450000          0
ACHIESON             UKL           10500          0
ADAM                 FRA          159980      23000
```

Underlining Character for Titles and Headers - UC Parameter

By default, titles and headers are underlined with a hyphen (-).

With the UC parameter, you can specify another character to be used as underlining character.

The UC parameter can be specified in a FORMAT statement to apply to the whole report, or it can be specified in a DISPLAY statement at both statement level and element (field) level.

```
** Example 'FORMAX02': FORMAT (with parameter UC)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
*
FORMAT UC==
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED 'EMPLOYEES REPORT'
SKIP 1
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID (UC=*) NAME JOB-TITLE
END-READ
END
```

In the above program, the UC parameter is specified at program level and at element (field) level: the underlining character specified with the FORMAT statement (=) applies for the whole report - except for the field PERSONNEL-ID, for which a different underlining character (*) is specified.

Output of Program FORMAX02:

```
EMPLOYEES REPORT
===============================================================================

PERSONNEL         NAME                 CURRENT
   ID                                  POSITION
********* ==================== =========================

30020013  GARRET               TYPIST
30016112  TAILOR               WAREHOUSEMAN
20017600  PIETSCH              SECRETARY
```

Suppressing Column Headers - Slash Notation

With the notation apostrophe-slash-apostrophe ('/'), you can suppress default column headers for individual fields displayed with a `DISPLAY` statement. While the `NOHDR` option suppresses the headers of all columns, the notation `'/'` can be used to suppress the header for an individual column.

The apostrophe-slash-apostrophe ('/') notation is specified in the `DISPLAY` statement immediately before the name of the field for which the column header is to be suppressed.

Compare the following two examples:

Example 1:

```
DISPLAY NAME PERSONNEL-ID JOB-TITLE
```

In this case, the default column headers of all three fields will be displayed:

```
Page      1                                                  04-11-11  14:15:54

        NAME         PERSONNEL          CURRENT
                          ID             POSITION
-------------------- --------- -------------------------

ABELLAN              60008339  MAQUINISTA
ACHIESON             30000231  DATA BASE ADMINISTRATOR
ADAM                 50005800  CHEF DE SERVICE
ADKINSON             20008800  PROGRAMMER
ADKINSON             20009800  DBA
ADKINSON             20011000  SALES PERSON          ↩
```

Example 2:

```
DISPLAY '/' NAME PERSONNEL-ID JOB-TITLE
```

In this case, the notation `'/'` causes the column header for the field `NAME` to be suppressed:

```
Page      1                                                  04-11-11  14:15:54

                     PERSONNEL          CURRENT
                        ID            POSITION
                     --------- -------------------------

ABELLAN              60008339  MAQUINISTA
ACHIESON             30000231  DATA BASE ADMINISTRATOR
ADAM                 50005800  CHEF DE SERVICE
ADKINSON             20008800  PROGRAMMER
```

```
ADKINSON                20009800  DBA
ADKINSON                20011000  SALES PERSON      ↩
```

Further Examples of Column Headers

See the following example programs:

- *DISPLX15 - DISPLAY (with FC, UC)*
- *DISPLX16 - DISPLAY (with '/', 'text', 'text/text')*

36 Parameters to Influence the Output of Fields

This chapter discusses the use of those Natural profile and/or session parameters which you can use to control the output format of fields.

Overview of Field-Output-Relevant Parameters

Natural provides several profile and/or session parameters you can use to control the format in which fields are output:

Parameter	Function
LC, IC and TC	With these session parameters, you can specify characters that are to be displayed before or after a field or before a field value.
LCU, ICU and TCU	With these session parameters, you can specify characters in Unicode format that are to be displayed before or after a field or before a field value.
AL and NL	With these session parameters, you can increase or reduce the output length of fields.
DL	With this session parameter, you can specify the default output length for an alphanumeric map field of format U.
SG	With this session parameter, you can determine whether negative values are to be displayed with or without a minus sign.
IS	With this session parameter, you can suppress the display of subsequent identical field values.
ZP	With this profile and session parameter, you can determine whether field values of 0 are to be displayed or not.
ES	With this session parameter, you can suppress the display of empty lines generated by a DISPLAY or WRITE statement.

These parameters are discussed in the following sections.

Leading Characters - LC Parameter

With the session parameter LC, you can specify leading characters that are to be displayed immediately *before a field* that is output with a DISPLAY statement. The width of the output column is enlarged accordingly. You can specify 1 to 10 characters.

By default, values are displayed left-justified in alphanumeric fields and right-justified in numeric fields. (These defaults can be changed with the AD parameter; see the *Parameter Reference*). When a leading character is specified for an alphanumeric field, the character is therefore displayed immediately before the field value; for a numeric field, a number of spaces may occur between the leading character and the field value.

The LC parameter can be used with the following statements:

- FORMAT
- DISPLAY

The LC parameter can be set at statement level and at element level.

Unicode Leading Characters - LCU Parameter

The session parameter LCU is identical to the session parameter LC. The difference is that the leading characters are always stored in Unicode format.

This allows you to specify leading characters with mixed characters from different code pages, and assures that always the correct character is displayed independent of the installed system code page.

For further information, see *Unicode and Code Page Support in the Natural Programming Language*, *Session Parameters*, section *EMU, ICU, LCU, TCU versus EM, IC, LC, TC*.

The parameters LCU and ICU cannot both be applied to one field.

Insertion Characters - IC Parameter

With the session parameter IC, you specify the characters to be inserted in the column immediately *preceding the value of a field* that is output with a DISPLAY statement. You can specify 1 to 10 characters.

For a numeric field, the insertion characters will be placed immediately before the first significant digit that is output, with no intervening spaces between the specified character and the field value. For alphanumeric fields, the effect of the IC parameter is the same as that of the LC parameter.

The parameters LC and IC cannot both be applied to one field.

The IC parameter can be used with the following statements:

- FORMAT
- DISPLAY

The IC parameter can be set at statement level and at element level.

Unicode Insertion Characters - ICU Parameter

The session parameter `ICU` is identical to the session parameter `IC`. The difference is that the insertion characters are always stored in Unicode format.

This allows you to specify insertion characters with mixed characters from different code pages, and assures that always the correct character is displayed independent of the installed system code page.

For further information, see *Unicode and Code Page Support in the Natural Programming Language*, *Session Parameters*, section *EMU, ICU, LCU, TCU versus EM, IC, LC, TC*.

The parameters `LCU` and `ICU` cannot both be applied to one field.

Trailing Characters - TC Parameter

With the session parameter `TC`, you can specify trailing characters that are to be displayed immediately *to the right of a field* that is output with a `DISPLAY` statement. The width of the output column is enlarged accordingly. You can specify 1 to 10 characters.

The `TC` parameter can be used with the following statements:

- `FORMAT`
- `DISPLAY`

The `TC` parameter can be set at statement level and at element level.

Unicode Trailing Characters - TCU Parameter

The session parameter `TCU` is identical to the session parameter `TC`. The difference is that the trailing characters are always stored in Unicode format.

This allows you to specify trailing characters with mixed characters from different code pages, and assures that always the correct character is displayed independent of the installed system code page.

For further information, see *Unicode and Code Page Support in the Natural Programming Language*, *Session Parameters*, section *EMU, ICU, LCU, TCU versus EM, IC, LC, TC*.

Output Length - AL and NL Parameters

With the session parameter `AL`, you can specify the *output length for an alphanumeric field*; with the `NL` parameter, you can specify the *output length for a numeric field*. This determines the length of a field as it will be output, which may be shorter or longer than the actual length of the field (as defined in the DDM for a database field, or in the `DEFINE DATA` statement for a user-defined variable).

Both parameters can be used with the following statements:

- `FORMAT`
- `DISPLAY`
- `WRITE`
- `PRINT`
- `INPUT`

Both parameters can be set at statement level and at element level.

> **Note:** If an edit mask is specified, it overrides an `NL` or `AL` specification. **Edit masks** are described in ***Edit Masks - EM Parameter***.

Display Length for Output - DL Parameter

> **Note:** You should use the Web I/O Interface to make use of the full functionality of the `DL` parameter. When using the terminal emulation, it is not possible, for example, to scroll in a field when the value defined with `DL` is smaller than the field length.

With the session parameter `DL`, you can specify the *display length for a field of format A or U*, since the display width of a Unicode string can be twice the length of the string, and the user must be able to display the whole string. The default will be the length, for example, for a format/length U10, the display length can be 10 to 20, whereas the default length (when `DL` is not specified) is 10.

The session parameter `DL` can be used with the following statements:

- `FORMAT`
- `DISPLAY`
- `WRITE`
- `PRINT`

- INPUT

The session parameter DL can be set at statement level and at element level.

The difference between the session parameters AL and DL is that AL defines the data length of a field whereas DL defines the number of columns which are used on the screen for displaying the field. The user can scroll in input fields to view the entire content of a field if the value specified with the DL session parameter is less than the length of the field data.

Using the DL parameter with a length that is smaller than the length of the field is only recommended with the Web I/O Interface. When running Natural in a terminal emulation, scrolling in a field is not possible and so the effect is the same as using the AL parameter. Moreover, when changing the field contents, all characters which are beyond the display length will be lost.

Note: DL is allowed for A-format fields as well. In conjunction with the Web I/O Interface, this would allow making the edit control size smaller than the content of a field.

Example:

```
DEFINE DATA LOCAL
1    #U1 (U10)
1    #U2 (U10)
END-DEFINE
*
#U1 := U'latintxt00'
#U2 := U'特別是伺服器都需要支'
*
INPUT (AD=M) #U1 #U2
END
```

The above program produces the following output where the content of the field #U2 is incomplete:

```
#U1 latintxt00 #U2 特別是伺服
```

When the session parameter DL is used with the field #U2 and is specified accordingly, the content of this field will be displayed correctly:

```
DEFINE DATA LOCAL
1    #U1 (U10)
1    #U2 (U10)
END-DEFINE
*
#U1 := U'latintxt00'
#U2 := U'特別是伺服器都需要支'
*
INPUT (AD=M) #U1 #U2 (DL=20)
END
```

Result:

```
#U1 latintxt00 #U2 特别是伺服器都需要支
```

Sign Position - SG Parameter

With the session parameter SG, you can determine whether or not a sign position is to be allocated for numeric fields.

- By default, SG=ON applies, which means that a sign position is allocated for numeric fields.
- If you specify SG=OFF, negative values in numeric fields will be output without a minus sign (-).

The SG parameter can be used with the following statements:

- FORMAT
- DISPLAY
- PRINT
- WRITE
- INPUT

The SG parameter can be set at both statement level and element level.

Note: If an edit mask is specified, it overrides an SG specification. **Edit masks** are described in ***Edit Masks - EM Parameter***.

Example Program without Parameters

```
** Example 'FORMAX03': FORMAT (without FORMAT and compare with FORMAX04)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:1)
  2 BONUS  (1:1,1:1)
END-DEFINE
*
READ (5) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME
          FIRST-NAME
          SALARY (1:1)
          BONUS  (1:1,1:1)
END-READ
END
```

The above program contains no parameter settings and produces the following output:

```
Page      1                                                  04-11-11  11:11:11

        NAME                FIRST-NAME        ANNUAL     BONUS
                                              SALARY
-------------------- -------------------- ---------- ----------

JONES                VIRGINIA                  46000       9000
JONES                MARSHA                    50000          0
JONES                ROBERT                    31000          0
JONES                LILLY                     24000          0
JONES                EDWARD                    37600          0
```

Example Program with Parameters AL, NL, LC, IC and TC

In this example, the session parameters `AL`, `NL`, `LC`, `IC` and `TC` are used.

```
** Example 'FORMAX04': FORMAT (with parameters AL, NL, LC, TC, IC and
**                     compare with FORMAX03)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:1)
  2 BONUS  (1:1,1:1)
END-DEFINE
*
FORMAT AL=10 NL=6
*
READ (5) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME            (LC=*)
          FIRST-NAME      (TC=*)
          SALARY (1:1)    (IC=$)
          BONUS (1:1,1:1) (LC=>)
END-READ
END
```

The above program produces the following output. Compare the layout of this output with that of the previous program to see the effect of the individual parameters:

```
Page      1                                              04-11-11  11:11:11

   NAME       FIRST-NAME   ANNUAL   BONUS
                           SALARY
----------- ----------- -------- --------

*JONES      VIRGINIA  *   $46000 >   9000
*JONES      MARSHA    *   $50000 >      0
*JONES      ROBERT    *   $31000 >      0
*JONES      LILLY     *   $24000 >      0
*JONES      EDWARD    *   $37600 >      0
```

As you can see in the above example, any output length you specify with the `AL` or `NL` parameter does not include any characters specified with the `LC`, `IC` and `TC` parameters: the width of the `NAME` column, for example, is 11 characters - 10 for the field value (`AL=10`) plus 1 leading character.

The width of the `SALARY` and `BONUS` columns is 8 characters - 6 for the field value (`NL=6`), plus 1 leading/inserted character, plus 1 sign position (because `SG=ON` applies).

Identical Suppress - IS Parameter

With the session parameter `IS`, you can suppress the display of identical information in successive lines created by a `WRITE` or `DISPLAY` statement.

- By default, `IS=OFF` applies, which means that identical field values will be displayed.
- If `IS=ON` is specified, a value which is identical to the previous value of that field will not be displayed.

The `IS` parameter can be specified

- with a `FORMAT` statement to apply to the whole report, or
- in a `DISPLAY` or `WRITE` statement at both statement level and element level.

The effect of the parameter `IS=ON` can be suspended for one record by using the statement `SUSPEND IDENTICAL SUPPRESS`; see the *Statements* documentation for details.

Compare the output of the following two example programs to see the effect of the `IS` parameter. In the second one, the display of identical values in the `NAME` field is suppressed.

Example Program without IS Parameter

```
** Example 'FORMAX05': FORMAT (without parameter IS
**                     and compare with FORMAX06)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME FIRST-NAME
END-READ
END
```

The above program produces the following output:

```
  Page      1                                                04-11-11  11:11:11

        NAME              FIRST-NAME
  -------------------- --------------------

  JONES                VIRGINIA
  JONES                MARSHA
  JONES                ROBERT            ↩
```

Example Program with IS Parameter

```
** Example 'FORMAX06': FORMAT (with parameter IS
**                     and compare with FORMAX05)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
END-DEFINE
*
FORMAT IS=ON
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME FIRST-NAME
END-READ
END
```

The above program produces the following output:

```
Page      1                                          04-11-11  11:54:01

        NAME                FIRST-NAME
-------------------- --------------------

JONES                VIRGINIA
                     MARSHA
                     ROBERT
```

Zero Printing - ZP Parameter

With the profile and session parameter `ZP`, you determine how a field value of zero is to be displayed.

- By default, `ZP=ON` applies, which means that one `0` (for numeric fields) or all zeros (for time fields) will be displayed for each field value that is zero.
- If you specify `ZP=OFF`, the display of each field value which is zero will be suppressed.

The `ZP` parameter can be specified

- with a `FORMAT` statement to apply to the whole report, or
- in a `DISPLAY` or `WRITE` statement at both statement level and element level.

Compare the output of the following two **example programs** to see the effect of the parameters `ZP` and `ES`.

Empty Line Suppression - ES Parameter

With the session parameter `ES`, you can suppress the output of empty lines created by a `DISPLAY` or `WRITE` statement.

- By default, `ES=OFF` applies, which means that lines containing all blank values will be displayed.
- If `ES=ON` is specified, a line resulting from a `DISPLAY` or `WRITE` statement which contains all blank values will not be displayed. This is particularly useful when displaying multiple-value fields or fields which are part of a periodic group if a large number of empty lines are likely to be produced.

The `ES` parameter can be specified

- with a `FORMAT` statement to apply to the whole report, or
- in a `DISPLAY` or `WRITE` statement at statement level.

Note: To achieve empty suppression for numeric values, in addition to `ES=ON` the parameter `ZP=OFF` must also be set for the fields concerned in order to have null values turned into blanks and thus not output either.

Compare the output of the following two example programs to see the effect of the parameters `ZP` and `ES`.

Example Program without Parameters ZP and ES

```
** Example 'FORMAX07': FORMAT (without parameter ES and ZP
**                     and compare with FORMAX08)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BONUS (1:2,1:1)
END-DEFINE
*
READ (4) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME FIRST-NAME BONUS (1:2,1:1)
END-READ
END
```

The above program produces the following output:

```
Page      1                                                  04-11-11  11:58:23

        NAME                FIRST-NAME        BONUS
-------------------- -------------------- ----------

JONES                VIRGINIA                   9000
                                                6750
JONES                MARSHA                        0
                                                   0
JONES                ROBERT                        0
                                                   0
JONES                LILLY                         0
                                                   0
```

Example Program with Parameters ZP and ES

```
** Example 'FORMAX08': FORMAT (with parameters ES and ZP
**                     and compare with FORMAX07)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BONUS (1:2,1:1)
END-DEFINE
*
FORMAT ES=ON
*
READ (4) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME FIRST-NAME BONUS (1:2,1:1)(ZP=OFF)
END-READ
END
```

The above program produces the following output:

```
Page      1                                                04-11-11  11:59:09

        NAME                FIRST-NAME        BONUS
-------------------- -------------------- ----------

JONES                VIRGINIA                   9000
                                                6750
JONES                MARSHA
JONES                ROBERT
JONES                LILLY
```

Further Examples of Field-Output-Relevant Parameters

For further examples of the parameters `LC`, `IC`, `TC`, `AL`, `NL`, `IS`, `ZP` and `ES`, and the `SUSPEND IDENTICAL SUPPRESS` statement, see the following example programs:

- *DISPLX17 - DISPLAY (with NL, AL, IC, LC, TC)*
- *DISPLX18 - DISPLAY (using default settings for SF, AL, UC, LC, IC, TC and compare with DISPLX19)*
- *DISPLX19 - DISPLAY (with SF, AL, LC, IC, TC and compare with DISPLX18)*
- *SUSPEX01 - SUSPEND IDENTICAL SUPPRESS (in conjunction with parameters IS, ES, ZP in DISPLAY)*
- *SUSPEX02 - SUSPEND IDENTICAL SUPPRESS (in conjunction with parameters IS, ES, ZP in DISPLAY). Identical to SUSPEX01, but with IS=OFF.*

COMPRX03 - COMPRESS

37 Code Page Edit Masks - EM Parameter

This chapter describes how you can specify an edit mask for an alphanumeric or numeric field.

Use of EM Parameter

With the session parameter EM you can specify an edit mask for an alphanumeric or numeric field, that is, determine character by character the format in which the field values are to be output. Using the session parameter EMU, you can define edit masks with Unicode characters in the same way as described below for the EM session parameter.

Example:

```
DISPLAY NAME (EM=X^X^X^X^X^X^X^X^X^X)
```

In this example, each X represents one character of an alphanumeric field value to be displayed, and each ^ represents a blank. If displayed via the DISPLAY statement, the name JOHNSON would appear as follows:

```
J O H N S O N
```

You can specify the session parameter EM

- at report level (in a FORMAT statement),
- at statement level (in a DISPLAY, WRITE, INPUT, MOVE EDITED or PRINT statement) or
- at element level (in a DISPLAY, WRITE or INPUT statement).

An edit mask specified with the session parameter EM will override a default edit mask specified for a field in the **DDM**; see *Using the DDM Editor*, *Specifying Extended Field Attributes*.

If EM=OFF is specified, no edit mask at all will be used.

An edit mask specified at statement level will override an edit mask specified at report level.

An edit mask specified at element level will override an edit mask specified at statement level.

Edit Masks for Numeric Fields

An edit mask specified for a field of format N, P, I, or F must contain at least one 9 or Z. If more nines or Zs exist, the number of positions contained in the field value, the number of print positions in the edit mask will be adjusted to the number of digits defined for the field value. If fewer nines or Zs exist, the high-order digits before the decimal point and/or low-order digits after the decimal point will be truncated.

For further information, see session parameter `EM`, *Edit Masks for Numeric Fields* in the *Parameter Reference* documentation.

Edit Masks for Alphanumeric Fields

Edit masks for alphanumeric fields must include an `X` for each alphanumeric character that is to be output.

With a few exceptions, you may add leading, trailing and insertion characters (with or without enclosing them in apostrophes).

The circumflex character (^) is used to insert blanks in edit mask for both numeric and alphanumeric fields.

For further information, see session parameter `EM`, *Edit Masks for Alphanumeric Fields* in the *Parameter Reference* documentation.

Length of Fields

It is important to be aware of the length of the field to which you assign an edit mask.

- If the edit mask is longer than the field, this will yield unexpected results.
- If the edit mask is shorter than the field, the field output will be truncated to just those positions specified in the edit mask.

Examples:

Assuming an alphanumeric field that is 12 characters long and the field value to be output is `JOHNSON`, the following edit masks will yield the following results:

Edit Mask	Output
`EM=X.X.X.X.X`	`J.O.H.N.S`
`EM=****XXXXXX****`	`****JOHNSO**`

Edit Masks for Date and Time Fields

Edit masks for date fields can include the characters `D` (day), `M` (month) and `Y` (year) in various combinations.

Edit masks for time fields can include the characters `H` (hour), `I` (minute), `S` (second) and `T` (tenth of a second) in various combinations.

In conjunction with edit masks for date and time fields, see also the date and time system variables.

Customizing Separator Character Displays

Natural programs are used in business applications all over the world. Depending on the local conventions, it is usual to present numeric data fields and those with a date or time content in a special output style, when displayed in I/O statements. The different appearance should not be realized by alternate program coding that is processed selectively as a function of the locale where the program is being executed, but should be carried out with the same program image in conjunction with a set of runtime parameters to specify the decimal point character and the “thousands separator character”.

The following topics are covered below:

Decimal Separator

The Natural parameter `DC` is available to specify the character to be inserted in place of any characters used to represent the decimal separator (also called “radix” character) in edit masks. This parameter enables the users of a Natural program or application to choose any (special) character to separate the integer positions from the decimal positions of a numeric data item and enables, for example, U.S. shops to use the decimal point (.) and European shops to use the comma (,).

Dynamic Thousands Separator

To structure the output of a large integer values, it is common practice to insert separators between every three digits of an integer to separate groups of thousands. This separator is called a "thousands separator". For example, shops in the United States generally use a comma for this purpose (1,000,000), whereas shops in Germany use the period (1.000.000), in France a space (1 000 000), etc.

In a Natural edit mask, a "dynamic thousands separator" is a comma (or period) indicating the position where thousands separator characters (defined with the `THSEPCH` parameter) are inserted at runtime. At compile time, the Natural profile parameter `THSEP` or the option `THSEP` of system command `COMPOPT` enables or disables the interpretation of the comma (or period) as dynamic thousands separator.

If `THSEP` is set to `OFF` (default), any character used as thousands separator in the edit mask is treated as literal and displayed unchanged at runtime. This setting retains downwards compatibility.

If `THSEP` is set to `ON`, any comma (or period) in the edit mask is interpreted as dynamic thousands separators. In general, the dynamic thousands separator is a comma, but if the comma is already in use as decimal character (`DC`), the period is used as dynamic thousands separator.

At runtime the dynamic thousands separators are replaced by the current value of the `THSEPCH` parameter (thousands separator character).

Examples

A Natural program that is cataloged with parameter settings `DC='.'` and `THSEP=ON` uses the edit mask (`EM=ZZ,ZZZ,ZZ9.99`).

Parameter Settings at Runtime	Displays as
`DC='.'` and `THSEPCH=','`	`1,234,567.89`
`DC=','` and `THSEPCH='.'`	`1.234.567,89`
`DC=','` and `THSEPCH='/'`	`1/234/567,89`
`DC=','` and `THSEPCH=' '`	`1 234 567,89`
`DC=','` and `THSEPCH=''''`	`1'234'567,89`

Examples of Edit Masks

Some examples of edit masks, along with possible output they produce, are provided below.

In addition, the abbreviated notation for each edit mask is given. You can use either the abbreviated or the long notation.

Edit Mask	Abbreviation	Output A	Output B
EM=999.99	EM=9(3).9(2)	367.32	005.40
EM=ZZZZZ9	EM=Z(5)9(1)	0	579
EM=X^XXXXX	EM=X(1)^X(5)	B LUE	A 19379
EM=XXX...XX	EM=X(3)...X(2)	BLU...E	AAB...01
EM=MM.DD.YY	*	01.05.87	12.22.86
EM=HH.II.SS.T	**	08.54.12.7	14.32.54.3

[*] Use a date system variable.

[**] Use a time system variable.

For further information about edit masks, see the session parameter EM in the *Parameter Reference*.

Example Program without EM Parameters

```
** Example 'EDITMX01': Edit mask (using default edit masks)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 SALARY (1:3)
  2 CITY
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY 'N A M E'    NAME         /
          'OCCUPATION' JOB-TITLE
          'SALARY'     SALARY (1:3)
          'LOCATION'   CITY
  SKIP 1
END-READ
END
```

Output of Program EDITMX01:

The output of this program shows the default edit masks available.

```
Page      1                                                 04-11-11  14:15:54

         N A M E           SALARY         LOCATION
       OCCUPATION
------------------------- ---------- --------------------


JONES                          46000 TULSA
MANAGER                        42300
                               39300

JONES                          50000 MOBILE
DIRECTOR                       46000
                               42700

JONES                          31000 MILWAUKEE
PROGRAMMER                     29400
                               27600
```

Example Program with EM Parameters

```
** Example 'EDITMX02': Edit mask (using EM)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 JOB-TITLE
  2 SALARY (1:3)
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
 DISPLAY 'N A M E'    NAME         (EM=X^X^X^X^X^X^X^X^X^X^X^X^X^X^X)  /
                       FIRST-NAME   (EM=...X(10)...)
          'OCCUPATION' JOB-TITLE    (EM=' ___ 'X(12))
          'SALARY'     SALARY (1:3) (EM=' USD 'ZZZ,999)
  SKIP 1
END-READ
END
```

Output of Program `EDITMX02`:

Compare the output with that of the previous program (*Example Program without EM Parameters*) to see how the `EM` specifications affect the way the fields are displayed.

```
Page      1                                                04-11-11  14:15:54

           N A M E                OCCUPATION      SALARY
         FIRST-NAME
---------------------------- ---------------- -----------

J O N E S                    ___ MANAGER      USD  46,000
..VIRGINIA   ...                              USD  42,300
                                              USD  39,300

J O N E S                    ___ DIRECTOR     USD  50,000
..MARSHA     ...                              USD  46,000
                                              USD  42,700

J O N E S                    ___ PROGRAMMER   USD  31,000
..ROBERT     ...                              USD  29,400
                                              USD  27,600
```

Further Examples of Edit Masks

See the following example programs:

- *EDITMX03 - Edit mask (different EM for alpha-numeric fields)*
- *EDITMX04 - Edit mask (different EM for numeric fields)*
- *EDITMX05 - Edit mask (EM for date and time system variables)*

38 Unicode Edit Masks - EMU Parameter

Unicode edit masks can be used similar to code page edit masks. The difference is that the edit mask is always stored in Unicode format.

This allows you to specify edit masks with mixed characters from different code pages and assures that always the correct character is displayed, independent of the installed system code page.

For the general usage of edit masks, see ***Edit Masks - EM Parameter***.

For information on the session parameter `EMU`, see *EMU - Unicode Edit Mask* (in the *Parameter Reference*).

39 Vertical Displays

This chapter describes how you can combine the features of the statements DISPLAY and WRITE to produce vertical displays of field values.

Creating Vertical Displays

There are two ways of creating vertical displays:

- You can use a combination of the statements DISPLAY and WRITE.
- You can use the VERT option of the DISPLAY statement.

Combining DISPLAY and WRITE

As described in ***Statements DISPLAY and WRITE***, the DISPLAY statement normally presents the data in columns with default headers, while the WRITE statement presents data horizontally without headers.

You can combine the features of the two statements to produce vertical displays of field values.

The DISPLAY statement produces the values of different fields for the same record across the page with a column for each field. The field values for each record are displayed below the values for the previous record.

By using a WRITE statement after a DISPLAY statement, you can insert text and/or field values specified in the WRITE statement between records displayed via the DISPLAY statement.

The following program illustrates the combination of DISPLAY and WRITE:

```
** Example 'WRITEX04': WRITE (in combination with DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 CITY
  2 DEPT
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'SAN FRANCISCO'
  DISPLAY NAME JOB-TITLE
  WRITE 22T 'DEPT:' DEPT
  SKIP 1
END-READ
END
```

Output of Program WRITEX04:

```
Page      1                                                   04-11-11  14:15:55

        NAME                  CURRENT
                             POSITION
-------------------- -------------------------

KOLENCE              MANAGER
                     DEPT: TECH05

GOSDEN               ANALYST
                     DEPT: TECH10

WALLACE              SALES PERSON
                     DEPT: SALE20
```

Tab Notation - T*field

In the previous example, the position of the field DEPT is determined by the tab notation *n*T (in this case 20T, which means that the display begins in column 20 on the screen).

Field values specified in a WRITE statement can be lined up automatically with field values specified in the first DISPLAY statement of the program by using the tab notation T**field* (where *field* is the name of the field to which the field is to be aligned).

In the following program, the output produced by the WRITE statement is aligned to the field JOB-TITLE by using the notation T*JOB-TITLE:

```
** Example 'WRITEX05': WRITE  (in combination with DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 DEPT
  2 CITY
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'SAN FRANCISCO'
  DISPLAY NAME JOB-TITLE
  WRITE T*JOB-TITLE 'DEPT:' DEPT
  SKIP 1
END-READ
END
```

Output of Program WRITEX05:

```
Page      1                                                    04-11-11  14:15:55

        NAME                  CURRENT
                             POSITION
-------------------- -------------------------

KOLENCE              MANAGER
                     DEPT: TECH05

GOSDEN               ANALYST
                     DEPT: TECH10

WALLACE              SALES PERSON
                     DEPT: SALE20
```

Positioning Notation x/y

When you use the `DISPLAY` and `WRITE` statements in sequence and multiple lines are to be produced by the `WRITE` statement, you can use the notation *x/y* (number-slash-number) to determine in which row/column something is to be displayed. The positioning notation causes the next element in the `DISPLAY` or `WRITE` statement to be placed *x* lines below the last output, beginning in column *y* of the output.

The following program illustrates the use of this notation:

```
** Example 'WRITEX06': WRITE  (with n/n)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 MIDDLE-I
  2 ADDRESS-LINE (1:1)
  2 CITY
  2 ZIP
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  DISPLAY 'NAME AND ADDRESS' NAME
  WRITE   1/5  FIRST-NAME
          1/30 MIDDLE-I
          2/5  ADDRESS-LINE (1:1)
          3/5  CITY
          3/30 ZIP /
END-READ
END
```

Output of Program `WRITEX06`:

```
Page      1                                                    04-11-11  14:15:55

  NAME AND ADDRESS
--------------------

RUBIN
    SYLVIA                   L
    2003 SARAZEN PLACE
    NEW YORK                 10036

WALLACE
    MARY                     P
    12248 LAUREL GLADE C
    NEW YORK                 10036

KELLOGG
    HENRIETTA                S
    1001 JEFF RYAN DR.
    NEWARK                   19711
```

DISPLAY VERT Statement

The standard display mode in Natural is horizontal.

With the VERT clause option of the DISPLAY statement, you can override the standard display and produce a vertical field display.

The HORIZ clause option, which can be used in the same DISPLAY statement, re-activates the standard horizontal display mode.

Column headings in vertical mode are controlled with various forms of the AS clause. The following example programs illustrate the use of the DISPLAY VERT statement:

- DISPLAY VERT without AS Clause
- DISPLAY with VERT AS CAPTIONED and HORIZ Clause
- DISPLAY with VERT AS 'text' Clause
- DISPLAY with VERT AS 'text' CAPTIONED Clause

- Tab Notation P*field

DISPLAY VERT without AS Clause

The following program has no `AS` clause, which means that no column headings are output.

```
** Example 'DISPLX09': DISPLAY  (without column title)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  DISPLAY VERT NAME FIRST-NAME / CITY
  SKIP 2
END-READ
END
```

Output of Program `DISPLX09`:

Note that all field values are displayed vertically underneath one another.

```
Page      1                                                   04-11-11  14:15:54



RUBIN
SYLVIA

NEW YORK


WALLACE
MARY

NEW YORK


KELLOGG
HENRIETTA

NEWARK
```

DISPLAY with VERT AS CAPTIONED and HORIZ Clause

The following program contains a VERT and a HORIZ clause, which causes some column values to be output vertically and others horizontally; moreover AS CAPTIONED causes the default column headers to be displayed.

```
** Example 'DISPLX10': DISPLAY (with VERT as CAPTIONED and HORIZ clause)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 JOB-TITLE
  2 SALARY (1:1)
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  DISPLAY VERT AS CAPTIONED NAME FIRST-NAME
          HORIZ JOB-TITLE SALARY (1:1)
  SKIP 1
END-READ
END
```

Output of Program DISPLX10:

```
Page      1                                                 04-11-11  14:15:54

        NAME                     CURRENT             ANNUAL
     FIRST-NAME                  POSITION            SALARY
-------------------- ------------------------- ----------


RUBIN                SECRETARY                      17000
SYLVIA

WALLACE              ANALYST                        38000
MARY

KELLOGG              DIRECTOR                       52000
HENRIETTA
```

DISPLAY with VERT AS 'text' Clause

The following program contains an `AS 'text'` clause, which displays the specified `'text'` as column header.

Note: A slash (/) within the text element in a `DISPLAY` statement causes a line advance.

```
** Example 'DISPLX11': DISPLAY  (with VERT AS 'text' clause)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 JOB-TITLE
  2 SALARY (1:1)
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  DISPLAY VERT AS 'EMPLOYEES' NAME FIRST-NAME
          HORIZ JOB-TITLE SALARY (1:1)
  SKIP 1
END-READ
END
```

Output of Program `DISPLX11`:

```
Page      1                                                04-11-11  14:15:54

     EMPLOYEES                 CURRENT            ANNUAL
                              POSITION            SALARY
-------------------- ------------------------- ----------

RUBIN                SECRETARY                      17000
SYLVIA

WALLACE              ANALYST                        38000
MARY

KELLOGG              DIRECTOR                       52000
HENRIETTA
```

DISPLAY with VERT AS 'text' CAPTIONED Clause

The `AS 'text' CAPTIONED` clause causes the specified text to be displayed as column heading, and the default column headings to be displayed immediately before the field value in each line that is output.

The following program contains an `AS 'text' CAPTIONED` clause.

```
** Example 'DISPLX12': DISPLAY (with VERT AS 'text' CAPTIONED clause)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 JOB-TITLE
  2 SALARY (1:1)
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  DISPLAY VERT AS 'EMPLOYEES' CAPTIONED NAME FIRST-NAME
          HORIZ JOB-TITLE SALARY (1:1)
  SKIP 1
END-READ
END
```

Output of Program `DISPLX12`:

This clause causes the default column headers (`NAME` and `FIRST-NAME`) to be placed before the field values:

```
Page      1                                            04-11-11  14:15:54

           EMPLOYEES                      CURRENT            ANNUAL
                                         POSITION            SALARY
------------------------------- ------------------------- ----------

NAME RUBIN                      SECRETARY                      17000
FIRST-NAME SYLVIA

NAME WALLACE                    ANALYST                        38000
FIRST-NAME MARY

NAME KELLOGG                    DIRECTOR                       52000
FIRST-NAME HENRIETTA
```

Tab Notation P*field

If you use a combination of `DISPLAY VERT` statement and subsequent `WRITE` statement, you can use the tab notation `P*field-name` in the `WRITE` statement to align the position of a field to the column *and* line position of a particular field specified in the `DISPLAY VERT` statement.

In the following program, the fields `SALARY` and `BONUS` are displayed in the same column, `SALARY` in every first line, `BONUS` in every second line. The text `***SALARY PLUS BONUS***` is aligned to `SALARY`, which means that it is displayed in the same column as `SALARY` and in the first line, whereas the text `(IN US DOLLARS)` is aligned to `BONUS` and therefore displayed in the same column as `BONUS` and in the second line.

```
** Example 'WRITEX07': WRITE  (with P*field)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
  2 SALARY (1:1)
  2 BONUS  (1:1,1:1)
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'LOS ANGELES'
  DISPLAY NAME JOB-TITLE
          VERT AS 'INCOME' SALARY (1) BONUS (1,1)
  WRITE P*SALARY '***SALARY PLUS BONUS***'
        P*BONUS  '(IN US DOLLARS)'
  SKIP 1
END-READ
END
```

Output of Program `WRITEX07`:

```
Page      1                                                04-11-11  14:15:55

        NAME                 CURRENT            INCOME
                            POSITION
-------------------- ------------------------- ----------

SMITH                                                    0
                                                         0
                                               ***SALARY PLUS BONUS***
                                               (IN US DOLLARS)

POORE JR             SECRETARY                       25000
                                                         0
                                               ***SALARY PLUS BONUS***
                                               (IN US DOLLARS)
```

```
PREPARATA            MANAGER                        46000
                                                     9000
                                               ***SALARY PLUS BONUS***
                                               (IN US DOLLARS)
```

Further Example of DISPLAY VERT with WRITE Statement

See the following example program:

- ***WRITEX10 - WRITE (with `nT`, T*`field` and P*`field`)***

VII Further Programming Aspects

40 Text Notation

In an `INPUT`, `DISPLAY`, `WRITE`, `WRITE TITLE` or `WRITE TRAILER` statement, you can use text notation to define a text to be used in conjunction with such a statement.

Defining a Text to Be Used with a Statement - the 'text' Notation

The text to be used with the statement (for example, a prompting message) must be enclosed in either apostrophes (') or quotation marks ("). Do not confuse double apostrophes ('') with a quotation mark (").

Text enclosed in quotation marks can be converted automatically from lower-case letters to upper case. To switch off automatic conversion, change the settings in the editor profile.

For details, see the `CAPS` option in *Displaying and Hiding Profile Settings* in *Editor Basics* (*Editor Profile*, *Editors* documentation).

The text itself may be 1 to 72 characters and must not be continued from one line to the next.

Text elements may be concatenated by using a hyphen.

Examples:

```
DEFINE DATA LOCAL
1 #A(A10)
END-DEFINE

INPUT 'Input XYZ' (CD=BL) #A
WRITE '=' #A
WRITE 'Write1 ' - 'Write2 ' - 'Write3' (CD=RE)
END
```

Using Apostrophes as Part of a Text String

The following applies, if Natural profile parameter `TQMARK` (Translate Quotation Marks) is set to `ON`. This is the default setting.

If you want an apostrophe to be part of a text string that is enclosed in apostrophes, you must write this as double apostrophes ('') or as a quotation mark ("). Either notation will be output as a single apostrophe.

If you want an apostrophe to be part of a text string that is enclosed in quotation marks, you write this as a single apostrophe.

Examples of Apostrophe:

```
#FIELDA = 'O''CONNOR'
#FIELDA = 'O"CONNOR'
#FIELDA = "O'CONNOR"
```

In all three cases, the result will be:

```
O'CONNOR
```

Using Quotation Marks as Part of a Text String

The following applies, if the Natural profile parameter `TQ` (Translate Quotation Marks) is set to `OFF`. The default setting is `TQ=ON`.

If you want a quotation mark to be part of a text string that is enclosed in single apostrophes, write a quotation mark.

If you want a quotation mark to be part of a text string that is enclosed in quotation marks, write double quotation marks ("").

Example of Quotation Mark:

```
#FIELDA = 'O"CONNOR'
#FIELDA = "O""CONNOR"
```

In both cases, the result will be:

```
O"CONNOR
```

Defining a Character to Be Displayed n Times before a Field Value - the 'c'(n) Notation

If a single character is to be output several times as text, you use the following notation:

```
'c'(n)
```

As *c* you specify the character, and as *n* the number of times the character is to be generated. The maximum value for *n* is `249`.

Example:

```
WRITE '*'(3)
```

Instead of apostrophes before and after the character *c* you can also use quotation marks.

41 User Comments

User comments are descriptions or explanatory notes added to or interspersed among the statements of the source code. Such information may be particularly helpful in understanding and maintaining source code that was written or edited by another programmer. Also, the characters marking the beginning of a comment can be used to temporarily disable the function of a statement or several source code lines for test purposes.

Using an Entire Source Code Line for Comments

If you wish to use an entire source-code line for a user comment, you enter one of the following at the beginning of the line:

- an asterisk and a blank (*),
- two asterisks (**), or
- a slash and an asterisk (/*).

```
*  USER COMMENT
** USER COMMENT
/* USER COMMENT
```

Example:

As can be seen from the following example, comment lines may also be used to provide for a clear source code structure.

```
** Example 'LOGICX03': BREAK option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BIRTH
*
1 #BIRTH (A8)
END-DEFINE
*
LIMIT 10
READ EMPLOY-VIEW BY BIRTH
  MOVE EDITED BIRTH (EM=YYYYMMDD) TO #BIRTH
  /*
  IF BREAK OF #BIRTH /6/
    NEWPAGE IF LESS THAN 5 LINES LEFT
    WRITE / '-' (50) /
  END-IF
  /*
  DISPLAY NOTITLE BIRTH (EM=YYYY-MM-DD) NAME FIRST-NAME
```

```
END-READ
END
```

Using the Latter Part of a Source Code Line for Comments

If you wish to use only the latter part of a source-code line for a user comment, you enter a blank, a slash and an asterisk (/*); the remainder of the line after this notation is thus marked as a comment:

```
ADD 5 TO #A          /* USER COMMENT
```

Example:

```
** Example 'LOGICX04': IS option as format/length check
************************************************************************
DEFINE DATA LOCAL
1 #FIELDA (A10)          /* INPUT FIELD TO BE CHECKED
1 #FIELDB (N5)           /* RECEIVING FIELD OF VAL FUNCTION
1 #DATE   (A10)          /* INPUT FIELD FOR DATE
END-DEFINE
*
INPUT #DATE #FIELDA
IF #DATE IS(D)
  IF #FIELDA IS (N5)
    COMPUTE #FIELDB = VAL(#FIELDA)
    WRITE NOTITLE 'VAL FUNCTION OK' //  '=' #FIELDA '=' #FIELDB
  ELSE
    REINPUT 'FIELD DOES NOT FIT INTO N5 FORMAT'
            MARK *#FIELDA
  END-IF
ELSE
  REINPUT 'INPUT IS NOT IN DATE FORMAT (YY-MM-DD) '
          MARK *#DATE
END-IF
*
END
```

42 Data Computation

This chapter discusses arithmetic statements that are used for computing data:

- COMPUTE
- ADD
- SUBTRACT
- MULTIPLY
- DIVIDE

In addition, the following statements are discussed which are used to transfer the value of an operand into one or more fields:

- MOVE
- COMPRESS

Important: For optimum processing, user-defined variables used in arithmetic statements should be defined with format P (packed numeric).

COMPUTE Statement

The COMPUTE statement is used to perform arithmetic operations. The following connecting operators are available:

**	Exponentiation
*	Multiplication
/	Division
+	Addition
-	Subtraction
( )	Parentheses may be used to indicate logical grouping.

Example 1:

```
COMPUTE LEAVE-DUE = LEAVE-DUE * 1.1
```

In this example, the value of the field LEAVE-DUE is multiplied by 1.1, and the result is placed in the field LEAVE-DUE.

Example 2:

```
COMPUTE #A = SQRT (#B)
```

In this example, the square root of the value of the field #B is evaluated, and the result is assigned to the field #A.

SQRT is a mathematical function supported in the following arithmetic statements:

- COMPUTE
- ADD
- SUBTRACT
- MULTIPLY
- DIVIDE

For an overview of mathematical functions, see **Mathematical Functions** below.

Example 3:

```
COMPUTE #INCOME = BONUS (1,1) + SALARY (1)
```

In this example, the first bonus of the current year and the current salary amount are added and assigned to the field #INCOME.

Statements MOVE and COMPUTE

The statements MOVE and COMPUTE can be used to transfer the value of an operand into one or more fields. The operand may be a constant such as a text item or a number, a database field, a user-defined variable, a system variable, or, in certain cases, a system function.

The difference between the two statements is that in the MOVE statement the value to be moved is specified on the left; in the COMPUTE statement the value to be assigned is specified on the right, as shown in the following examples.

Examples:

```
MOVE NAME TO #LAST-NAME
COMPUTE #LAST-NAME = NAME
```

Statements ADD, SUBTRACT, MULTIPLY and DIVIDE

The `ADD`, `SUBTRACT`, `MULTIPLY` and `DIVIDE` statements are used to perform arithmetic operations.

Examples:

```
ADD +5 -2 -1 GIVING #A
SUBTRACT 6 FROM 11 GIVING #B
MULTIPLY 3 BY 4 GIVING #C
DIVIDE 3 INTO #D GIVING #E
```

All four statements have a `ROUNDED` option, which you can use if you wish the result of the operation to be rounded.

For rules on rounding, see ***Rules for Arithmetic Assignment***.

The *Statements* documentation provides more detailed information on these statements.

Example of MOVE, SUBTRACT and COMPUTE Statements

The following program demonstrates the use of **user-defined variables** in arithmetic statements. It calculates the ages and wages of three employees and outputs these.

```
** Example 'COMPUX01': COMPUTE
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 BIRTH
  2 JOB-TITLE
  2 SALARY          (1:1)
  2 BONUS           (1:1,1:1)
*
1 #DATE             (N8)
1 REDEFINE #DATE
  2 #YEAR           (N4)
  2 #MONTH          (N2)
  2 #DAY            (N2)
1 #BIRTH-YEAR       (A4)
1 REDEFINE #BIRTH-YEAR
```

```
  2 #BIRTH-YEAR-N  (N4)
1 #AGE             (N3)
1 #INCOME          (P9)
END-DEFINE
*
MOVE *DATN TO #DATE
*
READ (3) MYVIEW BY NAME STARTING FROM 'JONES'
  MOVE EDITED BIRTH (EM=YYYY) TO #BIRTH-YEAR
  SUBTRACT #BIRTH-YEAR-N FROM #YEAR GIVING #AGE
  /*
  COMPUTE #INCOME = BONUS (1:1,1:1) + SALARY (1:1)
  /*
  DISPLAY NAME 'POSITION' JOB-TITLE #AGE #INCOME
END-READ
END
```

Output of Program COMPUX01:

```
Page      1                                             14-01-14  14:15:54

        NAME                   POSITION          #AGE  #INCOME
-------------------- ------------------------- ---- ----------


JONES                MANAGER                     63      55000
JONES                DIRECTOR                    58      50000
JONES                PROGRAMMER                  48      31000
```

COMPRESS Statement

The COMPRESS statement is used to transfer (combine) the contents of two or more operands into a single alphanumeric field.

Leading zeros in a numeric field and trailing blanks in an alphanumeric field are suppressed before the field value is moved to the receiving field.

By default, the transferred values are separated from one another by a single blank in the receiving field. For other separating possibilities, see the COMPRESS statement option LEAVING NO SPACE (in the *Statements* documentation).

Example:

```
COMPRESS 'NAME:' FIRST-NAME #LAST-NAME INTO #FULLNAME
```

In this example, a `COMPRESS` statement is used to combine a text constant (`'NAME:'`), a database field (`FIRST-NAME`) and a user-defined variable (`#LAST-NAME`) into one user-defined variable (`#FULLNAME`).

For further information on the `COMPRESS` statement, please refer to the `COMPRESS` statement description (in the *Statements* documentation).

Example of COMPRESS and MOVE Statements

The following program illustrates the use of the statements `MOVE` and `COMPRESS`.

```
** Example 'COMPRX01': COMPRESS
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 MIDDLE-I
*
1 #LAST-NAME (A15)
1 #FULL-NAME (A30)
END-DEFINE
*
READ (3) MYVIEW BY NAME STARTING FROM 'JONES'
  MOVE NAME TO #LAST-NAME
  /*
  COMPRESS 'NAME:' FIRST-NAME MIDDLE-I #LAST-NAME INTO #FULL-NAME
  /*
  DISPLAY #FULL-NAME (UC==) FIRST-NAME 'I' MIDDLE-I (AL=1) NAME
END-READ
END
```

Output of Program `COMPRX01`:

Notice the output format of the compressed field.

```
Page      1                                                 14-01-14  14:15:54

          #FULL-NAME                FIRST-NAME      I         NAME
============================== -------------------- - --------------------

NAME: VIRGINIA J JONES         VIRGINIA             J JONES
NAME: MARSHA JONES             MARSHA                 JONES
NAME: ROBERT B JONES           ROBERT               B JONES
```

In multiple-line displays, it may be useful to combine fields/text in a **user-defined variables** by using a `COMPRESS` statement.

Examples of COMPRESS Statement

Example 1:

In the following program, three **user-defined variables** are used: `#FULL-SALARY`, `#FULL-NAME`, and `#FULL-CITY`. `#FULL-SALARY`, for example, contains the text `'SALARY:'` and the database fields `SALARY` and `CURR-CODE`. The `WRITE` statement then references only the compressed variables.

```
** Example 'COMPRX02': COMPRESS
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY       (1:1)
  2 CURR-CODE    (1:1)
  2 CITY
  2 ADDRESS-LINE (1:1)
  2 ZIP
*
1 #FULL-SALARY   (A25)
1 #FULL-NAME     (A25)
1 #FULL-CITY     (A25)
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  COMPRESS 'SALARY:' CURR-CODE(1) SALARY(1) INTO #FULL-SALARY
  COMPRESS FIRST-NAME NAME                  INTO #FULL-NAME
  COMPRESS ZIP CITY                         INTO #FULL-CITY
  /*
  DISPLAY 'NAME AND ADDRESS' NAME (EM=X^X^X^X^X^X^X^X^X^X^X^X)
  WRITE 1/5  #FULL-NAME
        1/37 #FULL-SALARY
        2/5  ADDRESS-LINE (1)
        3/5  #FULL-CITY
  SKIP 1
```

```
END-READ
END
```

Output of Program `COMPRX02`:

```
Page      1                                                  14-01-14  14:15:54

   NAME AND ADDRESS
-----------------------

R U B I N
    SYLVIA RUBIN                    SALARY: USD 17000
    2003 SARAZEN PLACE
    10036 NEW YORK

W A L L A C E
    MARY WALLACE                    SALARY: USD 38000
    12248 LAUREL GLADE C
    10036 NEW YORK

K E L L O G G
    HENRIETTA KELLOGG               SALARY: USD 52000
    1001 JEFF RYAN DR.
    19711 NEWARK
```

Example 2:

The following program is similar to COMPRX02, but the three database fields `NAME`, `SALARY`, and `COUNTRY` are treated with an edit mask in the `COMPRESS` statement.

```
** Example 'COMPRX04': COMPRESS with Edit Mask (EM)
** see similar example COMPRX02 with DISPLAY statement etc.
************************************************************************
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
  2 COUNTRY
*
1 #FULL-SALARY   (A25)
1 #FULL-NAME     (A25)
1 #FULL-CITY     (A25)
1 #COUNTRY       (A10)
1 #NAME          (A25)
END-DEFINE
*
```

```
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  COMPRESS NAME (EM=X^X^X^X^X^X^X^X^X^X^X^X) INTO #NAME
  COMPRESS FIRST-NAME NAME                    INTO #FULL-NAME
  COMPRESS 'SALARY:' CURR-CODE(1)
           SALARY(1) (EM=ZZZ,ZZ9)             INTO #FULL-SALARY
  COMPRESS ZIP CITY                           INTO #FULL-CITY
  COMPRESS COUNTRY (EM=X'-'X'-'X)             INTO #COUNTRY
  /*
  DISPLAY 'NAME AND ADDRESS' #NAME
  WRITE 1/5  #FULL-NAME
        1/37 #FULL-SALARY
        2/5  ADDRESS-LINE (1)
        3/5  #FULL-CITY
        4/5  #COUNTRY
  SKIP 1
END-READ
END
```

Output of Program `COMPRX04`:

```
Page      1                                                  21-12-20  17:25:24

    NAME AND ADDRESS
-------------------------

R U B I N
    SYLVIA RUBIN                    SALARY: USD  17,000
    2003 SARAZEN PLACE
    10036 NEW YORK
    U-S-A
                                                                                  ↩

W A L L A C E
    MARY WALLACE                    SALARY: USD  38,000
    12248 LAUREL GLADE C
    10036 NEW YORK
    U-S-A

K E L L O G G
    HENRIETTA KELLOGG               SALARY: USD  52,000
    1001 JEFF RYAN DR.
    19711 NEWARK
    U-S-A
```

Mathematical Functions

The following Natural mathematical functions are supported in arithmetic processing statements (ADD, COMPUTE, DIVIDE, SUBTRACT, MULTIPLY).

Mathematical Function	Natural System Function
Absolute value of *field*.	ABS(*field*)
Arc tangent of *field*.	ATN(*field*)
Cosine of *field*.	COS(*field*)
Exponential of *field*.	EXP(*field*)
Fractional part of *field*.	FRAC(*field*)
Integer part of *field*.	INT(*field*)
Natural logarithm of *field*.	LOG(*field*)
Sign of *field*.	SGN(*field*)
Sine of *field*.	SIN(*field*)
Square root of *field*.	SQRT(*field*)
Tangent of *field*.	TAN(*field*)
Numeric value of an alphanumeric *field*.	VAL(*field*)

See also the *System Functions* documentation for a detailed explanation of each mathematical function.

Further Examples of COMPUTE, MOVE and COMPRESS Statements

See the following example programs:

- *WRITEX11 - WRITE (with nX, n/n and COMPRESS)*
- *IFX03 - IF statement*
- *COMPRX03 - COMPRESS (using parameters LC and TC)*

43 Rules for Arithmetic Assignment

Field Initialization

A field (user-defined variable or database field) which is to be used as an operand in an arithmetic operation must be defined with one of the following formats:

Format	
N	Numeric unpacked
P	Packed numeric
I	Integer
F	Floating point
D	Date
T	Time

Note: For reporting mode: A field which is to be used as an operand in an arithmetic operation must have been previously defined. A user-defined variable or database field used as a result field in an arithmetic operation need not have been previously defined.

All user-defined variables and all database fields defined in a `DEFINE DATA` statement are initialized to the appropriate zero or blank value when the program is invoked for execution.

Data Transfer

Data transfer is performed with a `MOVE` or `COMPUTE` statement. The following table summarizes the data transfer compatibility of the formats an operand may take.

Sending Field Format	Receiving Field Format												
	N or P	A	U	B*n* (*n*<5)	B*n* (*n*>4)	I	L	C	D	T	F	G	O
N or P	Y	[2]	[14]	[3]	-	Y	-	-	-	Y	Y	-	-
A	-	Y	[13]	[1]	[1]	-	-	-	-	-	-	-	-
U	-	[11]	Y	[12]	[12]	-	-	-	-	-	-	-	-
B*n* (*n*<5)	[4]	[2]	[14]	[5]	[5]	Y	-	-	-	Y	Y	-	-
B*n* (*n*>4)	-	[6]	[15]	[5]	[5]	-	-	-	-	-	-	-	-
I	Y	[2]	[14]	[3]	-	Y	-	-	-	Y	Y	-	-
L	-	[9]	[16]	-	-	-	Y	-	-	-	-	-	-
C	-	-	-	-	-	-	-	Y	-	-	-	-	-

D	Y	[9]	[16]	Y	-	Y	-	-	Y	[7]	Y	-	-
T	Y	[9]	[16]	Y	-	Y	-	-	[8]	Y	Y	-	-
F	Y	[9] [10]	[10] [16]	[3]	-	Y	-	-	-	Y	Y	-	-
G	-	-	-	-	-	-	-	-	-	-	-	Y	-
O	-	-	-	-	-	-	-	-	-	-	-	-	Y

Where:

Y	Indicates data transfer compatibility.
-	Indicates data transfer incompatibility.
[ ]	Numbers in brackets [ ] refer to the corresponding rule for data transfer given below.

Data Conversion

The following rules apply to converting data values:

1. **Alphanumeric to binary:**
   The value will be moved byte by byte from left to right. The result may be truncated or padded with trailing blank characters depending on the length defined and the number of bytes specified.
2. **(N,P,I) and binary (length 1-4) to alphanumeric:**
   The value will be converted to unpacked form and moved into the alphanumeric field left justified, that is, leading zeros will be suppressed and the field will be filled with trailing blank characters. For negative numeric values, the sign will be converted to the hexadecimal notation `Dx`. Any decimal point in the numeric value will be ignored. All digits before and after the decimal point will be treated as one integer value.
3. **(N,P,I,F) to binary (1-4 bytes):**
   The numeric value will be converted to binary (4 bytes). Any decimal point in the numeric value will be ignored (the digits of the value before and after the decimal point will be treated as an integer value). The resulting binary number will be positive or a two's complement of the number depending on the sign of the value.
4. **Binary (1-4 bytes) to numeric:**
   The value will be converted and assigned to the numeric value right justified, that is, with leading zeros. (Binary values of the length 1-3 bytes are always assumed to have a positive sign. For binary values of 4 bytes, the leftmost bit determines the sign of the number: 1=negative, 0=positive.) Any decimal point in the receiving numeric value will be ignored. All digits before and after the decimal point will be treated as one integer value.
5. **Binary to binary:**
   The value will be moved from right to left byte by byte. Leading binary zeros will be inserted into the receiving field.

6. **Binary (>4 bytes) to alphanumeric:**
   The value will be moved byte by byte from left to right. The result may be truncated or padded with trailing blanks depending on the length defined and the number of bytes specified.
7. **Date (D) to time (T):**
   If date is moved to time, it is converted to time assuming time 00:00:00:0.
8. **Time (T) to date (D):**
   If time is moved to date, the time information is truncated, leaving only the date information.
9. **L,D,T,F to A:**
   The values are converted to display form and are assigned left justified.
10. **F:**
    If F is assigned to an alphanumeric or Unicode field which is too short, the mantissa is reduced accordingly.
11. **Unicode to alphanumeric:**
    The Unicode value will be converted to alphanumeric character codes according to the default code page (value of the system variable `*CODEPAGE`) using the International Components for Unicode (ICU) library. The result may be truncated or padded with trailing blank characters, depending on the length defined and the number of bytes specified.
12. **Unicode to binary:**
    The value will be moved code unit by code unit from left to right. The result may be truncated or padded with trailing blank characters, depending on the length defined and the number of bytes specified. The length of the receiving binary field must be even.
13. **Alphanumeric to Unicode:**
    The alphanumeric value will be converted from the default code page to a Unicode value using the International Components for Unicode (ICU) library. The result may be truncated or padded with trailing blank characters, depending on the length defined and the number of code units specified.
14. **(N,P,I) and binary (length 1-4) to Unicode:**
    The value will be converted to unpacked form from which an alphanumeric value will be obtained by suppression of leading zeros. For negative numeric values, the sign will be converted to the hexadecimal notation `Dx`. Any decimal point in the numeric value will be ignored. All digits before and after the decimal point will be treated as one integer value. The resulting value will be converted from alphanumeric to Unicode. The result may be truncated or padded with trailing blank characters, depending on the length defined and the number of code units specified.
15. **Binary (>4 bytes) to Unicode:**
    The value will be moved byte by byte from left to right. The result may be truncated or padded with trailing blanks, depending on the length defined and the number of bytes specified. The length of the sending binary field must be even.

16. **L,D,T,F to U:**
The values are converted to an alphanumeric display form. The resulting value will be converted from alphanumeric to Unicode and assigned left justified.

If source and target format are identical, the result may be truncated or padded with trailing blank characters (format A and U) or leading binary zeros (format B) depending on the length defined and the number of bytes (format A and B) or code units (format U) specified.

See also ***Using Dynamic Variables***.

Field Truncation and Field Rounding

The following rules apply to field truncation and rounding:

- High-order numeric field truncation is allowed only when the digits to be truncated are leading zeros. Digits following an expressed or implied decimal point may be truncated.
- Trailing positions of an alphanumeric field may be truncated.
- If the option `ROUNDED` is specified, the last position of the result will be rounded up if the first truncated decimal position of the value being assigned contains a value greater than or equal to 5. For the result precision of a division, see also ***Precision of Results of Arithmetic Operations***.

Result Format and Length in Arithmetic Operations

The following table shows the format and length of the result of an arithmetic operation:

	I1	I2	I4	N or P	F4	F8
I1	I1	I2	I4	P*	F4	F8
I2	I2	I2	I4	P*	F4	F8
I4	I4	I4	I4	P*	F4	F8
N or P	P*	P*	P*	P*	F4	F8
F4	F4	F4	F4	F4	F4	F8
F8	F8	F8	F8	F8	F8	F8

On a mainframe computer, format/length F8 is used instead of F4 for improved precision of the results of an arithmetic operation.

P* is determined from the integer length and precision of the operands individually for each operation, as shown under ***Precision of Results of Arithmetic Operations***.

The following decimal integer lengths and possible values are applicable for format I:

Format/Length	Decimal Integer Length	Possible Values
I1	3	-128 to 127
I2	5	-32768 to 32767
I4	10	-2147483648 to 2147483647

Arithmetic Operations with Floating-Point Numbers

The following topics are covered below:

- General Considerations
- Precision of Floating-Point Numbers
- Conversion to Floating-Point Representation
- Platform Dependency

General Considerations

Floating-point numbers (format F) are represented as a sum of powers of two (as are integer numbers (format I)), whereas unpacked and packed numbers (formats N and P) are represented as a sum of powers of ten.

In unpacked or packed numbers, the position of the decimal point is fixed. In floating-point numbers, however, the position of the decimal point (as the name indicates) is “floating”, that is, its position is not fixed, but depends on the actual value.

Floating-point numbers are essential for the computing of trigonometric functions or mathematical functions such as sinus or logarithm.

Precision of Floating-Point Numbers

Due to the nature of floating-point numbers, their precision is limited:

- For a variable of format/length F4, the precision is limited to approximately 7 digits.
- For a variable of format/length F8, the precision is limited to approximately 15 digits.

Values which have more significant digits cannot be represented exactly as a floating-point number. No matter how many additional digits there are before or after the decimal point, a floating-point number can cover only the leading 7 or 15 digits respectively.

An integer value can only be represented exactly in a variable of format/length F4 if its absolute value does not exceed $2^{23}-1$.

Conversion to Floating-Point Representation

When an alphanumeric, unpacked numeric or packed numeric value is converted to floating-point format (for example, in an assignment operation), the representation has to be changed, that is, a sum of powers of ten has to be converted to a sum of powers of two.

Consequently, only numbers that are representable as a finite sum of powers of two can be represented exactly; all other numbers can only be represented approximately.

Examples:

This number has an exact floating-point representation:

$$1.25 = 2^{0} + 2^{-2}$$

This number is a periodic floating-point number without an exact representation:

$$1.2 = 2^{0} + 2^{-3} + 2^{-4} + 2^{-7} + 2^{-8} + 2^{-11} + 2^{-12} + \ldots$$

Thus, the conversion of alphanumeric, unpacked numeric or packed numeric values to floating-point values, and vice versa, can introduce small errors.

Platform Dependency

Because of different hardware architecture, the representation of floating-point numbers varies according to platforms. This explains why the same application, when run on different platforms, may return slightly different results when floating-point arithmetic are involved. The respective representation also determines the range of possible values for floating-point variables, which is (approximately)

- $\pm 1.17 * 10^{-38}$ to $\pm 3.40 * 10^{38}$ for F4 variables,
- $\pm 2.22 * 10^{-308}$ to $\pm 1.79 * 10^{308}$ for F8 variables.

Note: The representation used by your pocket calculator may also be different from the one used by your computer - which explains why results for the same computation may differ.

Arithmetic Operations with Date and Time

With formats D (date) and T (time), only addition, subtraction, multiplication and division are allowed. Multiplication and division are allowed on intermediate results of additions and subtractions only.

Date/time values can be added to/subtracted from one another; or integer values (no decimal digits) can be added to/subtracted from date/time values. Such integer values can be contained in fields of formats N, P, I, D, or T.

The intermediate results of such an addition or subtraction may be used as a multiplicand or dividend in a subsequent operation.

An integer value added to/subtracted from a date value is assumed to be in days. An integer value added to/subtracted from a time value is assumed to be in tenths of seconds.

For arithmetic operations with date and time, certain restrictions apply, which are due to the Natural's internal handling of arithmetic operations with date and time, as explained below.

Internally, Natural handles an arithmetic operation with date/time variables as follows:

```
COMPUTE result-field = operand1 +/- operand2
```

The above statement is resolved as:

1. *intermediate-result = operand1 +/- operand2*
2. *result-field = intermediate-result*

That is, in a first step Natural computes the result of the addition/subtraction, and in a second step assigns this result to the result field.

More complex arithmetic operations are resolved following the same pattern:

```
COMPUTE result-field = operand1 +/- operand2 +/- operand3 +/- operand4
```

The above statement is resolved as:

1. *intermediate-result1 = operand1 +/- operand2*
2. *intermediate-result2 = intermediate-result1 +/- operand3*
3. *intermediate-result3 = intermediate-result2 +/- operand4*
4. *result-field = intermediate-result3*

The resolution of multiplication and division operations is similar to the resolution for addition and subtraction.

The internal format of such an intermediate result depends on the formats of the operands, as shown in the tables below.

Addition

The following table shows the format of the intermediate result of an addition (*intermediate-result = operand1 + operand2*):

Format of *operand1*	Format of *operand2*	Format of *intermediate-result*
D	D	Di
D	T	T
D	Di, Ti, N, P, I	D
T	D, T, Di, Ti, N, P, I	T
Di, Ti, N, P, I	D	D
Di, Ti, N, P, I	T	T
Di, N, P, I	Di	Di
Ti, N, P, I	Ti	Ti
Di	Ti, N, P, I	Di
Ti	Di, N, P, I	Ti

Subtraction

The following table shows the format of the intermediate result of a subtraction (*intermediate-result = operand1 - operand2*):

Format of *operand1*	Format of *operand2*	Format of *intermediate-result*
D	D	Di
D	T	Ti
D	Di, Ti, N, P, I	D
T	D, T	Ti
T	Di, Ti, N, P, I	T
Di, N, P, I	D	Di
Di, N, P, I	T	Ti
Di	Di, Ti, N, P, I	Di
Ti	D, T, Di, Ti, N, P, I	Ti
N, P, I	Di, Ti	P12

Multiplication or Division

The following table shows the format of the intermediate result of a multiplication (`intermediate-result = operand1 * operand2`) or division (`intermediate-result = operand1 / operand2`):

Format of `operand1`	Format of `operand2`	Format of `intermediate-result`
D	D, Di, Ti, N, P, I	Di
D	T	Ti
T	D, T, Di, Ti, N, P, I	Ti
Di	T	Ti
Di	D, Di, Ti, N, P, I	Di
Ti	D	Di
Ti	Di, T, Ti, N, P, I	Ti
N, P, I	D, Di	Di
N, P, I	T, Ti	Ti

Internal Assignments

Di is a value in internal date format; Ti is a value in internal time format; such values can be used in further arithmetic date/time operations, but they cannot be assigned to a result field of format D (see the assignment table below).

In complex arithmetic operations in which an intermediate result of internal format Di or Ti is used as operand in a further addition/subtraction/multiplication/division, its format is assumed to be D or T respectively.

The following table shows which intermediate results can internally be assigned to which result fields (`result-field = intermediate-result`).

Format of `result-field`	Format of `intermediate-result`	Assignment possible
D	D, T	yes
D	Di, Ti, N, P, I	no
T	D, T, Di, Ti, N, P, I	yes
N, P, I	D, T, Di, Ti, N, P, I	yes

A result field of format D or T must not contain a negative value.

Examples 1 and 2 (invalid):

```
COMPUTE DATE1 (D) = DATE2 (D) + DATE3 (D)
COMPUTE DATE1 (D) = DATE2 (D) - DATE3 (D)
```

These operations are not possible, because the intermediate result of the addition/subtraction would be format Di, and a value of format D*i* cannot be assigned to a result field of format D.

Examples 3 and 4 (invalid):

```
COMPUTE DATE1 (D) = TIME2 (T) - TIME3 (T)
COMPUTE DATE1 (D) = DATE2 (D) - TIME3 (T)
```

These operations are not possible, because the intermediate result of the addition/subtraction would be format Ti, and a value of format Ti cannot be assigned to a result field of format D.

Example 5 (valid):

```
COMPUTE DATE1 (D) = DATE2 (D) - DATE3 (D) + TIME3 (T)
```

This operation is possible. First, `DATE3` is subtracted from `DATE2`, giving an intermediate result of format Di; then, this intermediate result is added to `TIME3`, giving an intermediate result of format T; finally, this second intermediate result is assigned to the result field `DATE1`.

Examples 6 and 7 (invalid):

```
COMPUTE DATE1 (D) = DATE2 (D) + DATE3 (D) * 2
COMPUTE TIME1 (T) = TIME2 (T) - TIME3 (T) / 3
```

These operations are not possible, because the attempted multiplication/division is performed with date/time fields and not with intermediate results.

Example 8 (valid):

```
COMPUTE DATE1 (D) = DATE2 (D) + (DATE3(D) - DATE4 (D)) * 2
```

This operation is possible. First, `DATE4` is subtracted from `DATE3` giving an intermediate result of format Di; then, this intermediate result is multiplied by two giving an intermediate result of format Di; this intermediate result is added to `DATE2` giving an intermediate result of format D; finally, this third intermediate result is assigned to the result field `DATE1`.

If a format T value is assigned to a format D field, you must ensure that the time value contains a valid date component.

Performance Considerations for Mixed Format Expressions

When doing arithmetic operations, the choice of field formats has considerable impact on performance:

For business arithmetic, only fields of format I (integer) should be used, if possible.

For scientific arithmetic, fields of format F (floating point) should be used, if possible.

In expressions where formats are mixed between numeric (N, P) and floating point (F), a conversion to floating point format is performed. This conversion results in considerable CPU load. Therefore it is recommended to avoid mixed format expressions in arithmetic operations.

Precision of Results of Arithmetic Operations

Operation	Digits Before Decimal Point	Digits After Decimal Point
Addition/Subtraction	Fi + 1 or Si + 1 (whichever is greater)	Fd or Sd (whichever is greater)
Multiplication	Fi + Si	■ If Fd + Sd is less than `MAXPREC`: Fd + Sd ■ If Fd + Sd is greater than or equal to `MAXPREC`: Fd, Sd or `MAXPREC` (whichever is greater)
Division	Fi + Sd	(see below)
Exponentiation	29 - Fd (See *Exception* below)	Fd

- where:

F	First operand
S	Second operand
R	Result
i	Digits before decimal point
d	Digits after decimal point

Exception:

If the exponent has one or more digits after the decimal point, the exponentiation is internally carried out in floating point format and the result will also have floating point format. See *Arithmetic Operations with Floating-Point Numbers* for further information.

Digits after Decimal Point for Division Results

The precision of the result of a division depends whether a result field is available or not:

- If a result field is available, the precision is: Fd or Rd (whichever is greater) *.
- If no result field is available, the precision is: Fd or Sd (whichever is greater) *.

* If the `ROUNDED` option is used, the precision of the result is internally increased by one digit before the result is actually rounded.

A result field is available (or assumed to be available) in a `COMPUTE` and `DIVIDE` statement, and in a logical condition in which the division is placed after the comparison operator (for example: `IF #A = #B / #C THEN ...`).

A result field is not (or assumed to be not) available in a logical condition in which the division is placed before the comparison operator (for example: `IF #B / #C = #A THEN ...`).

Exception:

If both dividend and divisor are of integer format and at least one of them is a variable, the division result is always of integer format (regardless of the precision of the result field and of whether the `ROUNDED` option is used or not).

Precision of Results for Arithmetic Expressions

The precision of arithmetic expressions, for example: `#A / (#B * #C) + #D * (#E - #F + #G)`, is derived by evaluating the results of the arithmetic operations in their processing order. For further information on arithmetic expressions, see *`arithmetic-expression`* in the `COMPUTE` statement description.

Error Conditions in Arithmetic Operations

In an addition, subtraction, multiplication or division, an error can occur if the total number of digits (before and after the decimal point) of the result is greater than 31.

In an exponentiation, an error occurs in any of the following situations:

- if the base is of packed format with precision digits (for example, P3.2) and an exponent greater than 16;
- if the base is of floating-point format and the result is greater than approximately $7 * 10^{75}$.

Processing of Arrays

Generally, the following rules apply:

- All scalar operations may be applied to array elements which consist of a single occurrence.
- If a variable is defined with a constant value (for example, `#FIELD (I2) CONSTANT <8>`), the value will be assigned to the variable at compilation, and the variable will be treated as a constant. This means that if such a variable is used in an array index, the dimension concerned has a *definite* number of occurrences.
- If an assignment/comparison operation involves two arrays with a different number of dimensions, the "missing" dimension in the array with fewer dimensions is assumed to be (1:1).

  Example: If `#ARRAY1 (1:2)` is assigned to `#ARRAY2 (1:2,1:2)`, `#ARRAY1` is assumed to be `#ARRAY1 (1:1,1:2)`.

The following topics are covered below:

Definitions of Array Dimensions

The first, second and third dimensions of an array are defined as follows:

Number of Dimensions	Properties
3	#a3(3rd dim., 2nd dim., 1st dim.)
2	#a2(2nd dim., 1st dim.)
1	#a1(1st dim.)

Assignment Operations with Arrays

If an array range is assigned to another array range, the assignment is performed element by element.

Example:

```
DEFINE DATA LOCAL
1 #ARRAY(I4/1:5) INIT <10,20,30,40,50>
END-DEFINE
*
MOVE #ARRAY(2:4) TO #ARRAY(3:5)
/* is identical to
/* MOVE #ARRAY(2) TO #ARRAY(3)
/* MOVE #ARRAY(3) TO #ARRAY(4)
/* MOVE #ARRAY(4) TO #ARRAY(5)
/*
/* #ARRAY contains  10,20,20,20,20
```

If a single occurrence is assigned to an array range, each element of the range is filled with the value of the single occurrence. (For a mathematical function, each element of the range is filled with the result of the function.)

Before an assignment operation is executed, the individual dimensions of the arrays involved are compared with one another to check if they meet one of the conditions listed below. The dimensions are compared independently of one another; that is, the 1st dimension of the one array is compared with the 1st dimension of the other array, the 2nd dimension of the one array is compared with the 2nd dimension of the other array, and the 3rd dimension of the one array is compared with the 3rd dimension of the other array.

The assignment of values from one array to another is only allowed under one of the following conditions:

- The number of occurrences is the same for both dimensions compared.
- The number of occurrences is indefinite for both dimensions compared.
- The dimension that is assigned to another dimension consists of a single occurrence.

Example - Array Assignments:

The following program shows which array assignment operations are possible.

```
DEFINE DATA LOCAL
1 A1   (N1/1:8)
1 B1   (N1/1:8)
1 A2   (N1/1:8,1:8)
1 B2   (N1/1:8,1:8)
1 A3   (N1/1:8,1:8,1:8)
1 I    (I2)          INIT <4>
1 J    (I2)          INIT <8>
1 K    (I2)          CONST <8>
END-DEFINE
*
COMPUTE A1(1:3) = B1(6:8)                                    /* allowed
COMPUTE A1(1:I) = B1(1:I)                                    /* allowed
COMPUTE A1(*)   = B1(1:8)                                    /* allowed
COMPUTE A1(2:3) = B1(I:I+1)                                  /* allowed
```

```
 COMPUTE A1(1)   = B1(I)                                          /* allowed
 COMPUTE A1(1:I) = B1(3)                                          /* allowed
 COMPUTE A1(I:J) = B1(I+2)                                        /* allowed
 COMPUTE A1(1:I) = B1(5:J)                                        /* allowed
 COMPUTE A1(1:I) = B1(2)                                          /* allowed
 COMPUTE A1(1:2) = B1(1:J)                                        /* NOT ALLOWED ↩
(NAT0631)
 COMPUTE A1(*)   = B1(1:J)                                        /* NOT ALLOWED ↩
(NAT0631)
 COMPUTE A1(*)   = B1(1:K)                                        /* allowed
 COMPUTE A1(1:J) = B1(1:K)                                        /* NOT ALLOWED ↩
(NAT0631)
 *
 COMPUTE A1(*)       = B2(1,*)                                    /* allowed
 COMPUTE A1(1:3)     = B2(1,I:I+2)                                /* allowed
 COMPUTE A1(1:3)     = B2(1:3,1)                                  /* NOT ALLOWED ↩
(NAT0631)
 *
 COMPUTE A2(1,1:3)   = B1(6:8)                                    /* allowed
 COMPUTE A2(*,1:I)   = B1(5:J)                                    /* allowed
 COMPUTE A2(*,1)     = B1(*)                                      /* NOT ALLOWED ↩
(NAT0631)
 COMPUTE A2(1:I,1)   = B1(1:J)                                    /* NOT ALLOWED ↩
(NAT0631)
 COMPUTE A2(1:I,1:J) = B1(1:J)                                    /* allowed
 *
 COMPUTE A2(1,I)     = B2(1,1)                                    /* allowed
 COMPUTE A2(1:I,1)   = B2(1:I,2)                                  /* allowed
 COMPUTE A2(1:2,1:8) = B2(I:I+1,*)                                /* allowed
 *
 COMPUTE A3(1,1,1:I)   = B1(1)                                    /* allowed
 COMPUTE A3(1,1,1:J)   = B1(*)                                    /* NOT ALLOWED ↩
(NAT0631)
  COMPUTE A3(1,1,1:I)   = B1(1:I)                                 /* allowed
 COMPUTE A3(1,1:2,1:I) = B2(1,1:I)                                /* allowed
 COMPUTE A3(1,1,1:I)   = B2(1:2,1:I)                              /* NOT ALLOWED ↩
(NAT0631)
 END
```

Comparison Operations with Arrays

Generally, the following applies: if arrays with multiple dimensions are compared, the individual dimensions are handled independently of one another; that is, the 1st dimension of the one array is compared with the 1st dimension of the other array, the 2nd dimension of the one array is compared with the 2nd dimension of the other array, and the 3rd dimension of the one array is compared with the 3rd dimension of the other array.

The comparison of two array dimensions is only allowed under one of the following conditions:

- The array dimensions compared with one another have the same number of occurrences.

- The array dimensions compared with one another have an indefinite number of occurrences.
- All array dimensions of one of the arrays involved are single occurrences.

Example - Array Comparisons:

The following program shows which array comparison operations are possible:

```
DEFINE DATA LOCAL
1 A3  (N1/1:8,1:8,1:8)
1 A2  (N1/1:8,1:8)

1 A1  (N1/1:8)
1 I   (I2)   INIT <4>
1 J   (I2)   INIT <8>
1 K   (I2)   CONST <8>
END-DEFINE
*
IF A2(1,1)     = A1(1)              THEN IGNORE END-IF /* allowed
IF A2(1,1)     = A1(I)              THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A1(1)              THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A1(I)              THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A1(*)              THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A1(I -3:I+4)       THEN IGNORE END-IF /* allowed
IF A2(1,5:J)   = A1(1:I)            THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A1(1:I)            THEN IGNORE END-IF /* NOT ALLOWED(NAT0629)
IF A2(1,*)     = A1(1:K)            THEN IGNORE END-IF /* allowed
*
IF A2(1,1)     = A2(1,1)            THEN IGNORE END-IF /* allowed
IF A2(1,1)     = A2(1,I)            THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A2(1,1:8)          THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A2(I,I -3:I+4)     THEN IGNORE END-IF /* allowed
IF A2(1,1:I)   = A2(1,I+1:J)        THEN IGNORE END-IF /* allowed
IF A2(1,1:I)   = A2(1,I:I+1)        THEN IGNORE END-IF /* NOT ALLOWED(NAT0629)
IF A2(*,1)     = A2(1,*)            THEN IGNORE END-IF /* NOT ALLOWED(NAT0629)
IF A2(1,1:I)   = A1(2,1:K)          THEN IGNORE END-IF /* NOT ALLOWED(NAT0629)
*
IF A3(1,1,*)   = A2(1,*)            THEN IGNORE END-IF /* allowed
IF A3(1,1,*)   = A2(1,I -3:I+4)     THEN IGNORE END-IF /* allowed
IF A3(1,*,I:J) = A2(*,1:I+1)        THEN IGNORE END-IF /* allowed
IF A3(1,*,I:J) = A2(*,I:J)          THEN IGNORE END-IF /* allowed
END
```

When you compare two array ranges, note that the following two expressions lead to different results:

```
#ARRAY1(*) NOT EQUAL #ARRAY2(*)
NOT #ARRAY1(*) = #ARRAY2(*)
```

Example:

- **Condition A:**

  ```
  IF #ARRAY1(1:2) NOT EQUAL #ARRAY2(1:2)
  ```

  This is equivalent to:

  ```
  IF (#ARRAY1(1) NOT EQUAL #ARRAY2(1)) AND (#ARRAY1(2) NOT EQUAL #ARRAY2(2))
  ```

  Condition A is therefore true if the first occurrence of `#ARRAY1` does not equal the first occurrence of `#ARRAY2` *and* the second occurrence of `#ARRAY1` does not equal the second occurrence of `#ARRAY2`.

- **Condition B:**

  ```
  IF NOT #ARRAY1(1:2) = #ARRAY2(1:2)
  ```

  This is equivalent to:

  ```
  IF NOT (#ARRAY1(1)= #ARRAY2(1) AND #ARRAY1(2) = #ARRAY2(2))
  ```

  This in turn is equivalent to:

  ```
  IF (#ARRAY1(1) NOT EQUAL #ARRAY2(1)) OR (#ARRAY1(2) NOT EQUAL #ARRAY2(2))
  ```

  Condition B is therefore true if *either* the first occurrence of `#ARRAY1` does not equal the first occurrence of `#ARRAY2` *or* the second occurrence of `#ARRAY1` does not equal the second occurrence of `#ARRAY2`.

Arithmetic Operations with Arrays

A general rule about arithmetic operations with arrays is that the number of occurrences of the corresponding dimensions must be equal.

The following illustrates this rule:

```
#c(2:3,2:4) := #a(3:4,1:3) + #b(3:5)
```

In other words:

Array	Dimension Number	Number of Occurrences	Range
#c	2nd	2	2:3
#c	1st	3	2:4
#a	2nd	2	3:4
#a	1st	3	1:3
#b	1st	3	3:5

The operation is performed element by element.

Note: An arithmetic operation of a different number of dimensions is allowed.

For the example above, the following operations are executed:

```
#c(2,2) := #a(3,1) + #b(3)

#c(2,3) := #a(3,2) + #b(4)

#c(2,4) := #a(3,3) + #b(5)

#c(3,2) := #a(4,1) + #b(3)

#c(3,3) := #a(4,2) + #b(4)

#c(3,4) := #a(4,3) + #b(5)
```

Below is a list of examples of how array ranges may be used in the following ways in arithmetic operations (in `COMPUTE`, `ADD` or `MULTIPLY` statements). In the examples 1-4, the number of occurrences of the corresponding dimensions must be equal.

1. *range* := *range* + *range*

   The addition is performed element by element.

2. *range* := *range* * *range*

   The multiplication is performed element by element.

3. *range* := *range* + *scalar*

   The scalar is added to each element of the range.

4. *range* := *range* * *scalar*

   Each element of the range is multiplied by the scalar.

5. *scalar* := *range* + *scalar*

   Each element of the range is added to the scalar and the result is assigned to the scalar.

6. *scalar* := *range* * *scalar*

The scalar is multiplied by each element of the array and the result is assigned to the scalar.

If you use mixed source fields for the operation (`range` and `scalar`), the performed action depends on the type of the target field (`range` or `scalar`).

There is not specific input sequence, you can use `scalar` + `range` and `scalar` * `range` and also `range` + `scalar` and `range` * `scalar`.

Since intermediate results will be generated for arithmetic operations as shown in the above examples, the result of overlapping index ranges is computed element by element in an intermediate result array and finally the intermediate result array is assigned to the result field.

Example:

```
DEFINE DATA LOCAL
1 #ARRAY(I4/1:5) INIT <10,20,30,40,50>
END-DEFINE

#ARRAY(3:5) := #ARRAY(2:4) + 1

/* A temporary array for the
/* intermediate result values is
/* generated implicitly: #temp(1:3).
/* The following operations are
/* performed internally:
/* #temp(1) := #ARRAY(2) + 1
/* #temp(2) := #ARRAY(3) + 1
/* #temp(3) := #ARRAY(4) + 1
/* #ARRAY(3:5) := #temp(1:3)
/*
/* #ARRAY contains  10,20,21,31,41
```

44 Conditional Processing - 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.

Structure of IF Statement

The `IF` statement contains three components:

`IF`	In the `IF` clause, you specify the logical condition which is to be met.
`THEN`	In the `THEN` clause you specify the statement(s) to be executed if this condition is met.
`ELSE`	In the (optional) `ELSE` clause, you can specify the statement(s) to be executed if this condition is *not* met.

So, an `IF` statement takes the following general form:

```
IF condition
   THEN execute statement(s)
   ELSE execute other statement(s)
END-IF
```

Note: If you wish a certain processing to be performed only if the `IF` condition is *not* met, you can specify the clause `THEN IGNORE`. The `IGNORE` statement causes the `IF` condition to be ignored if it is met.

Example 1:

```
** Example 'IFX01': IF
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 BIRTH
  2 CITY
  2 SALARY (1:1)
END-DEFINE
*
LIMIT 7
READ MYVIEW BY CITY STARTING FROM 'C'
 IF SALARY (1) LT 40000 THEN
    WRITE NOTITLE '*****' NAME 30X 'SALARY LT 40000'
  ELSE
    DISPLAY NAME BIRTH (EM=YYYY-MM-DD) SALARY (1)
  END-IF
END-READ
END
```

The `IF` statement block in the above program causes the following conditional processing to be performed:

- `IF` the salary is less than 40000, `THEN` the `WRITE` statement is to be executed;
- otherwise (`ELSE`), that is, if the salary is 40000 or more, the `DISPLAY` statement is to be executed.

Output of Program `IFX01`:

```
        NAME            DATE      ANNUAL
                         OF       SALARY
                       BIRTH
-------------------- ---------- ----------

***** KEEN                                            SALARY LT 40000
***** FORRESTER                                       SALARY LT 40000
***** JONES                                           SALARY LT 40000
***** MELKANOFF                                       SALARY LT 40000
DAVENPORT            1948-12-25      42000
GEORGES              1949-10-26     182800
***** FULLERTON                                       SALARY LT 40000
```

Example 2:

```
** Example 'IFX03': IF
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 BONUS  (1,1)
  2 SALARY (1)
*
1 #INCOME  (N9)
1 #TEXT    (A26)
END-DEFINE
*
WRITE TITLE '-- DISTRIBUTION OF CATALOGS I AND II --' /
*
READ (3) EMPLOY-VIEW BY CITY = 'SAN FRANCISCO'
  COMPUTE #INCOME = BONUS(1,1) + SALARY(1)
  /*
 IF #INCOME > 40000
    MOVE 'CATALOGS I AND II' TO #TEXT
  ELSE
    MOVE 'CATALOG I'         TO #TEXT
  END-IF
  /*
  DISPLAY NAME 5X 'SALARY' SALARY(1) / BONUS(1,1)
  WRITE T*SALARY '-'(10) /
        16X 'INCOME:' T*SALARY #INCOME 3X #TEXT /
```

```
        16X '='(19)
  SKIP 1
END-READ
END
```

Output of Program IFX03:

```
                    -- DISTRIBUTION OF CATALOGS I AND II --
        NAME               SALARY
                           BONUS
--------------------     ----------

COLVILLE JR                   56000
                                  0
                         ----------
                INCOME:       56000   CATALOGS I AND II
                ===================

RICHMOND                       9150
                                  0
                         ----------
                INCOME:        9150   CATALOG I
                ===================

MONKTON                       13500
                                600
                         ----------
                INCOME:       14100   CATALOG I
                ===================
```

Nested IF Statements

It is possible to use various nested IF statements; for example, you can make the execution of a THEN clause dependent on another IF statement which you specify in the THEN clause.

Example:

```
** Example 'IFX02': IF (two IF statements nested)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 SALARY (1:1)
  2 BIRTH
  2 PERSONNEL-ID
1 MYVIEW2 VIEW OF VEHICLES
  2 PERSONNEL-ID
```

```
  2 MAKE
*
1 #BIRTH   (D)
END-DEFINE
*
MOVE EDITED '19450101' TO #BIRTH (EM=YYYYMMDD)
*
LIMIT 20
FND1. FIND MYVIEW WITH CITY = 'BOSTON'
                  SORTED BY NAME
 IF SALARY (1) LESS THAN 20000
    WRITE NOTITLE '*****' NAME 30X 'SALARY LT 20000'
  ELSE
    IF BIRTH GT #BIRTH
      FIND MYVIEW2 WITH PERSONNEL-ID = PERSONNEL-ID (FND1.)
        DISPLAY (IS=ON) NAME BIRTH (EM=YYYY-MM-DD)
                        SALARY (1) MAKE (AL=8 IS=OFF)
      END-FIND
    END-IF
  END-IF
  SKIP 1
END-FIND
END
```

Output of Program `IFX02`:

```
        NAME            DATE      ANNUAL     MAKE
                         OF       SALARY
                       BIRTH
-------------------- ---------- ---------- --------

***** COHEN                                            SALARY LT 20000

CREMER               1972-12-14      20000 FORD

***** FLEMING                                          SALARY LT 20000

PERREAULT            1950-05-12      30500 CHRYSLER

***** SHAW                                             SALARY LT 20000

STANWOOD             1946-09-08      31000 CHRYSLER
                                           FORD
```

45 Logical Condition Criteria

This chapter describes purpose and use of logical condition criteria that can be used in the statements FIND, READ, HISTOGRAM, ACCEPT/REJECT, IF, DECIDE FOR, REPEAT.

Introduction

The basic criterion is a **relational expression**. Multiple relational expressions may be combined with logical operators (AND, OR) to form complex criteria.

Arithmetic expressions may also be used to form a relational expression.

Logical condition criteria can be used in the following statements:

Statement	Usage
FIND	A WHERE clause containing logical condition criteria may be used to indicate criteria in addition to the basic selection criteria as specified in the WITH clause. The logical condition criteria specified with the WHERE clause are evaluated after the record has been selected and read. In a WITH clause, "basic search criteria" (as described with the FIND statement) are used, but not logical condition criteria.
READ	A WHERE clause containing logical condition criteria may be used to specify whether a record that has just been read is to be processed. The logical condition criteria are evaluated after the record has been read.
HISTOGRAM	A WHERE clause containing logical condition criteria may be used to specify whether the value that has just been read is to be processed. The logical condition criteria are evaluated after the value has been read.
ACCEPT/REJECT	An IF clause may be used with an ACCEPT or REJECT statement to specify logical condition criteria in addition to that specified when the record was selected/read with a FIND, READ, or HISTOGRAM statement. The logical condition criteria are evaluated after the record has been read and after record processing has started.
IF	Logical condition criteria are used to control statement execution.
DECIDE FOR	Logical condition criteria are used to control statement execution.
REPEAT	The UNTIL or WHILE clause of a REPEAT statement contain logical condition criteria which determine when a processing loop is to be terminated.

Relational Expression

Syntax:

```
             EQ
             =
             EQUAL
             EQUAL TO
             NE
             ^=
             <>
             NOT =
             NOT EQ
             NOTEQUAL
             NOT EQUAL
             NOT EQUAL TO
             LT
             LESS THAN
operand1  {  <                }  operand2
             GE
             GREATER EQUAL
             >=
             NOT <
             NOT LT
             GT
             GREATER THAN
             >
             LE
             LESS EQUAL
             <=
             NOT >
             NOT GT
```

Operand Definition Table:

Operand	Possible Structure						Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A		N	E	A	U	N	P	I	F	B	D	T	L		G	O	yes	yes
operand2	C	S	A		N	E	A	U	N	P	I	F	B	D	T	L		G	O	yes	no

For an explanation of the Operand Definition Table shown above, see *Syntax Symbols and Operand Definition Tables* in the *Statements* documentation.

In the “Possible Structure” column of the table above, “E” stands for arithmetic expressions; that is, any arithmetic expression may be specified as an operand within the relational expression. For further information on arithmetic expressions, see *arithmetic-expression* in the COMPUTE statement description.

Explanation of the comparison operators:

Comparison Operator	Explanation
EQ = EQUAL EQUAL TO	equal to
NE ^= <> NOT = NOT EQ NOTEQUAL NOT EQUAL NOT EQUAL TO	not equal to
LT LESS THAN <	less than
GE GREATER EQUAL >=	greater than or equal to
NOT < NOT LT	not less than
GT GREATER THAN >	greater than
LE LESS EQUAL <=	less than or equal to
NOT > NOT GT	not greater than

Examples of Relational Expressions:

```
IF NAME = 'SMITH'
IF LEAVE-DUE GT 40
IF NAME = #NAME
```

For information on comparing arrays in a relational expression, see ***Processing of Arrays***.

Note: If a floating-point operand is used, comparison is performed in floating point. **Floating-point numbers** as such have only a limited precision; therefore, rounding/truncation errors cannot be precluded when numbers are converted to/from floating-point representation.

Arithmetic Expressions in Logical Conditions

The following example shows how arithmetic expressions can be used in logical conditions:

```
IF #A + 3 GT #B - 5 AND #C * 3 LE #A + #B
```

Handles in Logical Conditions

If the operands in a relation expression are handles, only `EQUAL` and `NOT EQUAL` operators may be used.

SUBSTRING Option in Relational Expression

Syntax:

```
                                                 =
                                                 EQ
                                                 EQUAL [TO]
                                                 <>
                                                 NE
                                                 NOT =
                                                 NOT EQ
                                                 NOT EQUAL
                                                 NOT EQUAL TO
{ SUBSTRING                                 }    <               { operand2                        }
{ (operand1,operand3,operand4)              }    LT              { SUBSTRING                       }
{ operand1                                  }    LESS THAN       { (operand2,operand5,operand6)    }
                                                 <=
                                                 LE
                                                 LESS EQUAL
                                                 >
                                                 GT
                                                 GREATER THAN
                                                 >=
                                                 GE
                                                 GREATER EQUAL
```

Operand Definition Table:

Operand	Possible Structure						Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A		N		A	U					B							yes	yes
operand2	C	S	A		N		A	U					B							yes	no
operand3	C	S							N	P	I		B							yes	no
operand4	C	S							N	P	I									yes	no
operand5	C	S							N	P	I									yes	no
operand6	C	S							N	P	I									yes	no

With the `SUBSTRING` option, you can compare a *part* of an alphanumeric, a binary or a Unicode field. After the field name (*operand1*) you specify first the starting position (*operand3*) and then the **length** (*operand4*) of the field portion to be compared.

Also, you can compare a field value with part of another field value. After the field name (*operand2*) you specify first the starting position (*operand5*) and then the length (*operand6*) of the field portion *operand1* is to be compared with.

You can also combine both forms, that is, you can specify a `SUBSTRING` for both *operand1* and *operand2*.

Examples:

The following expression compares the 5th to 12th position inclusive of the value in field `#A` with the value of field `#B`:

```
SUBSTRING(#A,5,8) = #B
```

- where 5 is the starting position and 8 is the length.

The following expression compares the value of field `#A` with the 3rd to 6th position inclusive of the value in field `#B`:

```
#A = SUBSTRING(#B,3,4)
```

Note: If you omit *operand3*/*operand5*, the starting position is assumed to be 1. If you omit *operand4*/*operand6*, the length is assumed to be from the starting position to the end of the field.

Extended Relational Expression

Syntax:

```
operand1 { =            }          operand2
         { EQ           }
         { EQUAL [TO]   }

         {         { OR {   =          }    operand3 }    ...   }
         {         {    {   EQ         }             }          }
         {         {    {   EQUAL [TO] }             }          }
         {                                                      }
         { THRU operand4 [BUT NOT operand5 [THRU operand6]]     }
```

Operand Definition Table:

Operand	Possible Structure						Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	yes	no
operand2	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	yes	no
operand3	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	yes	no
operand4	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	yes	no
operand5	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	yes	no
operand6	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	yes	no

* Mathematical functions and system variables are permitted. Break functions are not permitted.

operand3 can also be specified using a `MASK` or `SCAN` option; that is, it can be specified as:

```
MASK (mask-definition) [operand]
MASK operand
SCAN operand
```

For details on these options, see the sections ***MASK Option*** and ***SCAN Option***.

Examples:

```
IF #A = 2 OR = 4 OR = 7
IF #A = 5 THRU 11 BUT NOT 7 THRU 8
```

Evaluation of a Logical Variable

Syntax:

```
operand1
```

This option is used in conjunction with a logical variable (format L). A logical variable may take the value `TRUE` or `FALSE`. As *operand1* you specify the name of the logical variable to be used.

Operand Definition Table:

Operand	Possible Structure						Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A													L				no	no

Example of Logical Variable:

```
** Example 'LOGICX05': Logical variable in logical condition
************************************************************************
DEFINE DATA LOCAL
1 #SWITCH (L)  INIT <true>
1 #INDEX  (I1)
END-DEFINE
*
FOR #INDEX 1 5
  WRITE NOTITLE #SWITCH (EM=FALSE/TRUE) 5X 'INDEX =' #INDEX
  WRITE NOTITLE #SWITCH (EM=OFF/ON) 7X 'INDEX =' #INDEX
  IF #SWITCH
    MOVE FALSE TO #SWITCH
  ELSE
    MOVE TRUE TO #SWITCH
  END-IF
  /*
  SKIP 1
END-FOR
END
```

Output of Program `LOGICX05`:

```
TRUE     INDEX =    1
ON       INDEX =    1

FALSE    INDEX =    2
OFF      INDEX =    2

TRUE     INDEX =    3
ON       INDEX =    3

FALSE    INDEX =    4
OFF      INDEX =    4

TRUE     INDEX =    5
ON       INDEX =    5
```

Fields Used within Logical Condition Criteria

Database fields and user-defined variables may be used to construct logical condition criteria. A database field which is a multiple-value field or is contained in a periodic group can also be used. If a range of values for a multiple-value field or a range of occurrences for a periodic group is specified, the condition is true if the search value is found in any value/occurrence within the specified range.

Each value used must be compatible with the field used on the opposite side of the expression. Decimal notation may be specified only for values used with numeric fields, and the number of decimal positions of the value must agree with the number of decimal positions defined for the field.

If the operands are not of the same format, the second operand is converted to the format of the first operand.

Note: A numeric constant without decimal point notation is stored with format I for the values `-2147483648` to `+2147483647`, see ***Numeric Constants***. Consequently the comparison with such an integer constant as *`operand1`* is performed by converting *`operand2`* to a integer value. This means that the digits after the decimal point of *`operand2`* are not considered due to truncation.

Example:

```
 IF 0 = 0.5    /* is true because 0.5 (operand2) is converted to 0 (format I of ↩
operand1)
 IF 0.0 = 0.5  /* is false
 IF 0.5 = 0    /* is false
 IF 0.5 = 0.0  /* is false ↩
```

The following table shows which operand formats can be used together in a logical condition:

operand1	*operand2*												
	A	**U**	**B***n* (*n*=<4)	**B***n* (*n*>=5)	**D**	**T**	**I**	**F**	**L**	**N**	**P**	**GH**	**OH**
A	Y	Y	Y	Y									
U	Y	Y	[2]	[2]									
B*n* (*n*=<4)	Y	Y	Y	Y	Y	Y	Y	Y		Y	Y		
B*n* (*n*>=5)	Y	Y	Y	Y									
D			Y		Y	Y	Y	Y		Y	Y		
T			Y		Y	Y	Y	Y		Y	Y		
I			Y		Y	Y	Y	Y		Y	Y		
F			Y		Y	Y	Y	Y		Y	Y		
L													
N			Y		Y	Y	Y	Y		Y	Y		
P			Y		Y	Y	Y	Y		Y	Y		
GH [1]												Y	
OH [1]													Y

Notes:

1. [1] where GH = GUI handle, OH = object handle.
2. [2] The binary value will be assumed to contain Unicode code points, and the comparison is performed as for a comparison of two Unicode values. The length of the binary field must be even.

If two values are compared as alphanumeric values, the shorter value is assumed to be extended with trailing blanks in order to get the same length as the longer value.

If two values are compared as binary values, the shorter value is assumed to be extended with leading binary zeroes in order to get the same length as the longer value.

If two values are compared as Unicode values, trailing blanks are removed from both values before the ICU collation algorithm is used to compare the two resulting values. See also *Logical Condition Criteria* in the *Unicode and Code Page Support* documentation.

Comparison Examples:

```
A1(A1) := 'A'
A5(A5) := 'A    '
B1(B1) := H'FF'
B5(B5) := H'00000000FF'
U1(U1) := UH'00E4'
U2(U2) := UH'00610308'
IF A1 = A5 THEN ...                /* TRUE
IF B1 = B5 THEN ...                /* TRUE
IF U1 = U2 THEN ...                /* TRUE ↩
```

If an array is compared with a scalar value, each element of the array will be compared with the scalar value. The condition will be true if at least one of the array elements meets the condition (`OR` operation).

If an array is compared with an array, each element in the array is compared with the corresponding element of the other array. The result is true only if all element comparisons meet the condition (`AND` operation).

See also ***Processing of Arrays***.

Note: An Adabas phonetic descriptor cannot be used within a logical condition.

Examples of Logical Condition Criteria:

```
FIND EMPLOYEES-VIEW WITH CITY = 'BOSTON' WHERE SEX = 'M'
READ EMPLOYEES-VIEW BY NAME WHERE SEX = 'M'
ACCEPT IF LEAVE-DUE GT 45
IF #A GT #B THEN COMPUTE #C = #A + #B
REPEAT UNTIL #X = 500
```

Logical Operators in Complex Logical Expressions

Logical condition criteria may be combined using the Boolean operators `AND`, `OR`, and `NOT`. Parentheses may also be used to indicate logical grouping.

The operators are evaluated in the following order:

Priority	Operator	Meaning
1	( )	Parentheses
2	NOT	Negation
3	AND	AND operation
4	OR	OR operation

The following *logical-condition-criteria* may be combined by logical operators to form a complex *logical-expression*:

- **Relational expressions**
- **Extended relational expressions**
- MASK **option**
- SCAN **option**
- BREAK **option**

The syntax for a *logical-expression* is as follows:

```
[NOT] { logical-condition-criterion } [ { OR  } logical-expression ] ...
      { (logical-expression)        }   { AND }
```

Examples of Logical Expressions:

```
FIND STAFF-VIEW WITH CITY = 'TOKYO'
     WHERE BIRTH GT 19610101 AND SEX = 'F'
IF NOT (#CITY = 'A' THRU 'E')
```

For information on comparing arrays in a logical expression, see ***Processing of Arrays***.

Note: If multiple logical-condition-criteria are connected with AND, the evaluation terminates as soon as the first of these criteria is not true. If multiple logical-condition-criteria are connected with OR, the evaluation terminates as soon as the first of these criteria is true.

BREAK Option - Compare Current Value with Value of Previous Loop Pass

The BREAK option allows the current value or a portion of a value of a field to be compared with the value contained in the same field in the previous pass through the processing loop.

Syntax:

`BREAK` [OF] *`operand1`* [/*n*/]

Operand Definition Table:

Operand	Possible Structure						Possible Formats													Referencing Permitted	Dynamic Definition
operand1		S					A	U	N	P	I	F	B	D	T	L				yes	no

Syntax Element Description:

<table>
<tr><td>operand1</td><td>Specifies the control field which is to be checked. A specific occurrence of an array can also be used as a control field.</td></tr>
<tr><td>/n/</td><td>The notation /n/ may be used to indicate that only the first n positions (counting from left to right) of the control field are to be checked for a change in value. This notation can only be used with operands of format A, B, N, or P.

The result of the <code>BREAK</code> operation is true when a change in the specified positions of the field occurs. The result of the <code>BREAK</code> operation is not true if an <code>AT END OF DATA</code> condition occurs.

Example:

In this example, a check is made for a different value in the first position of the field <code>FIRST-NAME</code>.

<code>BREAK FIRST-NAME /1/</code>

Natural system functions (which are available with the <code>AT BREAK</code> statement) are not available with this option.</td></tr>
</table>

Example of BREAK Option:

```
** Example 'LOGICX03': BREAK option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BIRTH
*
1 #BIRTH (A8)
END-DEFINE
*
LIMIT 10
READ EMPLOY-VIEW BY BIRTH
  MOVE EDITED BIRTH (EM=YYYYMMDD) TO #BIRTH
  /*
  IF BREAK OF #BIRTH /6/
    NEWPAGE IF LESS THAN 5 LINES LEFT
    WRITE / '-' (50) /
```

```
  END-IF
  /*
  DISPLAY NOTITLE BIRTH (EM=YYYY-MM-DD) NAME FIRST-NAME
END-READ
END
```

Output of Program LOGICX03:

```
  DATE            NAME                 FIRST-NAME
   OF
 BIRTH
---------- -------------------- --------------------

1940-01-01 GARRET               WILLIAM
1940-01-09 TAILOR               ROBERT
1940-01-09 PIETSCH              VENUS
1940-01-31 LYTTLETON            BETTY

--------------------------------------------------

1940-02-02 WINTRICH             MARIA
1940-02-13 KUNEY                MARY
1940-02-14 KOLENCE              MARSHA
1940-02-24 DILWORTH             TOM

--------------------------------------------------

1940-03-03 DEKKER               SYLVIA
1940-03-06 STEFFERUD            BILL
```

IS Option - Check whether Content of Alphanumeric or Unicode Field can be Converted

Syntax:

operand1 IS (*format*)

This option is used to check whether the content of an alphanumeric or Unicode field (*operand1*) can be converted to a specific other format.

Operand Definition Table:

Operand	Possible Structure						Possible Formats		Referencing Permitted	Dynamic Definition
`operand1`	C	S	A		N		A	U	yes	no

The `format` for which the check is performed can be:

N*ll.ll*	Numeric with length *ll.ll*.
F*ll*	Floating point with length *ll*.
D	Date. The following date formats are possible: *dd-mm-yy*, *dd-mm-yyyy*, *ddmmyyyy* (*dd* = day, *mm* = month, *yy* or *yyyy* = year). The sequence of the day, month and year components as well as the characters between the components are determined by the profile parameter `DTFORM` (which is described in the *Parameter Reference*).
T	Time (according to the default time display format).
P*ll.ll*	Packed numeric with length *ll.ll*.
I*ll*	Integer with length *ll*.

When the check is performed, leading and trailing blanks in `operand1` will be ignored.

The `IS` option may, for example, be used to check the content of a field before the mathematical function `VAL` (extract numeric value from an alphanumeric field) is used to ensure that it will not result in a runtime error.

Note: The `IS` option cannot be used to check if the value of an alphanumeric field is in the specified "format", but if it can be *converted* to that "format". To check if a value is in a specific format, you can use the `MASK` option. For further information, see *MASK Option Compared with IS Option* and *Checking Packed or Unpacked Numeric Data*.

Example of IS Option:

```
** Example 'LOGICX04': IS option as format/length check
************************************************************************
DEFINE DATA LOCAL
1 #FIELDA (A10)          /* INPUT FIELD TO BE CHECKED
1 #FIELDB (N5)           /* RECEIVING FIELD OF VAL FUNCTION
1 #DATE   (A10)          /* INPUT FIELD FOR DATE
END-DEFINE
*
INPUT #DATE #FIELDA
IF #DATE IS(D)
  IF #FIELDA IS (N5)
    COMPUTE #FIELDB = VAL(#FIELDA)
    WRITE NOTITLE 'VAL FUNCTION OK' //  '=' #FIELDA '=' #FIELDB
  ELSE
    REINPUT 'FIELD DOES NOT FIT INTO N5 FORMAT'
            MARK *#FIELDA
  END-IF
ELSE
  REINPUT 'INPUT IS NOT IN DATE FORMAT (YY-MM-DD) '
```

```
          MARK *#DATE
END-IF
*
END
```

Output of Program `LOGICX04`:

```
#DATE 150487    #FIELDA










INPUT IS NOT IN DATE FORMAT (YY-MM-DD)
```

MASK Option - Check Selected Positions of a Field for Specific Content

With the `MASK` option, you can check selected positions of a field for specific content.

The following topics are covered below:

Constant Mask

Syntax:

```
           {  =           }
           {  EQ          }
operand1   {  EQUAL TO    }  MASK (mask-definition) [operand2]
           {  NE          }
           {  NOT EQUAL   }
```

Operand Definition Table:

Operand	Possible Structure						Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A		N		A	U	N	P										yes	no
operand2	C	S					A	U	N	P			B							yes	no

operand2 can only be used if the *mask-definition* contains at least one `X`. *operand1* and *operand2* must be format-compatible:

- If *operand1* is of format A, *operand2* must be of format A, B, N or U.
- If *operand1* is of format U, *operand2* must be of format A, B, N or U.
- If *operand1* is of format N or P, *operand2* must be of format N or P.

An `X` in the *mask-definition* selects the corresponding positions of the content of *operand1* and *operand2* for comparison.

Variable Mask

Apart from a constant *mask-definition* (see above), you may also specify a variable mask definition.

Syntax:

```
            ⎧ =         ⎫
            ⎪ EQ        ⎪
operand1    ⎨ EQUAL TO  ⎬ MASK operand2
            ⎪ NE        ⎪
            ⎩ NOT EQUAL ⎭
```

Operand Definition Table:

Operand	Possible Structure						Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A		N		A	U	N	P										yes	no
operand2		S					A	U												yes	no

The content of *operand2* will be taken as the mask definition. Trailing blanks in *operand2* will be ignored.

- If *operand1* is of format A, N or P, *operand2* must be of format A.
- If *operand1* is of format U, *operand2* must be of format U.

Characters in a Mask

The following characters may be used within a mask definition (the mask definition is contained in *mask-definition* for a constant mask and *operand2* for a variable mask):

Character	Meaning
. or ? or _	A period, question mark or underscore indicates a single position that is not to be checked.
* or %	An asterisk or percent mark is used to indicate any number of positions not to be checked.
/	A slash (/) is used to check if a value ends with a specific character (or string of characters). For example, the following condition will be true if there is either an E in the last position of the field, or the last E in the field is followed by nothing but blanks: `IF #FIELD = MASK (*'E'/)`
A	The position is to be checked for an alphabetical character (upper or lower case).
'*c*'	One or more positions are to be checked for the characters bounded by apostrophes. The characters to be checked for are not dependent on the TQMARK parameter: a quotation mark (") is checked for a quotation mark ("). If *operand1* is in Unicode format, '*c*' must contain Unicode characters.
C	The position is to be checked for an alphabetical character (upper or lower case), a numeric character, or a blank.
DD	The two positions are to be checked for a valid day notation (01 - 31; dependent on the values of MM and YY/YYYY, if specified; see also ***Checking Dates***).
H	The position is to be checked for hexadecimal content (A - F, 0 - 9).
JJJ	The positions are to be checked for a valid Julian Day; that is, the day number in the year (001-366, dependent on the value of YY/YYYY, if specified. See also ***Checking Dates***.)
L	The position is to be checked for a lower-case alphabetical character (a - z).
MM	The positions are to be checked for a valid month (01 - 12); see also ***Checking Dates***.
N	The position is to be checked for a numeric digit.
n...	One (or more) positions are to be checked for a numeric value in the range 0 - *n*.
*n*1-*n*2 or *n*1:*n*2	The positions are checked for a numeric value in the range *n*1-*n*2. *n*1 and *n*2 must be of the same length.
P	The position is to be checked for a displayable character (U, L, N or S).
S	The position is to be checked for special characters. See also *Support of Different Character Sets with NATCONV.INI* in the *Operations* documentation.
U	The position is to be checked for an upper-case alphabetical character (A - Z).
X	The position is to be checked against the equivalent position in the value (*operand2*) following the mask-definition. X is not allowed in a variable mask definition, as it makes no sense.

Character	Meaning
YY	The two positions are to be checked for a valid year (00 - 99). See also *Checking Dates*.
YYYY	The four positions are checked for a valid year (0000 - 2699). Use the COMPOPT option MASKCME=ON to restrict the range of valid years to 1582 - 2699; see also *Checking Dates*. If the profile parameter MAXYEAR is set to 9999, the upper year limit is 9999.
Z	The position is to be checked for a character whose left half-byte is hexadecimal 3 or 7, and whose right half-byte is hexadecimal 0 - 9. This may be used to correctly check for numeric digits in negative numbers. With N (which indicates a position to be checked for a numeric digit), a check for numeric digits in negative numbers leads to incorrect results, because the sign of the number is stored in the last digit of the number, causing that digit to be hexadecimal represented as non-numeric. Within a mask, use only one Z for each sequence of numeric digits that is checked.

Mask Length

The length of the mask determines how many positions are to be checked.

Example:

```
DEFINE DATA LOCAL
1 #CODE (A15)
END-DEFINE
...
IF #CODE = MASK (NN'ABC'....NN)
...
```

In the above example, the first two positions of #CODE are to be checked for numeric content. The three following positions are checked for the contents ABC. The next four positions are not to be checked. Positions ten and eleven are to be checked for numeric content. Positions twelve to fifteen are not to be checked.

Checking Dates

Only one date may be checked within a given mask. When the same date component (JJJ, DD, MM, YY or YYYY) is specified more than once in the mask, only the value of the last occurrence is checked for consistency with other date components.

When dates are checked for a day (DD) and no month (MM) is specified in the mask, the current month will be assumed.

When dates are checked for a day (DD) or a Julian day (JJJ) and no year (YY or YYYY) is specified in the mask, the current year will be assumed.

When dates are checked for a 2-digit year (`YY`), the current century will be assumed if no Sliding or Fixed Window is set. For more details about Sliding or Fixed Windows, refer to profile parameter `YSLW` in the *Parameter Reference*.

Example 1:

```
MOVE 1131 TO #DATE (N4)
IF #DATE = MASK (MMDD)
```

In this example, month and day are checked for validity. The value for month (11) will be considered valid, whereas the value for day (31) will be invalid since the 11th month has only 30 days.

Example 2:

```
IF #DATE(A8) = MASK (MM'/'DD'/'YY)
```

In this example, the content of the field `#DATE` is be checked for a valid date with the format `MM/DD/YY` (month/day/year).

Example 3:

```
IF #DATE (A8) = MASK (1950-2020MMDD)
```

In this example, the content of field `#DATE` is checked for a four-digit number in the range 1950 to 2020 followed by a valid month and day in the current year.

Note: Although apparent, the above mask does not allow to check for a valid date in the years 1950 through 2020, because the numeric value range 1950-2020 is checked independent of the validation of month and day. The check will deliver the intended results except for February, 29, where the result depends on whether the current year is a leap year or not. To check for a specific year range in addition to the date validation, code one check for the date validation and another for the range validation:

```
IF #DATE (A8) = MASK (YYYYMMDD) AND #DATE = MASK (1950-2020)
```

Example 4:

```
IF #DATE (A4) = MASK (19-20YY)
```

In this example, the content of field `#DATE` is checked for a two-digit number in the range 19 to 20 followed by a valid two-digit year (00 through 99). The century is supplied by Natural as described above.

Note: Although apparent, the above mask does not allow to check for a valid year in the range 1900 through 2099, because the numeric value range 19-20 is checked independent of the year validation. To check for year ranges, code one check for the date validation and another for the range validation:

```
IF #DATE (A10) = MASK (YYYY'-'MM'-'DD) AND #DATE = MASK (19-20)
```

Checking Against the Content of Constants or Variables

If the value for the mask check is to be taken from either a constant or a variable, this value (*operand2*) must be specified immediately following the *mask-definition*.

operand2 must be at least as long as the mask.

In the mask, you indicate each position to be checked with an X, and each position not to be checked with a period (.) or a question mark (?) or an underscore (_).

Example:

```
DEFINE DATA LOCAL
1 #NAME (A15)
END-DEFINE
...
IF #NAME = MASK (..XX) 'ABCD'
...
```

In the above example, it is checked whether the field #NAME contains CD in the third and fourth positions. Positions one and two are not checked.

The length of the mask determines how many positions are to be checked. The mask is left-justified against any field or constant used in the mask operation. The format of the field (or constant) on the right side of the expression must be the same as the format of the field on the left side of the expression.

If the field to be checked (*operand1*) is of format A, any constant used (*operand2*) must be enclosed in apostrophes. If the field is numeric, the value used must be a numeric constant or the content of a numeric database field or user-defined variable.

In either case, any characters/digits within the value specified whose positions do not match the X indicator within the mask are ignored.

The result of the MASK operation is true when the indicated positions in both values are identical.

Example:

```
** Example 'LOGICX01': MASK option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
END-DEFINE
*
HISTOGRAM EMPLOY-VIEW CITY
  IF CITY =
```

```
MASK (....XX) '....NN'


    DISPLAY NOTITLE CITY *NUMBER
  END-IF
END-HISTOGRAM
*
END
```

In the above example, the record will be accepted if the fifth and sixth positions of the field `CITY` each contain the character `N`.

Range Checks

When performing range checks, the number of positions verified in the supplied variable is defined by the precision of the value supplied in the mask specification. For example, a mask of `(...193...)` will verify positions 4 to 6 for a three-digit number in the range 000 to 193.

Additional Examples of Mask Definitions:

- In this example, each character of `#NAME` is checked for an alphabetical character:

  ```
  IF #NAME (A10) = MASK (AAAAAAAAAA)
  ```

- In this example, positions 4 to 6 of `#NUMBER` are checked for a numeric value:

  ```
  IF #NUMBER (A6) = MASK (...NNN)
  ```

- In this example, positions 4 to 6 of `#VALUE` are to be checked for the value `123`:

  ```
  IF #VALUE(A10) = MASK (...'123')
  ```

- This example will check if `#LICENSE` contains a license number which begins with `NY-` and whose last five characters are identical to the last five positions of `#VALUE`:

  ```
  DEFINE DATA LOCAL
  1 #VALUE(A8)
  1 #LICENSE(A8)
  END-DEFINE
  INPUT 'ENTER KNOWN POSITIONS OF LICENSE PLATE:' #VALUE
  IF #LICENSE = MASK ('NY-'XXXXX) #VALUE
  ```

- The following condition would be met by any value which contains `NAT` and `AL` no matter which and how many other characters are between `NAT` and `AL` (this would include the values `NATURAL` and `NATIONALITY` as well as `NATAL`):

```
MASK('NAT'*'AL')
```

Checking Packed or Unpacked Numeric Data

Legacy applications often have packed or unpacked numeric fields redefined with alphanumeric or binary fields. Such redefinitions are not recommended, because using the packed or unpacked field in an assignment or computation may lead to errors or unpredictable results. To validate the contents of such a redefined field before the field is used, enter an `N` for each digit of the field (counting the digits before and after the decimal point), minus one, followed by a `Z` (see ***Characters in a Mask***).

Examples :

```
IF #P1 (P1) = MASK (Z)
IF #N4 (N4) = MASK (NNNZ)
IF #P5 (P5) = MASK (NNNNZ)
IF #P32 (P3.2) = MASK (NNNNZ)
```

For further information about checking field contents, see ***MASK Option Compared with IS Option***.

MASK Option Compared with IS Option

This section points out the difference between `MASK` option and `IS` option and contains a sample program to illustrate the difference.

The `IS` option can be used to check whether the content of an alphanumeric or Unicode field can be converted to a specific other format, but it cannot be used to check if the value of an alphanumeric field is in the specified format.

The `MASK` option can be used to validate the contents of a redefined packed or unpacked numeric variable.

Example Illustrating the Difference:

```
** Example 'LOGICX09': MASK versus IS option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 #A2   (A2)
1 REDEFINE #A2
  2 #N2 (N2)
1 REDEFINE #A2
  2 #P3 (P3)
1 #CONV-N2 (N2)
1 #CONV-P3 (P3)
END-DEFINE
```

```
*
#A2 := '12'
WRITE NOTITLE   'Assignment #A2 := "12" results in:'
PERFORM SUBTEST
#A2 := '-1'
WRITE NOTITLE / 'Assignment #A2 := "-1" results in:'
PERFORM SUBTEST
#N2 :=  12
WRITE NOTITLE / 'Assignment #N2 :=  12  results in:'
PERFORM SUBTEST
#N2 :=  -1
WRITE NOTITLE / 'Assignment #N2 :=  -1  results in:'
PERFORM SUBTEST
#P3 :=  12
WRITE NOTITLE / 'Assignment #P3 :=  12  results in:'
PERFORM SUBTEST
#P3 :=  -1
WRITE NOTITLE / 'Assignment #P3 :=  -1  results in:'
PERFORM SUBTEST
*                                                                              ↩

DEFINE SUBROUTINE SUBTEST
IF #A2 IS (N2) THEN
  #CONV-N2 := VAL(#A2)
  WRITE NOTITLE 12T '#A2 can be converted to' #CONV-N2 '(N2)'
END-IF
IF #A2 IS (P3) THEN
  #CONV-P3 := VAL(#A2)
  WRITE NOTITLE 12T '#A2 can be converted to' #CONV-P3 '(P3)'
END-IF
IF #N2 = MASK(NZ) THEN
  WRITE NOTITLE 12T '#N2 contains the valid unpacked number' #N2
END-IF
IF #P3 = MASK(NNZ) THEN
  WRITE NOTITLE 12T '#P3 contains the valid packed number' #P3
END-IF
END-SUBROUTINE
*
END                                                               ↩
```

Output of Program `LOGICX09`:

```
Assignment #A2 := '12' results in:
           #A2 can be converted to  12 (N2)
           #A2 can be converted to   12 (P3)
           #N2 contains the valid unpacked number  12

Assignment #A2 := '-1' results in:
           #A2 can be converted to  -1 (N2)
           #A2 can be converted to   -1 (P3)

Assignment #N2 :=  12  results in:
```

```
           #A2 can be converted to  12 (N2)
           #A2 can be converted to   12 (P3)
           #N2 contains the valid unpacked number  12

Assignment #N2 :=  -1  results in:
           #N2 contains the valid unpacked number  -1

Assignment #P3 :=  12  results in:
           #P3 contains the valid packed number   12

Assignment #P3 :=  -1  results in:
           #P3 contains the valid packed number   -1
```

MODIFIED Option - Check whether Field Content has been Modified

Syntax:

```
operand1 [NOT] MODIFIED
```

This option is used to determine whether the content of a field has been modified during the execution of an `INPUT` or `PROCESS PAGE` statement. As a precondition, a control variable must have been assigned using the parameter `CV`.

Operand Definition Table:

Operand	Possible Structure						Possible Formats													Referencing Permitted	Dynamic Definition
operand1		S	A														C			no	no

Attribute control variables referenced in an `INPUT` or `PROCESS PAGE` statement are always assigned the status “not modified” when the map is transmitted to the terminal.

Whenever the content of a field referencing an attribute control variable is modified, the attribute control variable has been assigned the status “modified”. When multiple fields reference the same attribute control variable, the variable is marked “modified” if any of these fields is modified.

If *operand1* is an array, the result will be true if at least one of the array elements has been assigned the status “modified” (`OR` operation).

Example of MODIFIED Option:

```
** Example 'LOGICX06': MODIFIED option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 #ATTR (C)
1 #A    (A1)
1 #B    (A1)
END-DEFINE
*
MOVE (AD=I) TO #ATTR
*
INPUT (CV=#ATTR) #A #B
IF #ATTR NOT MODIFIED
  WRITE NOTITLE 'FIELD #A OR #B HAS NOT BEEN MODIFIED'
END-IF
*
IF #ATTR MODIFIED
  WRITE NOTITLE 'FIELD #A OR #B HAS BEEN MODIFIED'
END-IF
*
END
```

Output of Program `LOGICX06`:

```

#A   #B
```

After entering any value and pressing ENTER, the following output is displayed:

```
FIELD #A OR #B HAS BEEN MODIFIED
```

SCAN Option - Scan for a Value within a Field

Syntax:

```
          { =         }
          { EQ        }        { operand2   }
operand1  { EQUAL TO  }  SCAN  { (operand2) }
          { NE        }
          { NOT EQUAL }
```

Operand Definition Table:

Operand	Possible Structure						Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A		N		A	U	N	P										yes	no
operand2	C	S					A	U					B*							yes	no

* `operand2` may only be binary if `operand1` is of format A or U. If `operand1` is of format U and `operand2` is of format B, then the length of `operand2` must be even.

The `SCAN` option is used to scan for a specific value within a field.

The characters used in the `SCAN` option (`operand2`) may be specified as an alphanumeric or Unicode constant (a character string bounded by apostrophes) or the contents of an alphanumeric or Unicode database field or user-defined variable.

Caution: Trailing blanks are automatically eliminated from `operand1` and `operand2`. Therefore, the `SCAN` option cannot be used to scan for values containing trailing blanks. `operand1` and `operand2` may contain leading or embedded blanks. If `operand2` consists of blanks only, scanning will be assumed to be successful, regardless of the value of `operand1`; confer `EXAMINE FULL` statement if trailing blanks are not to be ignored in the scan operation.

The field to be scanned (`operand1`) may be of format A, N, P or U. The `SCAN` operation may be specified with the equal (`EQ`) or not equal (`NE`) operators.

The length of the character string for the `SCAN` operation should be less than the length of the field to be scanned. If the length of the character string specified is identical to the length of the field to be scanned, then an `EQUAL` operator should be used instead of `SCAN`.

Example of SCAN Option:

```
** Example 'LOGICX02': SCAN option in logical condition
************************************************************************
DEFINE DATA
LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
*
1 #VALUE   (A4)
1 #COMMENT (A10)
END-DEFINE
*
INPUT 'ENTER SCAN VALUE:' #VALUE
LIMIT 15
*
HISTOGRAM EMPLOY-VIEW FOR NAME
  RESET #COMMENT
  IF NAME = SCAN #VALUE
    MOVE 'MATCH' TO #COMMENT
  END-IF
```

```
  DISPLAY NOTITLE NAME *NUMBER #COMMENT
END-HISTOGRAM
*
END
```

Output of Program LOGICX02:

```
ENTER SCAN VALUE:  ↩
```

A scan for example for LL delivers three matches in 15 names:

```
        NAME          NMBR      #COMMENT
-------------------- --------- ----------

ABELLAN                      1 MATCH
ACHIESON                     1
ADAM                         1
ADKINSON                     8
AECKERLE                     1
AFANASSIEV                   2
AHL                          1
AKROYD                       1
ALEMAN                       1
ALESTIA                      1
ALEXANDER                    5
ALLEGRE                      1 MATCH
ALLSOP                       1 MATCH
ALTINOK                      1
ALVAREZ                      1
```

SPECIFIED Option - Check whether a Value is Passed for an Optional Parameter

Syntax:

```
parameter-name [NOT] SPECIFIED
```

This option is used to check whether an optional parameter in an invoked object (subprogram, external subroutine or ActiveX control) has received a value from the invoking object or not.

An optional parameter is a field defined with the keyword OPTIONAL in the DEFINE DATA PARAMETER statement of the invoked object. If a field is defined as OPTIONAL, a value can - but need not - be passed from an invoking object to this field.

In the invoking statement, the notation *nX* is used to indicate parameters for which no values are passed.

If you process an optional parameter which has not received a value, this will cause a runtime error. To avoid such an error, you use the `SPECIFIED` option in the invoked object to check whether an optional parameter has received a value or not, and then only process it if it has.

`parameter-name` is the name of the parameter as specified in the `DEFINE DATA PARAMETER` statement of the invoked object.

For a field not defined as `OPTIONAL`, the `SPECIFIED` condition is always `TRUE`.

Example of SPECIFIED Option:

Calling Programming:

```
** Example 'LOGICX07': SPECIFIED option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 #PARM1 (A3)
1 #PARM3 (N2)
END-DEFINE
*
#PARM1 := 'ABC'
#PARM3 := 20
*
CALLNAT 'LOGICX08' #PARM1 1X #PARM3
*
END
```

Subprogram Called:

```
** Example 'LOGICX08': SPECIFIED option in logical condition
************************************************************************
DEFINE DATA PARAMETER
1 #PARM1 (A3)
1 #PARM2 (N2) OPTIONAL
1 #PARM3 (N2) OPTIONAL
END-DEFINE
*
WRITE '=' #PARM1
*
IF #PARM2 SPECIFIED
  WRITE '#PARM2 is specified'
  WRITE '=' #PARM2
ELSE
  WRITE '#PARM2 is not specified'
* WRITE '=' #PARM2                /* would cause runtime error NAT1322
END-IF
*
IF #PARM3 NOT SPECIFIED
  WRITE '#PARM3 is not specified'
ELSE
```

```
  WRITE '#PARM3 is specified'
  WRITE '=' #PARM3
END-IF
END
```

Output of Program `LOGICX07`:

```
Page      1                                                  14-01-14  11:25:41

#PARM1: ABC
#PARM2 is not specified
#PARM3 is specified
#PARM3:  20
```

46 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.

Use of Processing Loops

Processing loops can be subdivided into database loops and non-database loops:

- **Database processing loops**
  are those created automatically by Natural to process data selected from a database as a result of a `READ`, `FIND` or `HISTOGRAM` statement. These statements are described in the section ***Database Access***.
- **Non-database processing loops**
  are initiated by the statements `REPEAT`, `FOR`, `CALL FILE`, `CALL LOOP`, `SORT`, and `READ WORK FILE`.

More than one processing loop may be active at the same time. Loops may be embedded or nested within other loops which remain active (open).

A processing loop must be explicitly closed with a corresponding `END-...` statement (for example, `END-REPEAT`, `END-FOR`).

The `SORT` statement, which invokes the sort program of the operating system, closes all active processing loops and initiates a new processing loop.

Limiting Database Loops

The following topics are covered below:

- Priority of Limit Settings

Possible Ways of Limiting Database Loops

With the statements READ, FIND or HISTOGRAM, you have three ways of limiting the number of repetitions of the processing loops initiated with these statements:

- using the session parameter LT,
- using a LIMIT statement,
- or using a **limit notation** in a READ/FIND/HISTOGRAM statement itself.

LT Session Parameter

With the system command GLOBALS, you can specify the session parameter LT, which limits the number of records which may be read in a database processing loop.

Example:

```
GLOBALS LT=100
```

This limit applies to all READ, FIND and HISTOGRAM statements in the entire session.

LIMIT Statement

In a program, you can use the LIMIT statement to limit the number of records which may be read in a database processing loop.

Example:

```
LIMIT 100
```

The LIMIT statement applies to the remainder of the program unless it is overridden by another LIMIT statement or limit notation.

Limit Notation

With a READ, FIND or HISTOGRAM statement itself, you can specify the number of records to be read in parentheses immediately after the statement name.

Example:

```
READ (10) VIEWXYZ BY NAME
```

This limit notation overrides any other limit in effect, but applies only to the statement in which it is specified.

Priority of Limit Settings

If the limit set with the `LT` parameter is smaller than a limit specified with a `LIMIT` statement or a limit notation, the `LT` limit has priority over any of these other limits.

Limiting Non-Database Loops - REPEAT Statement

Non-database processing loops begin and end based on logical condition criteria or some other specified limiting condition.

The `REPEAT` statement is discussed here as representative of a non-database loop statement.

With the `REPEAT` statement, you specify one or more statements which are to be executed repeatedly. Moreover, you can specify a logical condition, so that the statements are only executed either until or as long as that condition is met. For this purpose you use an `UNTIL` or `WHILE` clause.

If you specify the logical condition

- in an `UNTIL` clause, the `REPEAT` loop will continue *until* the logical condition is met;
- in a `WHILE` clause, the `REPEAT` loop will continue *as long as* the logical condition remains true.

If you specify *no* logical condition, the `REPEAT` loop must be exited with one of the following statements:

- `ESCAPE` terminates the execution of the processing loop and continues processing outside the loop (see **below**).
- `STOP` stops the execution of the entire Natural application.
- `TERMINATE` stops the execution of the Natural application and also ends the Natural session.

Example of REPEAT Statement

```
** Example 'REPEAX01': REPEAT
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 SALARY (1:1)
*
1 #PAY1    (N8)
END-DEFINE
*
READ (5) MYVIEW BY NAME WHERE SALARY (1) = 30000 THRU 39999
  MOVE SALARY (1) TO #PAY1
  /*
  REPEAT WHILE #PAY1 LT 40000
    MULTIPLY #PAY1 BY 1.1
    DISPLAY NAME (IS=ON) SALARY (1)(IS=ON) #PAY1
  END-REPEAT
  /*
  SKIP 1
END-READ
END
```

Output of Program REPEAX01:

```
Page      1                                                  14-01-14  14:15:54

        NAME           ANNUAL     #PAY1
                       SALARY
-------------------- ---------- ---------

ADKINSON                  34500     37950
                                    41745

                          33500     36850
                                    40535

                          36000     39600
                                    43560

AFANASSIEV                37000     40700

ALEXANDER                 34500     37950
                                    41745
```

Terminating a Processing Loop - ESCAPE Statement

The `ESCAPE` statement is used to terminate the execution of a processing loop based on a logical condition.

You can place an `ESCAPE` statement within loops in conditional `IF` statement groups, in break processing statement groups (`AT END OF DATA`, `AT END OF PAGE`, `AT BREAK`), or as a stand-alone statement implementing the basic logical conditions of a non-database loop.

The `ESCAPE` statement offers the options `TOP` and `BOTTOM`, which determine where processing is to continue after the processing loop has been left via the `ESCAPE` statement:

- `ESCAPE TOP` is used to continue processing at the top of the processing loop.
- `ESCAPE BOTTOM` is used to continue processing with the first statement following the processing loop.

You can specify several `ESCAPE` statements within the same processing loop.

For further details and examples of the `ESCAPE` statement, see the *Statements* documentation.

Loops Within Loops

A database statement can be placed within a database processing loop initiated by another database statement. When database loop-initiating statements are embedded in this way, a "hierarchy" of loops is created, each of which is processed for each record which meets the selection criteria.

Multiple levels of loops can be embedded. For example, non-database loops can be nested one inside the other. Database loops can be nested inside non-database loops. Database and non-database loops can be nested within conditional statement groups.

Example of Nested FIND Statements

The following program illustrates a hierarchy of two loops, with one `FIND` loop nested or embedded within another `FIND` loop.

```
** Example 'FINDX06': FIND (two FIND statements nested)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 PERSONNEL-ID
1 VEH-VIEW VIEW OF VEHICLES
  2 MAKE
  2 PERSONNEL-ID
END-DEFINE
*
FND1. FIND EMPLOY-VIEW WITH CITY = 'NEW YORK' OR = 'BEVERLEY HILLS'
  FIND (1) VEH-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (FND1.)
    DISPLAY NOTITLE NAME CITY MAKE
  END-FIND
END-FIND
END
```

The above program selects data from multiple files. The outer `FIND` loop selects from the `EMPLOYEES` file all persons who live in New York or Beverley Hills. For each record selected in the outer loop, the inner `FIND` loop is entered, selecting the car data of those persons from the `VEHICLES` file.

Output of Program `FINDX06`:

```
        NAME                 CITY                 MAKE

-------------------- -------------------- --------------------

RUBIN                NEW YORK             FORD
OLLE                 BEVERLEY HILLS       GENERAL MOTORS
WALLACE              NEW YORK             MAZDA
JONES                BEVERLEY HILLS       FORD
SPEISER              BEVERLEY HILLS       GENERAL MOTORS
```

Referencing Statements within a Program

Statement reference notation is used for the following purposes:

- Referring to previous statements in a program in order to specify processing over a particular range of data.
- Overriding Natural's **default referencing**.
- Documenting.

Any Natural statement which causes a processing loop to be initiated and/or causes data elements in a database to be accessed can be referenced, for example:

- `READ`
- `FIND`
- `HISTOGRAM`
- `SORT`
- `REPEAT`
- `FOR`

When multiple processing loops are used in a program, reference notation is used to uniquely identify the particular database field to be processed by referring back to the statement that originally accessed that field in the database.

If a field can be referenced in such a way, this is indicated in the Referencing Permitted column of the *Operand Definition Table* in the corresponding statement description (in the *Statements* documentation). See also ***User-Defined Variables***, ***Referencing of Database Fields Using (r) Notation***.

In addition, reference notation can be specified in some statements. For example:

- `AT START OF DATA`
- `AT END OF DATA`
- `AT BREAK`
- `ESCAPE BOTTOM`

Without reference notation, an `AT START OF DATA`, `AT END OF DATA` or `AT BREAK` statement will be related to the *outermost* active `READ`, `FIND`, `HISTOGRAM`, `SORT` or `READ WORK FILE` loop. With reference notation, you can relate it to another active processing loop.

If reference notation is specified with an `ESCAPE BOTTOM` statement, processing will continue with the first statement following the processing loop identified by the reference notation.

Statement reference notation may be specified in the form of a *statement reference label* or a *source-code line number*.

- **Statement reference label**
  A statement reference label consists of several characters, the last of which must be a period (.). The period serves to identify the entry as a label.

  A statement that is to be referenced is marked with a label by placing the label at the beginning of the line that contains the statement. For example:

```
0030 ...
0040 READ1. READ VIEWXYZ BY NAME
0050 ...
```

In the statement that references the marked statement, the label is placed in parentheses at the location indicated in the statement's syntax diagram (as described in the *Statements* documentation). For example:

```
AT BREAK (READ1.) OF NAME
```

- **Source-code line number**
  If source-code line numbers are used for referencing, they must be specified as 4-digit numbers (leading zeros must not be omitted) and in parentheses. For example:

  ```
  AT BREAK (0040) OF NAME
  ```

  In a statement where the label/line number relates a particular field to a previous statement, the label/line number is placed in parentheses after the field name. For example:

  ```
  DISPLAY NAME (READ1.) JOB-TITLE (READ1.) MAKE MODEL
  ```

Line numbers and labels can be used interchangeably.

See also ***User-Defined Variables***, ***Referencing of Database Fields Using (r) Notation***.

Example of Referencing with Line Numbers

The following program uses source code line numbers (4-digit numbers in parentheses) for referencing.

In this particular example, the line numbers refer to the statements that would be referenced in any case by default.

```
0010 ** Example 'LABELX01': Labels for READ and FIND loops (line numbers)
0020 ************************************************************************
0030 DEFINE DATA LOCAL
0040 1 MYVIEW1 VIEW OF EMPLOYEES
0050   2 NAME
0060  2 FIRST-NAME
0070  2 PERSONNEL-ID
0080 1 MYVIEW2 VIEW OF VEHICLES
0090  2 PERSONNEL-ID
0100  2 MAKE
0110 END-DEFINE
0120 *
```

```
0130 LIMIT 15
0140 READ MYVIEW1 BY NAME STARTING FROM 'JONES'
0150  FIND MYVIEW2 WITH PERSONNEL-ID = PERSONNEL-ID (0140)
0160    IF NO RECORDS FOUND
0170      MOVE '***NO CAR***' TO MAKE
0180    END-NOREC
0190    DISPLAY NOTITLE NAME       (0140) (IS=ON)
0200                    FIRST-NAME (0140) (IS=ON)
0210                    MAKE       (0150)
0220  END-FIND /* (0150)
0230 END-READ   /* (0140)
0240 END
```

Example with Statement Reference Labels

The following example illustrates the use of statement reference labels.

It is identical to the previous program, except that labels are used for referencing instead of line numbers.

```
** Example 'LABELX02': Labels for READ and FIND loops (user labels)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW1 VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 PERSONNEL-ID
1 MYVIEW2 VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 15
RD. READ MYVIEW1 BY NAME STARTING FROM 'JONES'
  FD. FIND MYVIEW2 WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      MOVE '***NO CAR***' TO MAKE
    END-NOREC
    DISPLAY NOTITLE NAME       (RD.) (IS=ON)
                    FIRST-NAME (RD.) (IS=ON)
                    MAKE       (FD.)
  END-FIND /* (FD.)
END-READ   /* (RD.)
END ↩
```

Both programs produce the following output:

```
       NAME              FIRST-NAME             MAKE
-------------------- -------------------- --------------------


JONES                VIRGINIA             CHRYSLER
                     MARSHA               CHRYSLER
                                          CHRYSLER
                     ROBERT               GENERAL MOTORS
                     LILLY                FORD
                                          MG
                     EDWARD               GENERAL MOTORS
                     LAUREL               GENERAL MOTORS
                     KEVIN                DATSUN
                     GREGORY              FORD
JOPER                MANFRED              ***NO CAR***
JOUSSELIN            DANIEL               RENAULT
JUBE                 GABRIEL              ***NO CAR***
JUNG                 ERNST                ***NO CAR***
JUNKIN               JEREMY               ***NO CAR***
KAISER               REINER               ***NO CAR***
KANT                 HEIKE                ***NO CAR***
```

47 Control Breaks

This chapter describes how the execution of a statement can be made dependent on a control break, and how control breaks can be used for the evaluation of Natural system functions.

Use of Control Breaks

A control break occurs when the value of a control field changes.

The execution of statements can be made dependent on a control break.

A control break can also be used for the evaluation of Natural system functions.

System functions are discussed in ***System Variables and System Functions***. For detailed descriptions of the system functions available, refer to the *System Functions* documentation.

AT BREAK Statement

With the statement `AT BREAK`, you specify the processing which is to be performed whenever a control break occurs, that is, whenever the value of a control field which you specify with the `AT BREAK` statement changes. As a control field, you can use a database field or a user-defined variable.

The following topics are covered below:

- Control Break Based on a Database Field
- Control Break Based on a User-Defined Variable
- Multiple Control Break Levels

Control Break Based on a Database Field

The field specified as control field in an `AT BREAK` statement is usually a database field.

Example:

```
...
AT BREAK OF DEPT
  statements
END-BREAK
...
```

In this example, the control field is the database field `DEPT`; if the value of the field changes, for example, `FROM SALE01` to `SALE02`, the *`statements`* specified in the `AT BREAK` statement would be executed.

Instead of an entire field, you can also use only part of a field as a control field. With the slash-*n*-slash notation `/n/`, you can determine that only the first *n* positions of a field are to be checked for a change in value.

Example:

```
...
AT BREAK OF DEPT /4/
  statements
END-BREAK
...
```

In this example, the specified *statements* would only be executed if the value of the first 4 positions of the field `DEPT` changes, for example, FROM `SALE` to `TECH`; if, however, the field value changes from `SALE01` to `SALE02`, this would be ignored and no `AT BREAK` processing performed.

Example:

```
** Example 'ATBREX01': AT BREAK OF (with database field)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 COUNTRY
  2 JOB-TITLE
  2 SALARY (1:1)
END-DEFINE
*
READ (5) MYVIEW BY CITY WHERE COUNTRY = 'USA'
  DISPLAY CITY (AL=9) NAME 'POSITION' JOB-TITLE 'SALARY' SALARY(1)
  /*
  AT BREAK OF CITY
    WRITE /   OLD(CITY) (EM=X^X^X^X^X^X^X^X^X^X^X^)
          5X 'AVERAGE:' T*SALARY AVER(SALARY(1)) //
              COUNT(SALARY(1)) 'RECORDS FOUND' /
  END-BREAK
  /*
  AT END OF DATA
    WRITE 'TOTAL (ALL RECORDS):' T*SALARY(1) TOTAL(SALARY(1))
  END-ENDDATA
END-READ
END
```

In the above program, the first `WRITE` statement is executed whenever the value of the field `CITY` changes.

In the `AT BREAK` statement, the Natural system functions `OLD`, `AVER` and `COUNT` are evaluated (and output in the `WRITE` statement).

In the `AT END OF DATA` statement, the Natural system function `TOTAL` is evaluated.

Output of Program `ATBREX01`:

```
Page      1                                                 14-01-14  14:07:26

  CITY            NAME                    POSITION            SALARY
--------- -------------------- ------------------------- ----------


AIKEN     SENKO                PROGRAMMER                     31500

A I K E N                  AVERAGE:                           31500

       1 RECORDS FOUND

ALBUQUERQ HAMMOND              SECRETARY                      22000
ALBUQUERQ ROLLING              MANAGER                        34000
ALBUQUERQ FREEMAN              MANAGER                        34000
ALBUQUERQ LINCOLN              ANALYST                        41000

A L B U Q U E R Q U E      AVERAGE:                           32750

       4 RECORDS FOUND

TOTAL (ALL RECORDS):                                         162500           ↩
```

Control Break Based on a User-Defined Variable

A **user-defined variable** can also be used as control field in an `AT BREAK` statement.

In the following program, the user-defined variable `#LOCATION` is used as control field.

```
** Example 'ATBREX02': AT BREAK OF  (with user-defined variable and
**                     in conjunction with BEFORE BREAK PROCESSING)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
  2 COUNTRY
  2 JOB-TITLE
  2 SALARY (1:1)
*
1 #LOCATION (A20)
END-DEFINE
*
READ (5) MYVIEW BY CITY WHERE COUNTRY = 'USA'
  BEFORE BREAK PROCESSING
    COMPRESS CITY 'USA' INTO #LOCATION
  END-BEFORE
  DISPLAY #LOCATION 'POSITION' JOB-TITLE 'SALARY' SALARY (1)
```

```
  /*
  AT BREAK OF #LOCATION
    SKIP 1
  END-BREAK
END-READ
END
```

Output of Program ATBREX02:

```
Page      1                                                   14-01-14  14:08:36

     #LOCATION               POSITION            SALARY
-------------------- ------------------------- ----------

AIKEN USA            PROGRAMMER                     31500

ALBUQUERQUE USA      SECRETARY                      22000
ALBUQUERQUE USA      MANAGER                        34000
ALBUQUERQUE USA      MANAGER                        34000
ALBUQUERQUE USA      ANALYST                        41000                ↩
```

Multiple Control Break Levels

As explained **above**, the notation /*n*/ allows some portion of a field to be checked for a control break. It is possible to combine several AT BREAK statements, using an entire field as control field for one break and part of the same field as control field for another break.

In such a case, the break at the lower level (entire field) must be specified before the break at the higher level (part of field); that is, in the first AT BREAK statement the entire field must be specified as control field, and in the second one part of the field.

The following example program illustrates this, using the field DEPT as well as the first 4 positions of that field (DEPT /4/).

```
** Example 'ATBREX03': AT BREAK OF (two statements in combination)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 DEPT
  2 SALARY    (1:1)
  2 CURR-CODE (1:1)
END-DEFINE
*
READ MYVIEW BY DEPT STARTING FROM 'SALE40' ENDING AT 'TECH10'
     WHERE SALARY(1) GT 47000 AND CURR-CODE(1) = 'USD'
  /*
  AT BREAK OF DEPT
    WRITE '*** LOWEST BREAK LEVEL ***' /
```

```
  END-BREAK
  AT BREAK OF DEPT /4/
    WRITE '*** HIGHEST BREAK LEVEL ***'
  END-BREAK
  /*
  DISPLAY DEPT NAME 'POSITION' JOB-TITLE
END-READ
END
```

Output of Program ATBREX03:

```
Page      1                                                  14-01-14  14:09:20

DEPARTMENT         NAME                 POSITION
   CODE
---------- -------------------- -------------------------

TECH05     HERZOG               MANAGER
TECH05     LAWLER               MANAGER
TECH05     MEYER                MANAGER
*** LOWEST BREAK LEVEL ***

TECH10     DEKKER               DBA
*** LOWEST BREAK LEVEL ***

*** HIGHEST BREAK LEVEL ***    ↩
```

In the following program, one blank line is output whenever the value of the field DEPT changes; and whenever the value in the first 4 positions of DEPT changes, a record count is carried out by evaluating the system function COUNT.

```
** Example 'ATBREX04': AT BREAK OF (two statements in combination)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 DEPT
  2 REDEFINE DEPT
    3 #GENDEP (A4)
  2 NAME
  2 SALARY    (1)
END-DEFINE
*
WRITE TITLE '** PERSONS WITH SALARY > 30000, SORTED BY DEPARTMENT **' /
LIMIT 9
READ MYVIEW BY DEPT FROM 'A' WHERE SALARY(1) > 30000
  DISPLAY 'DEPT' DEPT NAME 'SALARY' SALARY(1)
  /*
  AT BREAK OF DEPT
    SKIP 1
  END-BREAK
  AT BREAK OF DEPT /4/
```

```
    WRITE COUNT(SALARY(1)) 'RECORDS FOUND IN:' OLD(#GENDEP) /
  END-BREAK
END-READ
END
```

Output of Program ATBREX04:

```
            ** PERSONS WITH SALARY > 30000, SORTED BY DEPARTMENT **

 DEPT          NAME           SALARY
------ -------------------- ----------

ADMA01 JENSEN                   180000
ADMA01 PETERSEN                 105000
ADMA01 MORTENSEN                320000
ADMA01 MADSEN                   149000
ADMA01 BUHL                     642000

ADMA02 HERMANSEN                391500
ADMA02 PLOUG                    162900
ADMA02 HANSEN                   234000

       8 RECORDS FOUND IN: ADMA

COMP01 HEURTEBISE               168800

       1 RECORDS FOUND IN: COMP                                                ↩
```

Automatic Break Processing

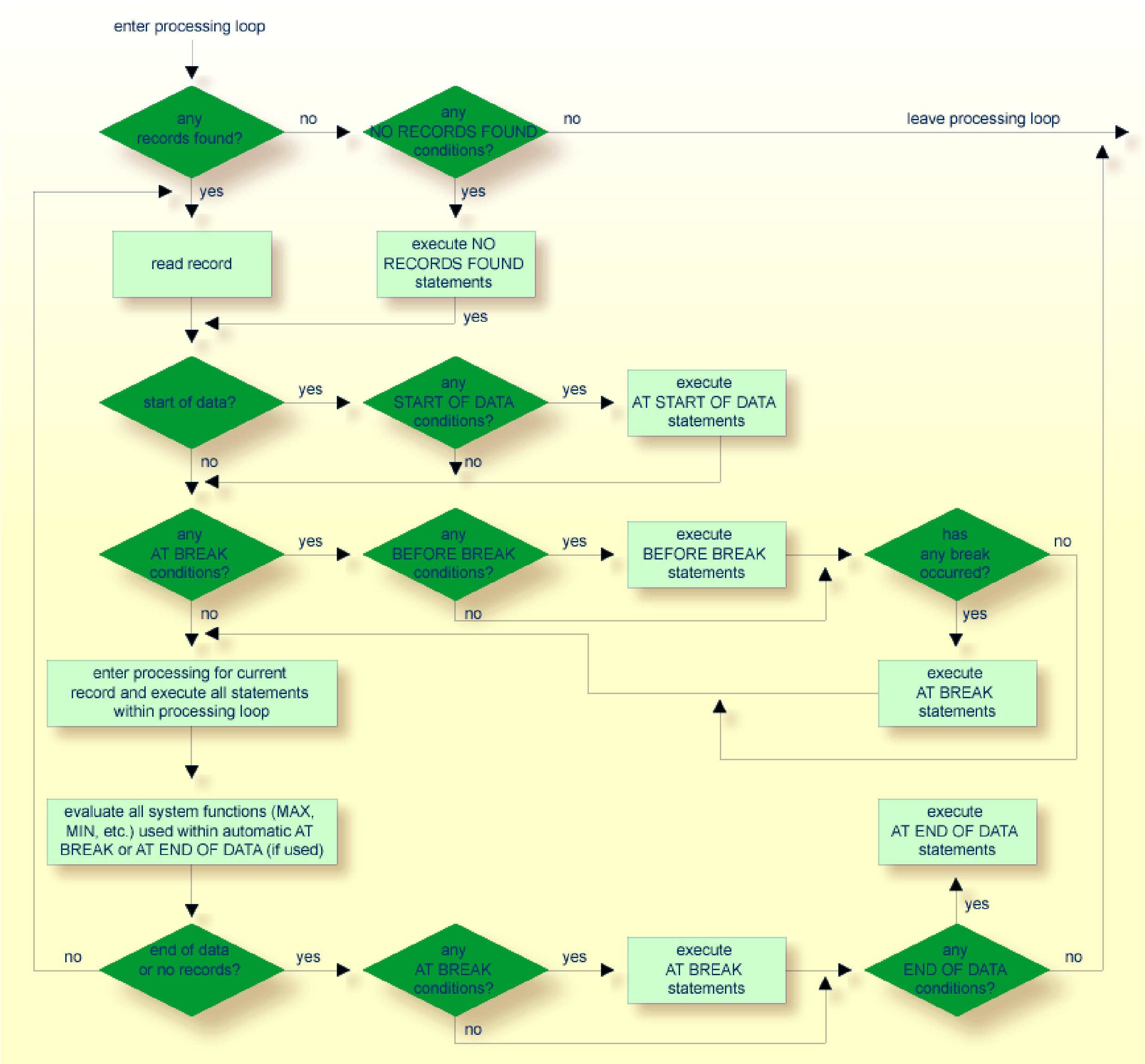
Automatic break processing is in effect for a processing loop which contains an AT BREAK statement. This applies to the following statements:

- FIND
- READ
- HISTOGRAM
- SORT
- READ WORK FILE

The value of the control field specified with the AT BREAK statement is checked only for records which satisfy the selection criteria of both the WITH clause and the WHERE clause.

Natural system functions (AVER, MAX, MIN, etc.) are evaluated for each record after all statements within the processing loop have been executed. System functions are not evaluated for any record which is rejected by WHERE criteria.

The figure below illustrates the flow logic of automatic break processing.



Example of System Functions with AT BREAK Statement

The following example shows the use of the Natural system functions `OLD`, `MIN`, `AVER`, `MAX`, `SUM` and `COUNT` in an `AT BREAK` statement (and of the system function `TOTAL` in an `AT END OF DATA` statement).

```
** Example 'ATBREX05': AT BREAK OF (with system functions)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 SALARY    (1:1)
  2 CURR-CODE (1:1)
END-DEFINE
*
LIMIT 3
READ MYVIEW BY CITY = 'SALT LAKE CITY'
  DISPLAY NOTITLE CITY NAME 'SALARY' SALARY(1) 'CURRENCY' CURR-CODE(1)
  /*
  AT BREAK OF CITY
    WRITE /  OLD(CITY) (EM=X^X^X^X^X^X^X^X^X^X^X^X^X^X^X)
      31T ' - MINIMUM:' MIN(SALARY(1))  CURR-CODE(1) /
      31T ' - AVERAGE:' AVER(SALARY(1)) CURR-CODE(1) /
      31T ' - MAXIMUM:' MAX(SALARY(1))  CURR-CODE(1) /
      31T ' -     SUM:' SUM(SALARY(1))  CURR-CODE(1) /
      33T COUNT(SALARY(1)) 'RECORDS FOUND' /
  END-BREAK
  /*
  AT END OF DATA
    WRITE 22T 'TOTAL (ALL RECORDS):'
               T*SALARY TOTAL(SALARY(1)) CURR-CODE(1)
  END-ENDDATA
END-READ
END
```

Output of Program ATBREX05:

```
        CITY                 NAME           SALARY   CURRENCY
-------------------- -------------------- ---------- --------

SALT LAKE CITY       ANDERSON                  50000 USD
SALT LAKE CITY       SAMUELSON                 24000 USD

S A L T   L A K E   C I T Y    - MINIMUM:      24000 USD
                               - AVERAGE:      37000 USD
                               - MAXIMUM:      50000 USD
                               -     SUM:      74000 USD
                                   2 RECORDS FOUND

SAN DIEGO            GEE                       60000 USD

S A N   D I E G O              - MINIMUM:      60000 USD
                               - AVERAGE:      60000 USD
                               - MAXIMUM:      60000 USD
                               -     SUM:      60000 USD
                                   1 RECORDS FOUND
```

```
                    TOTAL (ALL RECORDS):     134000 USD    ↩
```

Further Example of AT BREAK Statement

See the following example program:

- *ATBREX06 - AT BREAK OF (comparing NMIN, NAVER, NCOUNT with MIN, AVER, COUNT)*

BEFORE BREAK PROCESSING Statement

With the `BEFORE BREAK PROCESSING` statement, you can specify statements that are to be executed immediately before a control break; that is, before the value of the control field is checked, before the statements specified in the `AT BREAK` block are executed, and before any Natural system functions are evaluated.

Example of BEFORE BREAK PROCESSING Statement

```
** Example 'BEFORX01': BEFORE BREAK PROCESSING
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:1)
  2 BONUS  (1:1,1:1)
*
1 #INCOME  (P11)
END-DEFINE
*
LIMIT 5
READ MYVIEW BY NAME FROM 'B'
 BEFORE BREAK PROCESSING
    COMPUTE #INCOME = SALARY(1) + BONUS(1,1)
  END-BEFORE
  /*
  DISPLAY NOTITLE NAME FIRST-NAME (AL=10)
                 'ANNUAL/INCOME' #INCOME 'SALARY' SALARY(1) (LC==) /
                 '+ BONUS' BONUS(1,1) (IC=+)
  AT BREAK OF #INCOME
    WRITE T*#INCOME '-'(24)
  END-BREAK
```

```
END-READ
END
```

Output of Program BEFORX01:

```
        NAME             FIRST-NAME    ANNUAL      SALARY
                                       INCOME      + BONUS
-------------------- ---------- ------------ -----------

BACHMANN             HANS              56800 =     52800
                                                   +4000
                                ------------------------
BAECKER              JOHANNES          81000 =     74400
                                                   +6600
                                ------------------------
BAECKER              KARL              52650 =     48600
                                                   +4050
                                ------------------------
BAGAZJA              MARJAN           152700 =    129700
                                                  +23000
                                ------------------------
BAILLET              PATRICK          198500 =    188000
                                                  +10500
                                ------------------------
```

User-Initiated Break Processing - PERFORM BREAK PROCESSING Statement

With automatic break processing, the statements specified in an AT BREAK block are executed whenever the value of the specified control field changes - regardless of the position of the AT BREAK statement in the processing loop.

With a PERFORM BREAK PROCESSING statement, you can perform break processing at a specified position in a processing loop: the PERFORM BREAK PROCESSING statement is executed when it is encountered in the processing flow of the program.

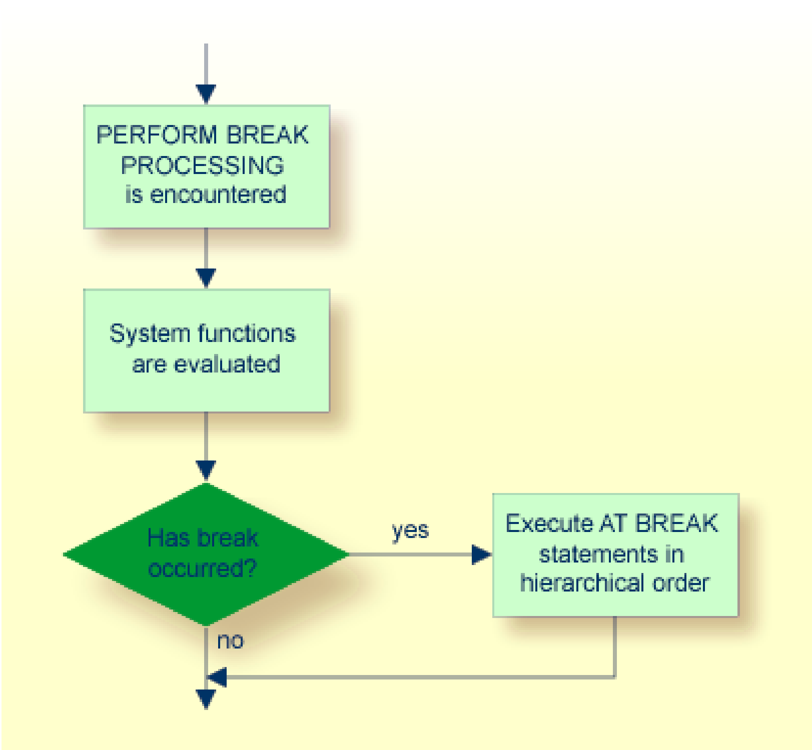
Immediately after the PERFORM BREAK PROCESSING, you specify one or more AT BREAK statement blocks:

```
...
PERFORM BREAK PROCESSING
  AT BREAK OF field1
    statements
  END-BREAK
  AT BREAK OF field2
    statements
  END-BREAK
...
```

When a PERFORM BREAK PROCESSING is executed, Natural checks if a break has occurred; that is, if the value of the specified control field has changed; and if it has, the specified statements are executed.

With PERFORM BREAK PROCESSING, system functions are evaluated *before* Natural checks if a break has occurred.

The following figure illustrates the flow logic of user-initiated break processing:



Example of PERFORM BREAK PROCESSING Statement

```
** Example 'PERFBX01': PERFORM BREAK PROCESSING (with BREAK option
**                     in IF statement)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 DEPT
  2 SALARY (1:1)
*
1 #CNTL    (N2)
END-DEFINE
*
```

```
LIMIT 7
READ MYVIEW BY DEPT
  AT BREAK OF DEPT              /* <- automatic break processing
    SKIP 1
    WRITE 'SUMMARY FOR ALL SALARIES         '
          'SUM:'   SUM(SALARY(1))
          'TOTAL:' TOTAL(SALARY(1))
    ADD 1 TO #CNTL
  END-BREAK
  /*
  IF SALARY (1) GREATER THAN 100000 OR BREAK #CNTL
    PERFORM BREAK PROCESSING    /* <- user-initiated break processing
    AT BREAK OF #CNTL
      WRITE 'SUMMARY FOR SALARY GREATER 100000'
            'SUM:'   SUM(SALARY(1))
            'TOTAL:' TOTAL(SALARY(1))
    END-BREAK
  END-IF
  /*
  IF SALARY (1) GREATER THAN 150000 OR BREAK #CNTL
    PERFORM BREAK PROCESSING    /* <- user-initiated break processing
    AT BREAK OF #CNTL
      WRITE 'SUMMARY FOR SALARY GREATER 150000'
            'SUM:'   SUM(SALARY(1))
            'TOTAL:' TOTAL(SALARY(1))
    END-BREAK
  END-IF
  DISPLAY NAME DEPT SALARY(1)
END-READ
END
```

Output of Program PERFBX01:

```
Page      1                                                  14-01-14  14:13:35

        NAME         DEPARTMENT   ANNUAL
                        CODE      SALARY
-------------------- ---------- ----------

JENSEN               ADMA01         180000
PETERSEN             ADMA01         105000
MORTENSEN            ADMA01         320000
MADSEN               ADMA01         149000
BUHL                 ADMA01         642000

SUMMARY FOR ALL SALARIES          SUM:    1396000 TOTAL:    1396000
SUMMARY FOR SALARY GREATER 100000 SUM:    1396000 TOTAL:    1396000
SUMMARY FOR SALARY GREATER 150000 SUM:    1142000 TOTAL:    1142000
HERMANSEN            ADMA02         391500
PLOUG                ADMA02         162900

SUMMARY FOR ALL SALARIES          SUM:     554400 TOTAL:    1950400
```

```
SUMMARY FOR SALARY GREATER 100000 SUM:      554400 TOTAL:    1950400
SUMMARY FOR SALARY GREATER 150000 SUM:      554400 TOTAL:    1696400          ↩
```

48 Stack Processing

The Natural stack is a kind of "intermediate storage" in which you can store Natural commands, user-defined commands, and input data to be used by an INPUT statement.

Use of Natural Stack

In the stack you can store a series of functions which are frequently executed one after the other, such as a series of logon commands.

The data/commands stored in the stack are "stacked" on top of one another. You can decide whether to put them on top or at the bottom of the stack. The data/command in the stack can only be processed in the order in which they are stacked, beginning from the top of the stack.

In a program, you may reference the system variable *DATA to determine the content of the stack (see the *System Variables* documentation for further information).

Processing Order for Stacked Commands/Data

The processing of the commands/data stored in the stack differs depending on the function being performed.

If a command is expected, that is, the NEXT prompt is about to be displayed, Natural first checks if a command is on the top of the stack. If there is, the NEXT prompt is suppressed and the command is read and deleted from the stack; the command is then executed as if it had been entered manually in response to the NEXT prompt.

If an INPUT statement containing input fields is being executed, Natural first checks if there are any input data on the top of the stack. If there are, these data are passed to the INPUT statement (in delimiter mode); the data read from the stack must be format-compatible with the variables in the INPUT statement; the data are then deleted from the stack. See also *Processing Data from the Natural Stack* in the INPUT statement description.

If an INPUT statement was executed using data from the stack, and this INPUT statement is re-executed via a REINPUT statement, the INPUT statement screen will be re-executed displaying the same data from the stack as when it was executed originally. With the REINPUT statement, no further data are read from the stack.

When a Natural program terminates normally, the stack is flushed beginning from the top until either a command is on the top of the stack or the stack is cleared. When a Natural program is terminated via the terminal command %% or with an error, the stack is cleared entirely.

Placing Data on the Stack

The following methods can be used to place data/commands on the stack:

- STACK Parameter
- STACK Statement
- FETCH and RUN Statements

STACK Parameter

The Natural profile parameter `STACK` may be used to place data/commands on the stack. The `STACK` parameter (described in the *Parameter Reference*) can be specified by the Natural administrator in the Natural parameter module at the installation of Natural; or you can specify it as a dynamic parameter when you invoke Natural.

When data/commands are to be placed on the stack via the `STACK` parameter, multiple commands must be separated from one another by a semicolon (;). If a command is to be passed within a sequence of data or command elements, it must be preceded by a semicolon.

Data for multiple `INPUT` statements must be separated from one another by a colon (:). Data that are to be read by a separate `INPUT` statement must be preceded by a colon. If a command is to be stacked which requires parameters, no colon is to be placed between the command and the parameters.

Semicolon and colon must not be used within the input data themselves as they will be interpreted as separation characters.

STACK Statement

The `STACK` statement can be used within a program to place data/commands in the stack. The data elements specified in one `STACK` statement will be used for one `INPUT` statement, which means that if data for multiple `INPUT` statements are to be placed on the stack, multiple `STACK` statements must be used.

Data may be placed on the stack either unformatted or formatted:

- If unformatted data are read from the stack, the data string is interpreted in delimiter mode and the characters specified with the session parameters `IA` (Input Assignment character) and `ID` (Input Delimiter character) are processed as control characters for **keyword** assignment and data separation.
- If formatted data are placed on the stack, each content of a field will be separated and passed to one input field in the corresponding `INPUT` statement. If the data to be placed on the stack contains delimiter, control or DBCS characters, it should be placed formatted on the stack to avoid unintentional interpretation of these characters.

See the *Statements* documentation for further information on the `STACK` statement.

FETCH and RUN Statements

The execution of a `FETCH` or `RUN` statement that contains parameters to be passed to the invoked program will result in these parameters being placed on top of the stack.

Clearing the Stack

The contents of the stack can be deleted with the `RELEASE` statement. See the *Statements* documentation for details on the `RELEASE` statement.

Note: When a Natural program is terminated via the terminal command `%%` or with an error, the stack is cleared entirely.

49 System Variables and System Functions

This chapter describes the purpose of Natural system variables and Natural system functions and how they are used in Natural programs.

System Variables

The following topics are covered below:

Purpose

System variables are used to display system information. They may be referenced at any point within a Natural program.

Natural system variables provide variable information, for example, about the current Natural session:

- the current library;
- the user and terminal identification;
- the current status of a loop processing;
- the current report processing status;
- the current date and time.

The typical use of system variables is illustrated in the ***Example of System Variables and System Functions*** below and in the examples contained in library `SYSEXPG`.

The information contained in a system variable may be used in Natural programs by specifying the appropriate system variables. For example, date and time system variables may be specified in a `DISPLAY`, `WRITE`, `PRINT`, `MOVE` or `COMPUTE` statement.

Characteristics of System Variables

The names of all system variables begin with an asterisk (*).

Format/Length

Information on format and length is given in the detailed descriptions in the *System Variables* documentation. The following abbreviations are used:

Format	
A	Alphanumeric
B	Binary
D	Date
I	Integer
L	Logical
N	Numeric (unpacked)
P	Packed numeric
T	Time

Content Modifiable

In the individual descriptions, this indicates whether in a Natural program you can assign another value to the system variable, that is, overwrite its content as generated by Natural.

System Variables Grouped by Function

The Natural system variables are grouped as follows:

- Application-Related System Variables
- Date and Time System Variables
- Input/Output Related System Variables
- JSON-Related System Variables
- Natural Environment-Related System Variables
- System Environment*Related System Variables
- XML-Related System Variables

For detailed descriptions of all system variables, see the *System Variables* documentation.

System Functions

Natural system functions comprise a set of statistical and mathematical functions that can be applied to the data after a record has been processed, but before break processing occurs.

System functions may be specified in a `DISPLAY`, `WRITE`, `PRINT`, `MOVE` or `COMPUTE` statement that is used in conjunction with an `AT END OF PAGE`, `AT END OF DATA` or `AT BREAK` statement.

In the case of an `AT END OF PAGE` statement, the corresponding `DISPLAY` statement must include the `GIVE SYSTEM FUNCTIONS` clause (as shown in the example **below**).

The following functional groups of system functions exist:

- System Functions for Use in Processing Loops
- Mathematical Functions
- Miscellaneous Functions

For detailed information on all system functions available, see the *System Functions* documentation.

See also *Using System Functions in Processing Loops* (in the *System Functions* documentation).

The typical use of system functions is explained in the example programs given below and in the examples contained in library SYSEXPG.

Example of System Variables and System Functions

The following example program illustrates the use of system variables and system functions:

```
** Example 'SYSVAX01': System variables and system functions
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
  2 INCOME    (1:1)
    3 CURR-CODE
    3 SALARY
    3 BONUS   (1:1)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED 'EMPLOYEE SALARY REPORT AS OF' *DAT4E /
*
READ (3) MYVIEW BY CITY STARTING FROM 'E'
  DISPLAY GIVE SYSTEM FUNCTIONS
          NAME (AL=15) JOB-TITLE (AL=15) INCOME (1:1)
  AT START OF DATA
    WRITE 'REPORT CREATED AT:' *TIME 'HOURS' /
  END-START
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD (NAME) /
  END-ENDDATA
END-READ
*
AT END OF PAGE
  WRITE 'AVERAGE SALARY:' AVER (SALARY(1))
END-ENDPAGE
END
```

Explanation:

- The system variable `*DATE` is output with the `WRITE TITLE` statement.
- The system variable `*TIME` is output with the `AT START OF DATA` statement.
- The system function `OLD` is used in the `AT END OF DATA` statement.
- The system function `AVER` is used in the `AT END OF PAGE` statement.

Output of Program `SYSVAX01`:

Note how the system variables and system function are displayed.

```
EMPLOYEE SALARY REPORT AS OF 11/11/2004

     NAME           CURRENT                 INCOME
                   POSITION
                               CURRENCY   ANNUAL     BONUS
                                 CODE     SALARY
--------------- --------------- -------- ---------- ----------

REPORT CREATED AT: 14:15:55.0 HOURS

DUYVERMAN       PROGRAMMER      USD           34000          0
PRATT           SALES PERSON    USD           38000       9000
MARKUSH         TRAINEE         USD           22000          0

LAST PERSON SELECTED: MARKUSH

AVERAGE SALARY:      31333
```

Further Examples of System Variables

See the following example programs:

- *EDITMX05 - Edit mask (EM for date and time system variables)*
- *READX04 - READ (in combination with FIND and the system variables *NUMBER and *COUNTER)*
- *WTITLX01 - WRITE TITLE (with *PAGE-NUMBER)*

Further Examples of System Functions

See the following example programs:

- *ATBREX06 - AT BREAK OF (comparing NMIN, NAVER, NCOUNT with MIN, AVER, COUNT)*
- *ATENPX01 - AT END OF PAGE (with system function available via GIVE SYSTEM FUNCTIONS in DISPLAY)*

50 Processing of Date Information

This chapter covers various aspects concerning the handling of date information in Natural applications.

Edit Masks for Date Fields and Date System Variables

If you wish the value of a date field to be output in a specific representation, you usually specify an **edit mask** for the field. With an edit mask, you determine character by character what the output is to look like.

If you wish to use the current date in a specific representation, you need not define a date field and specify an edit mask for it; instead you can simply use a *date system variable*. Natural provides various date system variables, which contain the current date in different representations. Some of these representations contain a 2-digit year, some a 4-digit year.

For more information and a list of all date system variables, see the *System Variables* documentation.

Default Edit Mask for Date - DTFORM Parameter

The profile parameter `DTFORM` determines the default format used for dates as part of the default title on Natural reports, for date constants and for date input.

This date format determines the sequence of the day, month and year components of a date, as well as the delimiter characters to be used between these components.

Possible `DTFORM` settings are:

Setting	Date Format[1]	Example[2]
`DTFORM=I`	*yyyy-mm-dd*	`2014-01-31`
`DTFORM=G`	*dd.mm.yyyy*	`31.01.2014`
`DTFORM=E`	*dd/mm/yyyy*	`31/01/2014`
`DTFORM=U`	*mm/dd/yyyy*	`01/31/2014`

[1] *dd* = day, *mm* = month, *yyyy* = year
[2] assumed that `DF` or `DFTITLE` is set to `L`

The `DTFORM` parameter can be set in the Natural parameter module/file or dynamically when Natural is invoked. By default, `DTFORM=I` applies.

Date Format for Alphanumeric Representation - DF Parameter

If an edit mask is specified, the representation of the field value is determined by the edit mask. If no edit mask is specified, the representation of the field value is determined by the session parameter DF in combination with the profile parameter DTFORM.

With the DF parameter, you can choose one of the following date representations:

DF=S	8-byte representation with a 2-digit year and delimiters. Example: *yy-mm-dd*.
DF=I	8-byte representation with a 4-digit year without delimiters. Example: *yyyymmdd*.
DF=L	10-byte representation with a 4-digit year and delimiters. Example: *yyyy-mm-dd*.

For each representation, the sequence of the day, month and year components, and the delimiter characters used are determined by the DTFORM parameter.

By default, DF=S applies (except for INPUT statements; see below).

The session parameter DF is evaluated at compilation.

It can be specified with the following statements:

- FORMAT,
- INPUT, DISPLAY, WRITE and PRINT at statement and element (field) level,
- MOVE, COMPRESS, STACK, RUN and FETCH at element (field) level.

When specified in one of these statements, the DF parameter applies to the following:

Statement	Effect of DF parameter
DISPLAY, WRITE, PRINT	When the value of a date variable is output with one of these statements, the value is converted to an alphanumeric representation before it is output. The DF parameter determines which representation is used.
MOVE, COMPRESS	When the value of a date variable is transferred to an alphanumeric field with a MOVE or COMPRESS statement, the value is converted to an alphanumeric representation before it is transferred. The DF parameter determines which representation is used.
STACK, RUN, FETCH	When the value of a date variable is placed on the stack, it is converted to alphanumeric representation before it is placed on the stack. The DF parameter determines which representation is used. The same applies when a date variable is specified as a parameter in a FETCH or RUN statement (as these parameters are also passed via the stack).

Statement	Effect of DF parameter
INPUT	When a data variable is used in an INPUT statement, the DF parameter determines how a value must be entered in the field. However, when a date variable for which *no* DF parameter is specified is used in an INPUT statement, the date can be entered either with a 2-digit year and delimiters or with a 4-digit year and no delimiters. In this case, too, the sequence of the day, month and year, and the delimiter characters to be used, are determined by the DTFORM parameter.

Note: With DF=S, only 2 digits are provided for the year information; this means that if a date value contained the century, this information would be lost during the conversion. To retain the century information, you set DF=I or DF=L.

Examples of DF Parameter with WRITE Statements

These examples assume that DTFORM=G applies.

```
/* DF=S (default)
WRITE *DATX   /* Output has this format: dd.mm.yy
END
```

```
FORMAT DF=I
WRITE *DATX   /* Output has this format: ddmmyyyy
END
```

```
FORMAT DF=L
WRITE *DATX   /* Output has this format: dd.mm.yyyy
END
```

Example of DF Parameter with MOVE Statement

This example assumes that DTFORM=E applies.

```
DEFINE DATA LOCAL
  1 #DATE  (D) INIT <D'31/01/2014'>
  1 #ALPHA (A10)
END-DEFINE
...
MOVE #DATE        TO #ALPHA /* Result: #ALPHA contains 31/01/14
MOVE #DATE (DF=I) TO #ALPHA /* Result: #ALPHA contains 31012014
MOVE #DATE (DF=L) TO #ALPHA /* Result: #ALPHA contains 31/01/2014
...
```

Example of DF Parameter with STACK Statement

This example assumes that `DTFORM=I` applies.

```
DEFINE DATA LOCAL
  1 #DATE  (D) INIT <D'2014-01-31'>
  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 14-01-31
/*         #ALPHA2 contains 20140131
/*         #ALPHA3 contains 2014-01-31
...
```

Example of DF Parameter with INPUT Statement

This example assumes that `DTFORM=I` applies.

```
DEFINE DATA LOCAL
  1 #DATE1 (D)
  1 #DATE2 (D)
  1 #DATE3 (D)
  1 #DATE4 (D)
END-DEFINE
...
INPUT #DATE1 (DF=S) /* Input must have this format: yy-mm-dd
      #DATE2 (DF=I) /* Input must have this format: yyyymmdd
      #DATE3 (DF=L) /* Input must have this format: yyyy-mm-dd
      #DATE4        /* Input must have this format: yy-mm-dd or yyyymmdd
...
```

Date Format for Output - DFOUT Parameter

The session/profile parameter `DFOUT` only applies to date fields in `INPUT`, `DISPLAY`, `WRITE` and `PRINT` statements for which no edit mask is specified, and for which no `DF` parameter applies.

For date fields which are displayed by `INPUT`, `DISPLAY`, `PRINT` and `WRITE` statements and for which neither an edit mask is specified nor a `DF` parameter applies, the profile/session parameter `DFOUT` determines the format in which the field values are displayed.

With the `DFOUT` parameter, you can choose one of the following date representations:

DFOUT=S	8-byte representation with a 2-digit year and delimiters. Example: *yy-mm-dd*.
DFOUT=I	8-byte representation with a 4-digit year and no delimiters. Example: *yyyymmdd*.

By default, DFOUT=S applies.

For each representation, the sequence of the day, month and year components and the delimiter characters used (if so) are determined by the DTFORM parameter.

The lengths of the date fields are not affected by the DFOUT setting, as either date value representation fits into an 8-byte field.

The DFOUT parameter can be set in the Natural parameter module/file, dynamically when Natural is invoked, or at session level with the system command GLOBALS. It is evaluated at runtime.

Example:

This example assumes that DTFORM=I applies.

```
DEFINE DATA LOCAL
1 #DATE  (D) INIT <D'2014-01-31'>
END-DEFINE
...
WRITE #DATE        /* Output if DFOUT=S is set ...: 14-01-31
                   /* Output if DFOUT=I is set ...: 20140131
WRITE #DATE (DF=L) /* Output (regardless of DFOUT): 2014-01-31
...
```

Date Format for Stack - DFSTACK Parameter

The session/profile parameter DFSTACK only applies to date fields used in STACK, FETCH and RUN statements for which no DF parameter has been specified.

The DFSTACK parameter determines the format in which the values of date variables are placed on the stack via a STACK, RUN or FETCH statement.

The DFSTACK parameter can be set in the Natural parameter module/file, dynamically when Natural is invoked, or at session level with the system command GLOBALS. It is evaluated at runtime.

Possible DFSTACK settings are:

DFSTACK=S	8-byte date variables are placed on the stack with a 2-digit year and delimiters. Example: *yy-mm-dd*. DFSTACK=S places a date on the stack 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 also ***Year Sliding Window***). 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	Same as DFSTACK=S. However: Natural issues a runtime error if the date value to be stacked would result in a century different from that of the original value, either because of the YSLW parameter or because the original century is not the same as the current century.
DFSTACK=I	8-byte date variables are placed on the stack with a 4-digit year and no delimiters. Example: *yyyymmdd*.

By default, DFSTACK=S applies.

For each date representation, the sequence of the day, month and year components and the delimiter characters used (if so) are determined by the DTFORM parameter.

Example:

This example assumes that DTFORM=I and YSLW=0 apply.

```
DEFINE DATA LOCAL
  1 #DATE  (D) INIT <D'2014-01-31'>
  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 14-01-31
/* Result if DFSTACK=I is set .....: #ALPHA1 contains 20140131
/* Result (regardless of DFSTACK) .: #ALPHA2 contains 2014-01-31
...
```

Year Sliding Window - YSLW Parameter

The profile parameter YSLW allows you determine the century of a 2-digit year value.

The YSLW parameter can be set in the Natural parameter module/file or dynamically when Natural is invoked. It is evaluated at runtime when an alphanumeric date value with a 2-digit year is moved into a date variable. This applies to data values which are:

- used with the **mathematical function** VAL(*field*),
- used with the IS(D) option in a logical condition,
- read from the **stack** as input data, or
- entered in an input field as input data.

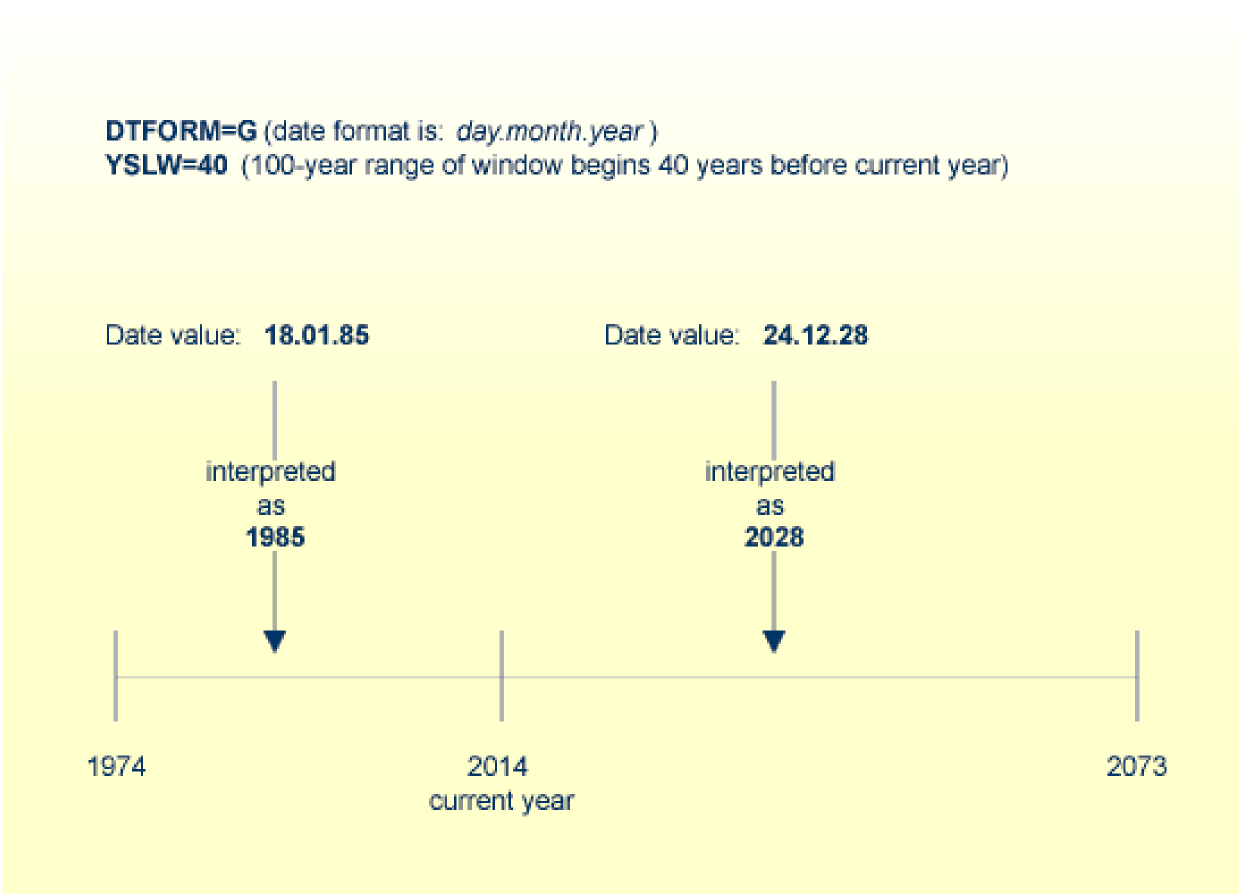
The YSLW parameter determines the range of years covered by a so-called "year sliding window". The sliding-window mechanism assumes a date with a 2-digit year to be within a "window" of 100 years. Within these 100 years, every 2-digit year value can be uniquely related to a specific century.

With the YSLW parameter, you determine how many years in the past that 100-year range is to begin: The YSLW value is subtracted from the current year to determine the first year of the window range.

Possible values of the YSLW parameter are 0 to 99. The default value is YSLW=0, which means that no sliding-window mechanism is used; that is, a date with a 2-digit year is assumed to be in the current century.

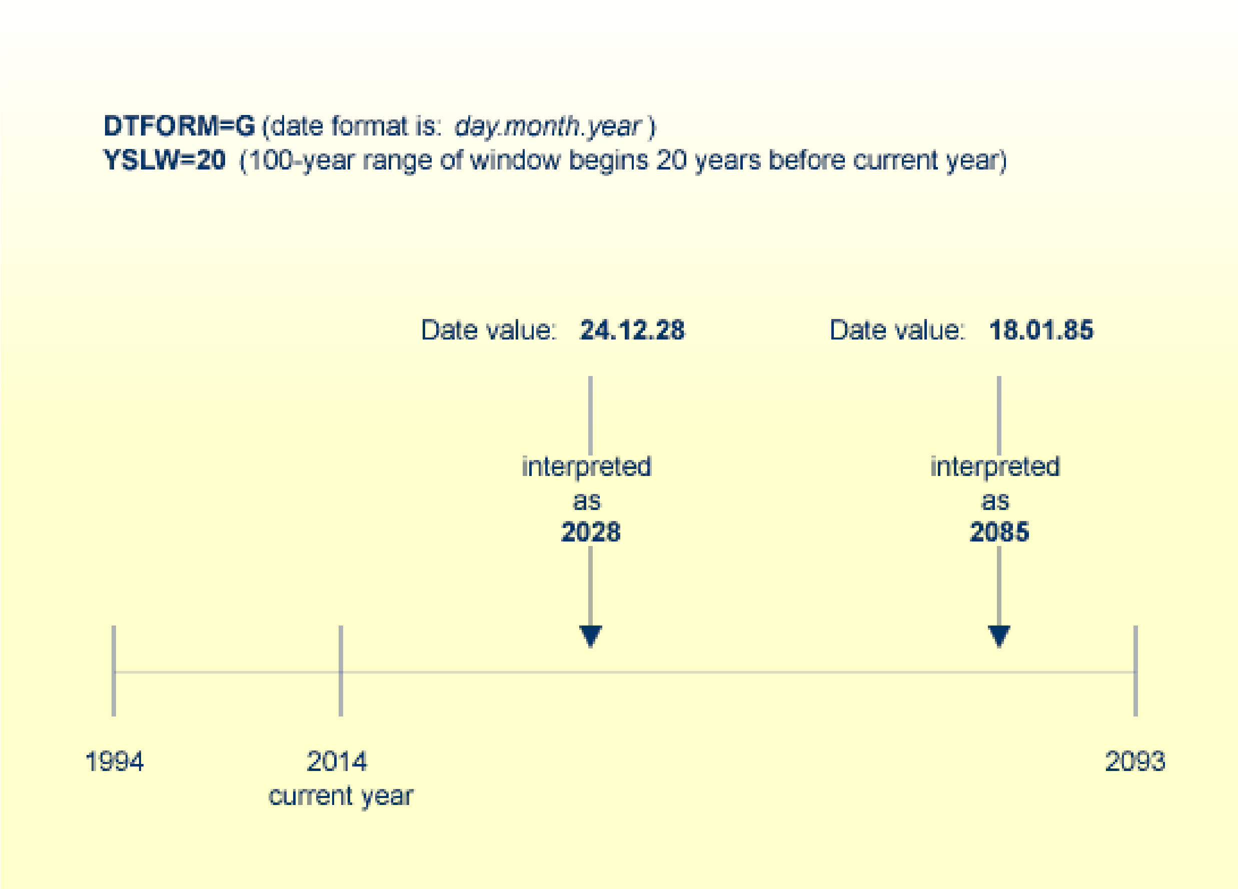
Example 1:

If the current year is 2014 and you specify YSLW=40, the sliding window will cover the years 1974 to 2073. A 2-digit year value *nn* from 74 to 99 is interpreted accordingly as 19*nn*, while a 2-digit year value *nn* from 00 to 73 is interpreted as 20*nn*.



Example 2:

If the current year is 2014 and you specify `YSLW=20`, the sliding window will cover the years 1994 to 2093. A 2-digit year value *nn* from 94 to 99 is interpreted accordingly as 19*nn*, while a 2-digit year value *nn* from 00 to 93 is interpreted as 20*nn*.



Combinations of DFSTACK and YSLW

The following examples illustrate the effects of using various combinations of the parameters DFSTACK and YSLW.

Note: All these examples assume that DTFORM=I applies.

Example 1:

This example assumes the current year to be 2014, and that the parameter settings DFSTACK=S (default) and YSLW=20 apply.

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'1956-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2          /* year sliding window determines 56 to be 2056
...
/* Result: #DATE2 contains 2056-12-31
```

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 2014, and that the parameter settings `DFSTACK=S` (default) and `YSLW=60` apply.

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'1956-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2          /* year sliding window determines 56 to be 1956
...
/* Result: #DATE2 contains 1956-12-31
```

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 2014, and that the parameter settings `DFSTACK=C` and `YSLW=0` (default) apply.

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'1956-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2          /* 56 is assumed to be in current century -> 2056
...
/* Result: NAT1130 runtime error (Unintended century switch...)
```

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 2014, and that the parameter settings DFSTACK=C and YSLW=60 apply.

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'2056-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2          /* year sliding window determines 56 to be 1956
...
/* Result: NAT1130 runtime error (Unintended century switch...)
```

In this case, the century information is changed due to the year sliding window. However, this change is intercepted by the DFSTACK=C setting.

Year Fixed Window

For information on this topic, see the description of the profile parameter YSLW.

Date Format for Default Page Title - DFTITLE Parameter

The session/profile parameter DFTITLE determines the format of the date in a default **page title** (as output with a DISPLAY, WRITE or PRINT statement).

With the DFTITLE parameter, you can choose one of the following date representations:

DFTITLE=S	8-byte representation with a 2-digit year and delimiters. Example: *yy-mm-dd*.
DFTITLE=L	10-byte representation with a 4-digit year and delimiters. Example: *yyyy-mm-dd*.
DFTITLE=I	8-byte representation with a 4-digit year and no delimiters. Example: *yyyymmdd*.

For each DFTITLE setting, the sequence of the day, month and year and the delimiter characters used are determined by the DTFORM parameter.

The DFTITLE parameter can be set in the Natural parameter module/file, dynamically when Natural is invoked, or at session level with the system command GLOBALS. It is evaluated at runtime.

Example:

This example assumes that DTFORM=I applies.

```
WRITE 'HELLO'
END
/*
/* Date in page title if DFTITLE=S is set ...: 14-01-31
/* Date in page title if DFTITLE=L is set ...: 2014-01-31
/* Date in page title if DFTITLE=I is set ...: 20140131
```

Note: The DFTITLE parameter has no effect on a user-defined page title as specified with a WRITE TITLE statement.

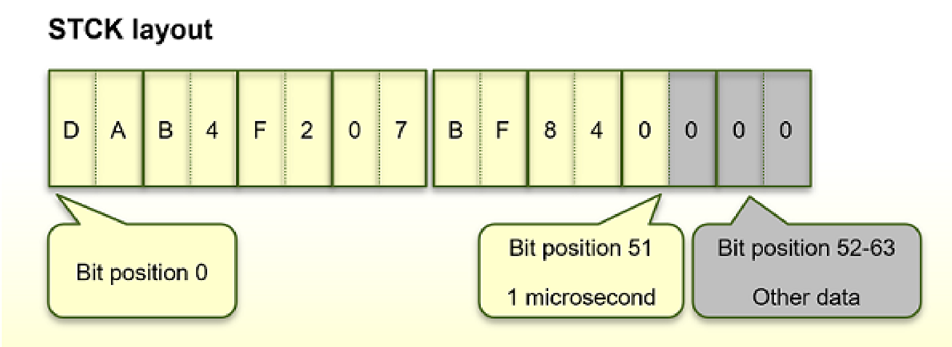
51 Processing of Store Clock Values

This document covers various aspects concerning the handling of store clock values in Natural applications.

The following topics are covered:

Original Store Clock and Year 2042 Issue

The Natural system variable *TIMESTMP provides the machine-internal store clock value as provided by the mainframe machine instruction STCK (Store Clock format). The store clock is an 8-byte binary field. It counts the microseconds since 1900-01-01 00:00:00 (UTC) whereby bit position 51 represents exactly one microsecond (bit position 0 is the leftmost bit). The remaining 12 bits can contain higher precision values such as nanoseconds or a processor ID, depending on the machine implementation. Store clock values are unique in a Natural session even if multiple store clock values are taken in one and the same microsecond.



The store clock will reach its highest possible value on 2042-09-17 23:53:47.370495 (UTC) at which time all 52 bits will be set to 1. Afterwards, the store clock will restart counting from 0. The date on which the clock is reset is referred to below as *wrap-date*.

After the wrap-date, the store clock will use the same values as in the range from 1900 to 2042. Therefore, it cannot be determined whether a store clock has been taken before or after the wrap-date.

If the store clock covers the years from 1900 to 2042, we call it the *original store clock*.

The year 2042 issue leads to the following impacts:

- **Compare and sort**
  Comparing time-related values should reflect the chronological sequence so that previous values are smaller than later values. Unfortunately, a store clock value taken just before the wrap-date is greater than a store clock value after that date when the store clock was reset to 0. Thus, comparing such store clock values will lead to incorrect results. For example, the store clock of 2040-01-01 (before the wrap-date) is greater than the store clock of 2043-01-01 (after the wrap-

date). If both the store clock values taken before and after the wrap-date are used for sorting, they will not correspond to the chronological sequence. A READ which uses the mentioned values as start and end values, will not return any value because the end value is smaller than the start value.

- **Conversion**
  If a store clock value is converted into a date, it is assumed that it was taken in the years from 1900 to 2042. If the store clock has actually been taken after the wrap-date, the conversion will therefore lead to an incorrect result. For example, a store clock value was taken on 2043-12-07 (after wrap-date). If this store clock is converted into a date, it will result in 1901-03-21.
- **Calculation of a time difference**
  If store clock values are converted into microseconds, they can be used to calculate a time difference. This works fine if the wrap-date is not involved. Subtracting a (big) store clock value taken before the wrap-date from a (small) store clock value taken after the wrap-date will result in an invalid (big negative) result. For example, the calculation of the store clock time difference between the years 2039 and 2043, will result in “-138 years”.

We denote the time while we process only store clock values smaller than the wrap-date (2042-09-17) as the transition period. During this time, the original store clock values should be replaced by one of the approaches described in the following sections.

How to Overcome the Year 2042 Issue

To overcome the year 2042 issue, there are three approaches:

- Temporary approach: *Store Clock with Sliding Window*
- General approach: *Extended Store Clock*
- Smart approach: *Smart Store Clock*

For these approaches, Software AG provides application programming interfaces (APIs) in the system library SYSEXT as described in the section *Natural APIs for Store Clock Processing*.

The *Store Clock with Sliding Window* interprets the store clock value in a sliding window. It supports the time range from 1971 to 2114. The binary values cannot be used for sorting or comparing.

The *Extended Store Clock* is 16 bytes long and supports the time range from 1900 to 38434. The binary values can be used for sorting and comparing. Extended store clock values are not compatible to store clock values (neither original nor with sliding window). Data and programs referring store clock values must be converted in one big step to extended store clock values.

The *Smart Store Clock* clock is 9 bytes long and supports the time range from 1900 to 38434. The binary values can be used for sorting and comparing. During the transition period, smart store clock values are compatible to original store clock values. Data and programs referring store clock values can be converted step by step to smart store clock values. This is the recommended approach to overcome the year 2042 issue.

Store Clock with Sliding Window

If a store clock value is converted into a date, the original store clock conversion assumes that it was taken in the range from 1900 to 2042. To extend the range, we use a *sliding window* which supports the range from 1971 to 2114. This implies that the years from 1900 to 1971 are no longer supported.

In the year 1971 the first bit in the store clock value switches to `1`. Therefore, this bit is used for the realization of the sliding window:

- From 1971 to 2042 the first bit is `1`. Store clock values of this range are interpreted as before.
- From 1900 to 1971 and from 2042 to 2114, the first bit is `0`. With the sliding window these store clock values are interpreted as 2042 – 2114.

When a store clock with sliding window is converted into microseconds and the first bit is `0`, the binary store clock value is extended at the left with `x'01'` before the conversion, so that the result always reflects the number of microseconds since 1900-01-01.

The range of the store clock with sliding window

- starts at 1971-05-11 11:56:53.685248 (UTC) and
- ends at 2114-01-26 11:50:41.055743 (UTC).

Note:

- If you sort 8-byte store clock values, they will not be in the chronological sequence even if you use the sliding window. For this task you must use smart or extended store clock values.
- If you compare 8-byte store clock values, the result will not reflect the chronological sequence even if you use the sliding window. You can use the copycode `USR9201D` for this task.
- If a store clock is taken after the end date of the sliding window, it will be wrongly interpreted. In this case you must also use smart or extended store clock values.
- A store clock value is saved in an 8-byte field in which the last 12 bits are never zero. This fact can be used to determine whether the field for the store clock is still unused. If all 8 bytes in the field are zero ("NULL value"), the field is still unused, and a store clock value was not yet saved. You must ensure that the store clock field does not contain a NULL value before using it in a conversion function.

Compared with the original store clock, the store clock with sliding window provides the following improvements:

- **Conversion**
  Natural APIs can be used to convert store clock values with sliding window correctly if store clock values have been taken from 1971 to 2114.
- **Calculation of a time difference**
  Natural APIs can be used to convert store clock values with sliding window into microseconds (since 1900-01-01). A difference calculation with these values provides the correct time duration (if the store clock values are from 1971 to 2114).
- **Outlook**
  Year 2042 issue is postponed until 2114.

Compared with the smart or extended store clock, the store clock with sliding window provides the following temporary advantages:

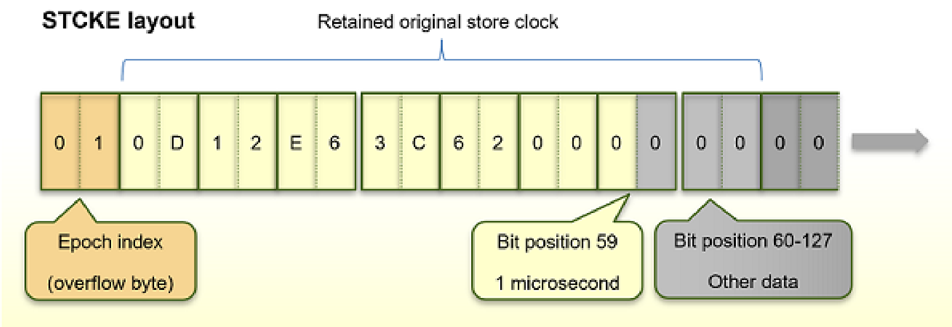
- **Size**
  The size of the store clock value remains at 8 bytes.
- **Data conversion (compared with extended store clock)**
  There is no need to migrate the store clock data to 16-byte values. Saved 8-byte store clock data will be interpreted correctly if it is in the range 1971-2114.
- **Program modifications**
  The conversion implies minor changes. The system variable *TIMESTMP can be used further on.

Before migrating original store clock to store clock with sliding window, consider the following:

- Further code changes will be required before the end of the sliding window in 2114.
- If your application calls a Natural API which supports the original store clock value, replace it by an API which supports the store clock value with sliding window.
- If you compare or sort binary store clock values, they will not be in the chronological sequence. See the note above.

Extended Store Clock

The Natural system variable *TIMESTMPX provides the machine-internal extended store clock value as provided by the mainframe machine instruction STCKE (Store Clock Extended format). The extended store clock is a 16-byte binary field. It counts the microseconds since 1900-01-01 00:00:00 (UTC) whereby bit position 59 represents exactly one microsecond (bit position `0` is the very left bit). The original store clock time representation is retained at byte 2-9. An extra overflow byte is added to the left, the so-called epoch index.



The extended store clock covers the years from 1900 to 38434.

Compared with the original store clock, the extended store clock provides the following improvements:

- **Compare and sort**
  The extended store clock always delivers the expected chronological sequence when you compare or sort values.
- **Conversion**
  Natural APIs can be used to convert extended store clock values. Because every extended store clock value in the extended store clock range is unique, the interpretation is always correct.
- **Calculation of a time difference**
  Natural APIs can be used to convert extended store clock values into microseconds (since 1900-01-01). A difference calculation with these values provides the correct time duration.
- **Outlook**
  No further code change is required in the future.

Before migrating from original store clock to extended store clock, consider the following:

- The size of the binary field increases from 8 bytes to 16 bytes.
- Extended store clock values are not compatible with (B8) store clock values. You cannot move a store clock value to an extended store clock value nor vice versa.

  Example:

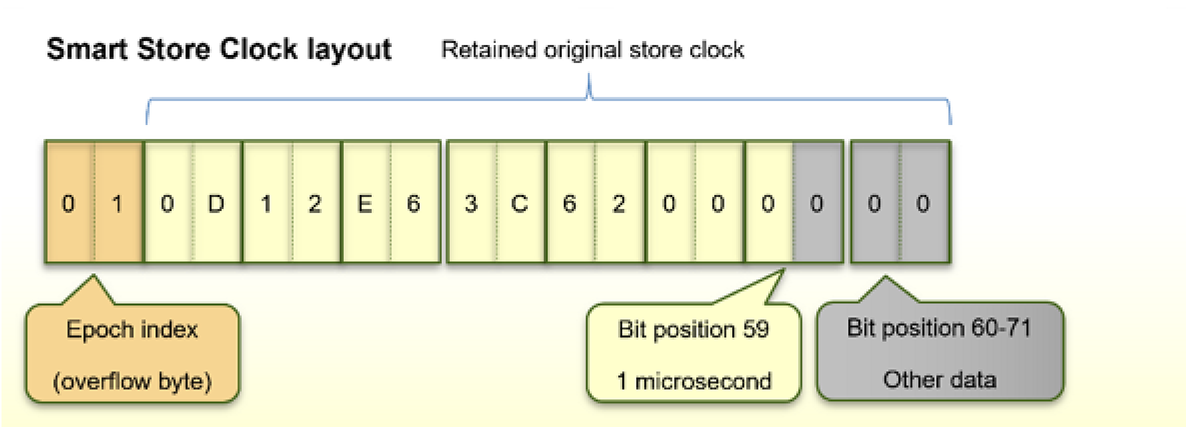
  ```
  #STORECLOCK: DD943485BC302002

  MOVE #STORECLOCK TO #EXTENDED

  Received: 0000000000000000DD943485BC302002
  Expected: 00DD943485BC302002000000000000000
  ```

- It is not possible to compare an extended store clock value directly against an original store clock value (nor against a *Store Clock with Sliding Window*). The extended store clock contains seven additional bytes added to the right and a comparison will not return a correct result.

- Saved store clock values must be converted into extended store clock values. You may use the copycode `USR9201R` for this task.
- If store clock values (B8) are saved in Adabas, you must edit the data so that the new (B16) field contains correct extended store clock values.
- Every program which processes store clock values must be adjusted to extended store clock values. Because store clock values and extended store clock values are not compatible, the changes must be done in one step.
- If your application calls a Natural API which supports the original store clock value, replace it by an API which supports the extended store clock value.
- This complete conversion implies major changes.
- It is recommended to use the smart store clock instead of the extended store clock. This is described in the following section.

Smart Store Clock

The smart store clock is a 9-byte binary field. It counts the microseconds since 1900-01-01 00:00:00 (UTC) whereby bit position 59 represents exactly one microsecond (bit position 0 is the very left bit). The original store clock time representation is retained at byte 2-9. An extra overflow byte is added to the left, the so-called epoch index. In other words, the smart store clock comprises the first 9 bytes of the extended store clock.



The smart store clock covers the years from 1900 to 38434.

Compared with the original store clock, the smart store clock provides the following improvements:

- **Compare and sort**
  The smart store clock always delivers the expected chronological sequence when you compare or sort values.
- **Conversion**
  Natural APIs can be used to convert smart store clock values. Because every smart store clock value in the smart store clock range is unique, the interpretation is always correct.
- **Calculation of a time difference**
  Natural APIs can be used to convert smart store clock values into microseconds (since 1900-01-01). A difference calculation with these values provides the correct time duration.
- **Outlook**
  No further code change is required in the future.

Before migrating from original store clock to smart store clock, consider the following:

- The size of the binary field increases from 8 bytes to 9 bytes
- During the transition period:
  - Increasing the length of an original store clock value from (B8) to (B9) gives the correct smart store clock value. This is because in Natural and Adabas, binary fields are right aligned. Increasing the length will fill the overflow byte with H’00’.
  - Smart store clock values are compatible with original store clock values. You can move an original store clock value to a smart store clock value and vice versa.

    Example:

    ```
    #STORECLOCK: DD943485BC302002

    MOVE #STORECLOCK TO #SMART

    Received: 00DD943485BC302002
    Expected: 00DD943485BC302002
    ```

  - Smart store clock values can be compared with original store clock values.
- Every program which processes store clock values must be adjusted to smart store clock values. Because store clock values and smart store clock values are compatible, the changes can be done step by step.
- If your application calls a Natural API which supports the original store clock value, replace it by an API which supports the smart store clock value.
- This complete conversion implies minor changes.

Transition to Smart Store Clock

Data Definition in Natural

A field containing store clock values is defined as

```
1 #TIMESTMP (B8)   /* original store clock value
```

Replace the definition by

```
1 #TIMESTMP (B9)   /* smart store clock value
```

Receive Current Store Clock Value

To receive the current store clock value, the following statement was executed

```
MOVE *TIMESTMP TO #TIMESTMP   /* get current timestamp
```

During the transition period, the statement can still be used even if the length of `#TIMESTMP` has increased to (B9). On the long term, the statement should be replaced by

```
INCLUDE USR9201F '*TIMESTMPX' '#TIMESTMP' /* get current timestamp
```

Alternatively, you can use a redefinition:

```
1 #TIMESTMPX (B16)   /* extended store clock value
1 REDEFINE #TIMESTMPX
  2 #TIMESTMP (B9)   /* smart store clock value
```

And use the following statement:

```
MOVE *TIMESTMPX TO #TIMESTMPX /* get current timestamp
```

Move Store Clock Values

During the transition period, store clock values can be moved from one to the other even if the length of one is (B9) and the other is still (B8).

```
MOVE #TIMESTMP-A TO #TIMESTMP-B   /* move store clock value
```

Calling a Subprogram

A subprogram uses a store clock value as parameter

```
1 #TIMESTMP (B8)   /* store clock value
```

If you increase the length of the field to (B9) and there are programs which call it with (B8), the program will fail. Use in this case the following definition

```
1 #TIMESTMP (B9) BY VALUE RESULT  /* store clock value
```

As soon as all callers use (B9) store clock values, the `BY VALUE RESULT` can be removed.

Compare Store Clock Values

During the transition period, store clock values can be compared with each other even if the length of one is (B9) and the other is still (B8).

```
If #TIMESTMP-A > #TIMESTMP-B   /* compare store clock values
```

Field Definition in Adabas

A field containing store clock values is defined as

```
1,AA,8,B,...
```

Replace the definition by

```
1,AA,9,B,...
```

On the mainframe, you can achieve it with the Adabas Online Service or the ADADBS (Database Services) utility; on Linux and Windows, with the ADAFDU (File Definition) utility. Note that on Linux and Windows, a field containing store clock values must be defined with the HF option (high order first).

With the change of the length, the database contains valid smart store clock values. There is no need to edit the data. If the field is a descriptor, it will provide the correct chronological sequence.

On the Natural side, you must update the DDM accordingly, and if a length is specified, also the related views in the data definitions.

Superdescriptor Definition in Adabas

A field AA containing store clock values is used as parent field of a superdescriptor:

```
SA=AC(1,20),AA(1,8)
```

During the transition period, the superdescriptor can still be used even if the length of the parent field has increased to 9 bytes. This is because the bytes in a binary field are counted from right to left.

On the long term, the definition should be replaced by

```
SA=AC(1,20),AA(1,9)
```

Every variable in Natural programs (like start and end values) referring the superdescriptor, must be adjusted together with the database changes. Alternatively, you can keep the original superdescriptor and create additionally a new superdescriptor with the new length. Then you can switch one program after the other to the new superdescriptor. Once all programs access the new superdescriptor, the old one can be released (deleted).

Local Store Clock

The Natural system variable *TIMX provides the local time in tenth of seconds. If the local time is required in a higher precision, the Natural API `USR9178N` can be used. It provides the local time and the corresponding UTC time in store clock format (B8), in microseconds since 1900-01-01 and in the Natural time (T) format. Additionally, it provides the time differential, a standard `DATETIME` value and the time zone. Functions are available to convert the values into other formats. The copycodes `USR9178X`, `USR9178Y` and `USR9178Z` provide mainly the same functionality as the subprogram `USR9178N` with an essentially better performance.

Layout of a Local Store Clock Value

The first 7 bytes in the local store clock value have the same content as the original store clock whereby bit positions 0 – 51 contain the microseconds of the local time. Therefore, standard APIs (like `USR1023`* or `USR9201`*) can be used for the interpretation of a local store clock value.

The last byte of a local store clock value generated by `USR9178`*, contains the time differential. With this information it can later be determined in which time zone the local store clock was taken, especially whether it has been taken in the DST (daylight saving time).

Time Differential and Time Zone

The time differential reflects the following difference:

```
Time differential = Local time - UTC time
```

The API `USR9178N` and the corresponding copycodes provide the time differential in units of 1/10 seconds.

The last byte of a local store clock value contains the time differential in units of 15 min as signed integer (format I1).

The API `USR9178N` also returns the time zone related to the time differential:

- `-hh:ii` for a negative time differential, or
- `+hh:ii` otherwise.

Standard `DATETIME`

The API `USR9178N` returns the standard `DATETIME` value:

`yyyy-mm-dd``T`*`hh`*`:`*`ii`*`:`*`ss.fff`*

where *`fff`* is the time fraction in milliseconds.

Natural APIs for Store Clock Processing

The following Natural Application Programming Interfaces (APIs) for processing of store clock values are provided in the Natural library SYSEXT:

Natural API	Function	Store Clock			Previous Version
		Original	Sliding Window	Smart / Extended	
USR1009N	Convert store clock with sliding window into microseconds	Yes	Yes	No	
USR1023N	Convert time-related variables	Yes	No	No	
USR9178N	Maintain local store clock value	No	Yes	No	
USR9201N	Convert time-related variables	No	Yes	Yes	USR1023N

USR1009N

The API converts a store clock value into microseconds since 1900-01-01. The store clock value is interpreted by default without sliding window (range 1900-2042). An optional parameter is available to interpret the store clock value with the sliding window (range 1971-2114).

It is recommended to use the copycode `USR9201Y` (with sliding window) or `USR1023Y` (without sliding window) instead of the API `USR1009N` for better performance.

USR1023N

The API converts a time-related variable into other formats. The API uses the original store clock (1900 – 2042).

It is strongly recommended to use the API `USR9201N` instead of `USR1023N`. Otherwise, store clock values will be incorrectly interpreted after the year 2042. If you take, for example, a store clock on 2043-12-07, the API `USR1023N` will interpret it as 1901-03-21.

The API `USR1023N` expects one of the following formats as input:

- Natural date and time,
- Natural date,
- microseconds (since 1900-01-01), or
- original store clock,

and converts it into all other formats.

Additional to the API, the following copycodes are provided:

- `USR1023W` - Convert microseconds into original store clock
- `USR1023X` - Convert microseconds into Natural time
- `USR1023Y` - Convert original store clock into microseconds
- `USR1023Z` - Convert Natural time into microseconds

The copycodes are essentially faster than the subprogram `USR1023N`.

USR9178N

The API maintains local store clock values. It supports store clock with sliding window (1971 - 2114).

For function “C”, it returns the current values of:

- Local store clock
- UTC store clock

- Time differential in 1/10 seconds
- Local time in microseconds
- UTC time in microseconds
- Local time in Natural (T) format
- UTC time in Natural (T) format
- Standard DATETIME
- Time zone

For function “L”, it converts a local store clock value into the other formats.

For function “U”, it converts a UTC store clock value and a time differential into the other formats.

Additional to the API, the following copycodes are provided:

- USR9178X - Get current local store clock, UTC store clock and time differential.
- USR9178Y - Convert local store clock into UTC store clock and time differential.
- USR9178Z - Convert UTC store clock and time differential into local store clock.

USR9201N

The API is the successor of `USR1023N`. It supports store clock with sliding window (1971 - 2114), smart and extended store clock (1900 - 38434).

The API expects one of the following formats as input:

- Natural date and time,
- Natural time,
- Natural date,
- microseconds (since 1900-01-01),
- store clock with sliding window,
- extended store clock, or
- smart store clock,

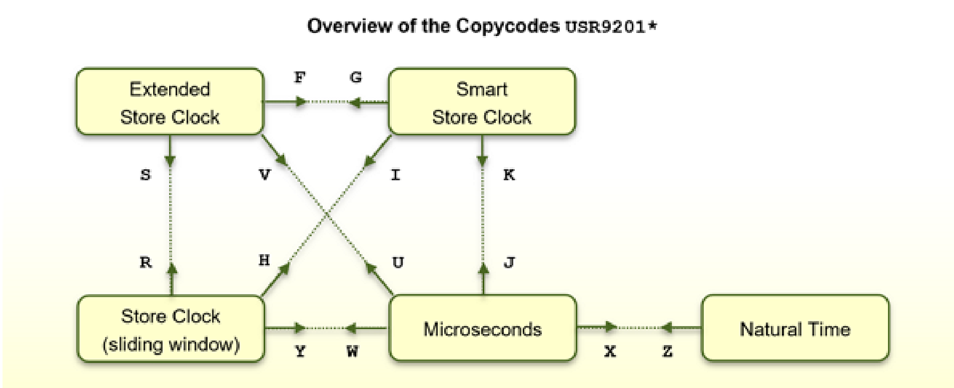
and converts it into all other formats.

Additional to the API, the following copycodes are provided:

- `USR9201D` - Calculate time difference of two store clock values with sliding window
- `USR9201F` - Convert extended store clock into smart store clock
- `USR9201G` - Convert smart store clock into extended store clock
- `USR9201H` - Convert store clock with sliding window into smart store clock

- `USR9201I` - Convert smart store clock into store clock with sliding window
- `USR9201J` - Convert microseconds into smart store clock
- `USR9201K` - Convert smart store clock into microseconds
- `USR9201R` - Convert store clock with sliding window into extended store clock
- `USR9201S` - Convert extended store clock into store clock with sliding window
- `USR9201U` - Convert microseconds into extended store clock
- `USR9201V` - Convert extended store clock into microseconds
- `USR9201W` - Convert microseconds into store clock with sliding window
- `USR9201X` - Convert microseconds into Natural time
- `USR9201Y` - Convert store clock with sliding window into microseconds
- `USR9201Z` - Convert Natural time into microseconds

Note: The copycode `USR9201D` can also be used to compare two store clock values which have been taken in the sliding window range (1971-2114).



The copycodes are essentially faster than the subprogram `USR9201N`.

Example - Convert a Store Clock with Sliding Window into Microseconds

A test for 10,000 conversions shows that:

- the subprogram `USR9201N` requires about 35 ms, and
- the copycode `USR9201Y` requires about 5 ms.

52 End of Statement, Program or Application

End of Statement

To explicitly mark the end of a statement, you can place a semicolon (;) between the statement and the next statement. This can be used to make the program structure clearer, but is not required.

End of Program

The `END` statement is used to mark the end of a Natural program, function, subprogram, external subroutine or helproutine.

Every one of these objects must contain an `END` statement as the last statement.

Every object may contain only one `END` statement.

End of Application

Ending the Execution of an Application by a STOP Statement

The `STOP` statement is used to terminate the execution of a Natural application. A `STOP` statement executed anywhere within an application immediately stops the execution of the entire application.

Ending the Execution of an Application by a TERMINATE Statement

The `TERMINATE` statement stops the execution of the Natural application and also ends the Natural session.

Interrupting a Running Natural Application

During the development of a Natural application and in test situations, the user should be able to interrupt a running Natural application that does not respond anymore, for example, due to an endless loop. As the Natural session should not need to be killed, the running Natural application can be interrupted via the typical system interrupt key combination (for example, CTRL+BREAK for Windows or CTRL+C for Linux). The Natural error NAT1016 is raised and the runtime error processing is activated. The error can be handled by an `ON ERROR` processing.

In a production environment, this feature will typically need to be disabled, because the application may not be able to recover from a user interrupt at an arbitrary program location.

The Natural profile parameter `RTINT` determines whether interrupts are allowed. By default, interrupts are not allowed.

If this parameter is set to `ON`, a running Natural application may be interrupted with the interrupt key combination of the operating system (for example, for Windows: CTRL+BREAK; for Linux: typically CTRL+C, but can be reconfigured using the `stty` command).

Natural catches this interrupt request and then offers the user the following possibilities:

- Perform standard error processing by raising a NAT1016 error.
- Continue application processing (cancel interrupt).

The choice is shown in a window that is opened after catching the interrupt signal.

53 Processing of Application Errors

This section discusses the two basic methods Natural offers for the handling of application errors: default processing and application-specific processing. Furthermore, it describes the options you have to enable the application specific error processing: coding an `ON ERROR` statement block within a Natural object or using a separate error transaction program.

Finally, this section gives an overview of the features that are provided to configure Natural's error processing behavior, to retrieve information on an error, to process or debug an application error.

For information on error handling in a Natural RPC environment, see *Handling Errors* in the *Natural RPC (Remote Procedure Call)* documentation.

Natural's Default Error Processing

When an error occurs in a Natural application, Natural will by default proceed in the following way:

1. Natural terminates the execution of the currently running application object;
2. Natural issues an error message;
3. Natural returns to command input mode.

“Command input mode” means that, depending on your Natural configuration, the Natural main menu, the `NEXT` command prompt, or a user-defined startup menu may appear.

The displayed error message contains the Natural error number, the corresponding message text and the affected Natural object and line number where the error has occurred.

Because after the occurrence of an error the execution of the affected application object is terminated, the status of any pending database transactions may be affected by actions required by the setting of the profile parameter `ETEOP`. Unless Natural has issued an `END TRANSACTION` statement as a result of the settings of these parameters, a `BACKOUT TRANSACTION` statement is issued when Natural returns to command input mode.

Application Specific Error Processing

Natural enables you to adapt the error processing if the default error processing does not meet your application’s requirements. Possible reasons to establish an application specific error processing may be:

- The information on the error is to be stored for further analysis by the application developer.
- The application execution is to be continued after error recovery, if possible.
- A specific transaction handling is necessary.

Because the execution of the affected Natural application object is terminated after an application error has occurred, the status of the pending database transactions may be influenced by actions which are triggered by the settings of the profile parameter `ETEOP`. Therefore, further transaction handling (`END TRANSACTION` or `BACKOUT TRANSACTION` statement) has to be performed by the application's error processing.

To enable the application specific error processing, you have the following options:

- You may code an `ON ERROR` statement block within a Natural object.
- You may use a separate error transaction program.

These options are described in the following sections.

Using an ON ERROR Statement Block

You may use the `ON ERROR` statement to intercept execution time errors within the application where an error occurs.

From within an `ON ERROR` statement block, it is possible to resume application execution on the current level or on a superior level.

Moreover, you may specify an `ON ERROR` statement in multiple objects of an application in order to process any errors that have occurred on subordinate levels. Thus, application specific error processing may exactly be tailored to the application's needs.

Exiting from an `ON ERROR` Statement Block

You may exit from an `ON ERROR` statement block by specifying one of the following statements:

- `RETRY`

  Application execution is resumed on the current level.
- `ESCAPE ROUTINE`

  Error processing is assumed to be complete and application execution is resumed on the superior level.
- `FETCH`

  Error processing is assumed to be complete and the "fetched" program is executed.

  `STOP`

  Natural stops the execution of the affected program, ends the application and returns to command input mode.

- TERMINATE

  The execution of the Natural application is stopped and also the Natural session is terminated.

Error Processing Rules

- If the execution of the ON ERROR statement block is not terminated by one of these statements, the error is percolated to the Natural object on the superior level for processing by an ON ERROR statement block that exists there.
- If none of the Natural objects on any of the superior levels contains an ON ERROR statement block, but if an error transaction program has been specified (as described in the **next section**), this error transaction program will receive control.
- If none of the Natural objects on any of the superior levels contains an ON ERROR statement block and no error transaction program has been specified there, Natural's default error processing will be performed as described **above**.

Using an Error Transaction Program

You may specify an error transaction program in the following places:

- In the profile parameter ETA.
- If Natural Security is installed, within the Natural Security library profile; see *Components of a Library Profile* in the *Natural Security* documentation.
- Within a Natural object by assigning the name of the program which is to receive control in the event of an error condition as a value to the system variable *ERROR-TA, using an ASSIGN, COMPUTE or MOVE statement.

If you assign the name of an error transaction program to the system variable *ERROR-TA during the Natural session, this assignment supersedes an error transaction program specified using the profile parameter ETA. Regardless of whether you use the ETA profile parameter or assign a value to the system variable *ERROR-TA, the error transaction program names are not saved and restored by Natural for different levels of the call hierarchy. Therefore, if you assign a name to the system variable *ERROR-TA in a Natural object, the specified program will be invoked to process any error that occurs in the current Natural session after the assignment.

On the one hand, if you specify an error transaction program by using the profile parameter ETA, an error transaction is defined for the complete Natural session without having the need for individual assignments in Natural objects. On the other hand, the method of assigning a program to the system variable *ERROR-TA provides more flexibility and, for example, allows you to have different error transaction programs in different application branches.

If the system variable *ERROR-TA is reset to blank, Natural's default error processing will be performed as described **above**.

If an error transaction program is specified and an application error occurs, execution of the application is terminated, and the specified error transaction program receives control to perform the following actions:

- Analyze the error;
- Log the error information;
- Terminate the Natural session;
- Continue the application execution by calling a program using the `FETCH` statement.

Because the error transaction program receives control in the same way as if it had been called from the command prompt, it is not possible to resume application execution in one of the Natural objects that were active at the time when the error occurred.

If a syntax error occurs and the Natural profile parameter `SYNERR` is set to `ON`, the error transaction program will also receive control.

An error transaction program must be located in the library to which you are currently logged on or in a current steplib library.

When an error occurs, Natural executes a `STACK TOP DATA` statement and places the following information at the top of the stack:

<table>
<tr><th>Stack Data</th><th>Format/Length</th><th colspan="2">Description</th></tr>
<tr><td>Error number</td><td>N4</td><td colspan="2">Natural error number.

Note: If session parameter <code>SG</code> is set to <code>ON</code>, the format/length will be N5.</td></tr>
<tr><td>Line number</td><td>N4</td><td colspan="2">Number of the line where the error has occurred.

If the status is <code>C</code> or <code>L</code>, the line number will be zero.</td></tr>
<tr><td rowspan="6">Status</td><td rowspan="6">A1</td><td colspan="2">Status code:</td></tr>
<tr><td><code>C</code></td><td>Command processing error</td></tr>
<tr><td><code>L</code></td><td>Logon processing error</td></tr>
<tr><td><code>O</code></td><td>Object (execution) time error</td></tr>
<tr><td><code>R</code></td><td>Error on remote server (in conjunction with Natural RPC)</td></tr>
<tr><td><code>S</code></td><td>Syntax error</td></tr>
<tr><td>Object name</td><td>A8</td><td colspan="2">Name of the Natural object where the error has occurred.</td></tr>
<tr><td>Level number</td><td>N2</td><td colspan="2">Level number of the Natural object where the error has occurred.

If a Natural syntax error occurs at compile time and profile parameter <code>SYNERR</code> is set to <code>ON</code>, the level number will be zero.

If a Natural runtime error occurs and the level number of the Natural object is greater than <code>99</code>, the value <code>99</code> will be stacked, and the current</td></tr>
</table>

Stack Data	Format/Length	Description
		value will be stacked in the additional stack data “Level number enhanced”.
If a Natural runtime error occurs and the level number of the Natural object is greater than 99:		
Level number enhanced	I4	Current level number (512 at maximum).
If a Natural syntax error occurs at compile time and profile parameter SYNERR is set to ON:		
Error position	N3	Position of the offending item in the source line.
Item length	N3	Length of the offending item.

This information can be retrieved in the error transaction program, using an INPUT statement.

Example:

```
DEFINE DATA LOCAL
1 #ERROR-NR          (N5)
1 #LINE              (N4)
1 #STATUS-CODE       (A1)
1 #PROGRAM           (A8)
1 #LEVEL             (N2)
1 #LEVELI4           (I4)
1 #POSITION-IN-LINE  (N3)
1 #LENGTH-OF-ITEM    (N3)
END-DEFINE
IF *DATA > 6 THEN       /* SYNERR = ON and a syntax error occurred
  INPUT
    #ERROR-NR
    #LINE
    #STATUS-CODE
    #PROGRAM
    #LEVEL
    #POSITION-IN-LINE
    #LENGTH-OF-ITEM
ELSE
  INPUT                 /* other error
    #ERROR-NR
    #LINE
    #STATUS-CODE
    #PROGRAM
    #LEVEL
    #LEVELI4
END-IF
WRITE #STATUS-CODE
* DECIDE ON FIRST VALUE OF STATUS-CODE
*  ... /* process error
* END-DECIDE
END
```

Some of the information placed on top of the stack is equivalent to the contents of several system variables that are available in an `ON ERROR` statement block:

Stack Data	Equivalent System Variable in ON ERROR Statement Block
Error number	`*ERROR-NR`
Line number	`*ERROR-LINE`
Object name	`*PROGRAM`
Level number	`*LEVEL`

Rules under Natural Security

If Natural Security is installed, the additional rules for the processing of logon errors apply. For further information, see *Transactions* in the *Natural Security* documentation.

Error Processing Related Features

Natural provides a variety of error processing related features that

- Enable you to configure Natural's error processing behavior;
- Help you in retrieving information about errors that have occurred;
- Support you in processing these errors;
- Support you in debugging application errors.

These features can be grouped as follows:

- Profile parameters
- System variables
- Terminal commands
- System commands
- Application programming interfaces

Profile Parameters

The following profile parameters have an influence on the behavior of Natural in the event of an error:

Profile Parameter	Purpose
CPCVERR	Code Page Conversion Error
DU	Dump generation after abnormal termination
CC	Error processing in batch mode
ETA	Error Transaction Program
ETEOP	Issue END TRANSACTION at End of Program
MADIO	Maximum DBMS calls between screen I/O operations
MAXCL	Maximum number of program calls
RCFIND	Handling of Response Code 113 for FIND Statement
RCGET	Handling of Response Code 113 for GET Statement
SYNERR	Control of Syntax Errors
ZD	Zero-division check

System Variables

The following application related system variables can be used to locate an error or to obtain/specify the name of the program which is to receive control in the event of an error condition:

System Variable	Content
*ERROR-LINE	Source-code line number of the statement that caused an error. See **Example 1**.
*ERROR-NR	Error number of the error which caused an ON ERROR condition to be entered.
*ERROR-TA	Name of the program which is to receive control in the event of an error condition. See **Example 2**.
*LEVEL	Level number of the Natural object where the error has occurred.
*LIBRARY-ID	Name of the library to which the user is currently logged on.
*PROGRAM	Name of the Natural object that is currently being executed. See **Example 1**.

Example 1:

```
...
    /*
    ON ERROR
      IF *ERROR-NR = 3009 THEN
        WRITE 'LAST TRANSACTION NOT SUCCESSFUL'
            / 'HIT ENTER TO RESTART PROGRAM'
        FETCH 'ONEEX1'
      END-IF
      WRITE 'ERROR' *ERROR-NR 'OCCURRED IN PROGRAM' *PROGRAM
            'AT LINE' *ERROR-LINE
```

```
      FETCH 'MENU'
    END-ERROR
    /*
...
```

Example 2:

```
...
    *ERROR-TA := 'ERRORTA1'
    /* from now on, program ERRORTA1 will be invoked
    /* to process application errors
...
    MOVE 'ERRORTA2' TO *ERROR-TA
    /* change error transaction program to ERRORTA2
...
```

For further information on these system variables, see the corresponding sections in the *System Variables* documentation.

Terminal Commands

The following terminal command has an influence on the behavior of Natural in the event of an error:

Terminal Command	Purpose
`%E=`	Activate/Deactivate Error Processing

System Commands

The following system commands provide additional information on an error situation or invoke the utilities for debugging or logging database calls:

System Command	Purpose
`LASTMSG`	Display additional information on the error situation which has occurred last.
`RPCERR`	Only applies in a Natural RPC (Remote Procedure Call) environment. Display Natural, EntireX Broker and EntireX RPC server errors that last occurred during an RPC session. For more information, see *Using the RPCERR Program* in the *Natural RPC (Remote Procedure Call)* documentation.
`TECH`	Display technical and other information about your Natural session, for example, information on the last error that occurred.

Application Programming Interfaces

The following application programming interfaces (APIs) are generally available for getting additional information on an error situation or to install an error transaction.

API	Purpose
`RPCINFO`	Only applies in a Natural RPC (Remote Procedure Call) environment. This subprogram retrieves Natural, EntireX Broker and EntireX RPC server errors that last occurred during an RPC session. For more information, see *Using the RPCINFO Subprogram* in the *Natural RPC (Remote Procedure Call)* documentation.
`USR0040N`	Get type of last error
`USR0622N`	Reset error counter in `ON ERROR` statement block
`USR1016N`	Get error level for error in nested copycodes
`USR2001N`	Get information on last error
`USR2006N`	Get information from error message collector
`USR2007N`	Get or set data for RPC default server
`USR2010N`	Get error information on last database call
`USR2026N`	Get `TECH` information
`USR2030N`	Get dynamic error message parts from the last error
`USR3320N`	Find user short error message (including steplibs search)
`USR4012N`	Set application error on RPC server
`USR4214N`	Get program level information

For further information, see *SYSEXT - Natural Application Programming Interfaces* in the *Utilities* documentation.

54 Invoking Natural Subprograms from 3GL Programs

Natural subprograms can be invoked from a Natural object written in a 3rd generation programming language (3GL). The invoking program can be written in any programming language that supports a standard `CALL` interface.

For this purpose, Natural provides the interface `ncxr_callnat`. The 3GL program invokes this interface with a specification of the name of the desired subprogram.

Note: Natural must have been activated beforehand; that is, the invoking 3GL program must in turn have been invoked by a Natural object with a `CALL` statement.

The subprogram is executed as if it had been invoked from another Natural object with a `CALLNAT` statement.

When the processing of the subprogram stops (either with the `END` statement or with an `ESCAPE ROUTINE` statement), control is returned to the 3GL program.

Passing Parameters from the 3GL Program to the Subprogram

Parameters can be passed from the invoking 3GL program to the Natural subprogram. For passing parameters, the same rules apply as for passing parameters with a `CALL` statement.

The 3GL program invokes the Natural interface `ncxr_callnat` with four parameters:

- The 1st parameter is the name of the Natural subprogram to be invoked.
- The 2nd parameter contains the number of parameters to be passed to the subprogram.
- The 3rd parameter contains the address of the table that contains the addresses of the parameters to be passed to the subprogram.
- The 4th parameter contains the address of the table that contains the format/length specifications of the parameters to be passed to the subprogram.

The sequence, format and length of the parameters in the invoking program must match exactly the sequence, format and length of the fields in the `DEFINE DATA PARAMETER` statement of the subprogram. The names of the fields in the invoking program and the invoked subprogram can be different.

Example of Invoking a Natural Subprogram from a 3GL Program

For an example of how to invoke a Natural subprogram from a 3GL program, refer to the following samples in `<install-dir>`*/natural/samples/sysexuex*.

- `MY3GL.NSP` (for the main program),
- `MY3GLSUB.NSN` (for the subprogram),
- `MYC3GL.C` (for the `C` function).

55 Issuing Operating System Commands from within a Natural Program

The Natural user exit `SHCMD` can be used to issue an operating system command, call a shell script or execute an executable program on Linux from within a Natural program.

Syntax

`CALL 'SHCMD' '`*`command`*`' ['`*`option`*`']`

Parameters

command	Command to be executed under control of the operating system command shell. Natural waits until the command is completed and then Natural returns control back to the Natural program. For more information, see ***Examples*** below.
option	*option* describes how the command should be executed. This parameter is optional. The following options are available: ■ `SCREENIO` ■ `NOSCREENIO` See *Parameter Options* below.

Parameter Options

The following parameter options are available:

Option	Description
`NOSCREENIO`	This option is used to hide the output generated by the command. The hidden output is redirected to the null device.
`SCREENIO`	This option is used to refresh the Natural screen output after the command is completed.

Note: The options `SCREENIO` and `NOSCREENIO` may be not set at the same time.

Return Codes

The following return code values are available:

Return Code	Description
0	Command successfully executed.
4	Illegal SHCMD parameter specified.
All other codes	Command shell return code.

Examples

Execute a command shell from within Natural:

```
CALL 'SHCMD' 'myshell.sh'
```

Execute an executable program from within Natural:

```
CALL 'SHCMD' 'myprogram'
```

Execute the operating system command `ls` on Linux to list the contents of a directory:

```
CALL 'SHCMD' 'ls'
```

After executing the `ls` command, you will recognize that the output generated by this command has changed the last Natural screen output. You have to press the refresh-screen key to clear the screen. To do this automatically, you can specify the `SCREENIO` option:

```
CALL 'SHCMD' 'ls' 'SCREENIO'
```

Retrieve the return code by using the `RET` function:

```
DEFINE DATA LOCAL
  1 rc (I4)
END-DEFINE
CALL 'SHCMD' 'lsDIRECTORY' 'SCREENIO'
ASSIGN rc = RET( 'SHCMD' )             /* retrieve return code
IF rc <> 0 THEN
   IF rc = 4 THEN
     WRITE NOTITLE 'Illegal option specified'
   ELSE
     WRITE NOTITLE 'Command not executed successfully (rc=' rc ')'
   END-IF
 ELSE
```

```
     WRITE NOTITLE 'Command executed successfully'
END-IF
END
```

VIII Statements for Internet Access and Parsing

56 Statements for Internet Access and Parsing

This chapter gives a functional overview of the Natural statements for internet access, PARSE JSON, and PARSE XML, specifies the general prerequisites for using these statements in a mainframe environment, informs about restrictions that apply and contains a list of further references. To take full advantage of these statements, a thorough knowledge of the underlying communication standards is required.

Statements Available

The following Natural statements are available for access to the internet and to JSON and XML documents:

- REQUEST DOCUMENT
- PARSE XML
- PARSE JSON

REQUEST DOCUMENT

- Functionality
- Example
- Syntax

Functionality

This statement enables you to use the Hypertext Transfer Protocol (HTTP) and the Hypertext Transfer Protocol Secure (HTTPS) in order to access documents on the web with a given Uniform Resource Identifier (URI) or Uniform Resource Locator (URL), that is, the internet or intranet address of a web site.

REQUEST DOCUMENT implements an HTTP client at Natural statement level, which allows applications to access any HTTP server on either the intranet or the internet. The statement has a set of operands, which allows it to formulate HTTP requests according to the needs of the user application. For example, using outbound operands it is possible to send user-defined HTTP headers, form data, or entire documents to an HTTP server. The inbound operands can be used to retrieve a document from the server, to view the entire HTTP header block returned from the server, or to return the values of dedicated headers, etc. Via binary format operands, binary objects such as gif files can be exchanged with the HTTP server as well. For basic authorization purposes, user ID and password operands can be specified. The content of this operand is sent with base64 encoding over the line, according to HTTP standards.

Natural supports the following REQUEST-METHODs:

- GET - retrieve a document and HTTP headers,
- HEAD - retrieve HTTP headers only,

- `POST` - transfer form data to an HTTP server, and
- `PUT` - upload a file to an HTTP server.

The `REQUEST-METHOD` is normally evaluated automatically, based on the operands coded for the executed `REQUEST DOCUMENT` statement. However, the predetermined `REQUEST-METHOD` can be overwritten by an explicit user specification of a `REQUEST-METHOD` header.

In addition to the standard `REQUEST-METHOD`s mentioned above, the following methods can be specified in a `REQUEST-METHOD` header:

- `DELETE` - delete a document from an HTTP server,
- `PATCH` - modify a document on an HTTP server,
- `OPTIONS` - retrieve the `REQUEST-METHOD`s supported by an HTTP server, and
- `TRACE` - retrieve the message received by an HTTP server.

Example

The following is an example of how the `REQUEST DOCUMENT` statement can be used to access an externally-located document:

```
REQUEST DOCUMENT FROM
"http://bolsap1:5555/invoke/sap.demo/handle_RFC_XML_POST"
WITH
USER #User PASSWORD #Password
DATA
NAME 'XMLData' VALUE #Queryxml
NAME 'repServerName' VALUE 'NT2'
RETURN
PAGE #Resultxml
RESPONSE #rc ↩
```

Syntax

The syntax of the `REQUEST DOCUMENT` statement and detailed application hints are to be found in the *Statements* documentation.

PARSE XML

Functionality

The `PARSE XML` statement allows you to parse XML documents from within a Natural program.

The `PARSE XML` statement integrates a full XML parser into Natural, thus allowing Natural applications to parse XML documents in order to easily process their content. The `PARSE XML` statement opens a processing loop and returns, whenever one of a list of events occurs during the parse process, the respective path through the document, name and value of parsed elements together with some parser status system variables.

Syntax

The syntax of the `PARSE XML` statement and detailed application hints are to be found in the *Statements* documentation.

PARSE JSON

Functionality

The `PARSE JSON` statement enables parsing of JSON documents within a Natural program.

The PARSE JSON statement incorporates a complete JSON parser into Natural, which enables Natural applications to parse JSON documents. Upon executing the `PARSE JSON` statement, it triggers a processing loop and produces the path, name, and value of the parsed elements, along with specific parser status system variables.

Technical Implementation

When it comes to parsing JSON documents, the following parsing strategies or models are commonly used:

- DOM (Document Object Model) – an object oriented approach
- SAX (Simple Access) – a stream-oriented parsing method

The implementation of the `PARSE JSON` statement in Natural for Linux and Cloud or Natural for Windows is based on the SAX method.

Internally, parsing is conducted on a UTF-8 encoded representation of the document being parsed. If the document is not provided in this encoding, an internal conversion to UTF-8 is carried out prior to parsing. The system automatically verifies the encoding of the document to be parsed. The criteria for this validation and conversion to UTF-8 are based on the following guidelines:

1. The system checks for a BOM (Byte Order Mark), which marks the encoding of the document.
2. If no BOM is found, then `ENCODED [IN] CODEPAGE` (operand2) is considered.
3. If no `ENCODED [IN] CODEPAGE` (operand2) is provided then the following guidelines are applied:
   - If the Data Type of the JSON Document (operand1) is specified as 'B' (Binary), then it is be treated as UTF-8 and no code page conversion is carried out.
   - If the Data Type of the JSON Document (operand1) is specified as 'U' (Unicode), then it is treated as UTF-16 and code page conversion to UTF-8 is carried out.
   - If the Data Type of the JSON Document (operand1) is specified as 'A' (Alphanumeric), the code page for the JSON Document (operand1) is identified based on the following criteria for converting the JSON Document (operand1) into UTF-8:
     - The default code page is used when the Natural parameter CP is defined. The default code page can be verified by the system variable `*CODEPAGE`
     - If a default code page is not available, then the error message NAT1328 is issued.

Note: Nested parse loops are not permitted in Natural for Linux and Cloud or Natural for Windows.

Syntax

You can find the syntax of the `PARSE JSON` statement and detailed application hints in the *Statements* documentation.

Further References

Below is a list of resources that you may find useful.

- Useful Links

Useful Links

Below is a collection of links that may be of interest.

- World Wide Web Consortium (W3C): ***http://www.w3.org/***
- Extensible Markup Language (XML): ***http://www.w3.org/XML/***
- HyperText Markup Language (HTML) Home Page: ***http://www.w3.org/MarkUp/***
- W3 Schools: ***https://www.w3schools.com/***

IX Portable Natural Generated Programs

57 Portable Natural Generated Programs

As of Natural Version 5, Natural generated programs (GPs) are portable across UNIX, OpenVMS and Windows platforms.

Compatibility

As of Natural Version 5, a source which was cataloged on any Natural-supported UNIX, OpenVMS and Windows platform is executable with all of these Open Systems platforms without recompilation. This feature simplifies the deployment of applications across Open Systems platforms.

Natural applications generated with Natural Version 4 or Natural Version 3 can be executed with Natural Version 5 or above without cataloging the applications again (upward compatibility). In this case, the portable GP functionality is not available. To make use of the portable GP and other improvements, cataloging with Natural Version 5 or above is required.

Command processor GPs are not portable. The portable GP feature is not available for mainframe platforms. This means that Natural GPs which are generated on mainframe computers are not executable on UNIX, OpenVMS and Windows platforms without recompilation of the application and vice versa.

Endian Mode Considerations

As of Natural Version 5, Natural acts as follows: Depending on which UNIX, OpenVMS or Windows platform it is running, Natural will consider the byte order in which multi-byte numbers are stored in the GP. The two byte order modes are called “Little Endian” and “Big Endian”.

- “Little Endian” means that the low-order byte of the number is stored in memory at the lowest address, and the high-order byte at the highest address (the little end comes first).
- “Big Endian” means that the high-order byte of the number is stored in memory at the lowest address, and the low-order byte at the highest address (the big end comes first).

The Linux and Windows platforms use both endian modes: Intel processors and AXP computers have “Little Endian” byte order, other processors such as HP-UX, Sun Solaris, or RS6000 use “Big Endian” mode.

Natural converts a portable GP automatically into the endian mode of the execution platform, if necessary. This endian conversion is not performed if the GP has been generated in the endian mode of the platform.

ENDIAN Parameter

In order to increase execution performance of portable GPs, the profile parameter `ENDIAN` has been introduced. `ENDIAN` determines the endian mode in which a GP is generated during compilation:

`DEFAULT`	The endian mode of the machine on which the GP is generated.
`BIG`	Big endian mode (high order byte first).
`LITTLE`	Little endian mode (low order byte first).

The values `DEFAULT`, `BIG` and `LITTLE` are alternatives whereby the default value is `DEFAULT`.

The `ENDIAN` mode parameter may be set

- as a profile parameter with the Natural Configuration Utility,
- as a start-up parameter,
- as a session parameter or with the `GLOBALS` command.

Transferring Natural Generated Programs

To make use of the portable GP on different platforms (Linux and Windows), the generated Natural objects must be transferred to the target platform or must be accessible from the target platform, for example, via NFS.

Using the Natural Object Handler is the recommended way to distribute Natural generated objects or even entire Natural applications. This is done by unloading the objects in the source environment into a work file, transferring the work file to the target environment and loading the objects from the work file.

To deploy your Natural generated objects across Open Systems platforms

1. Start the Natural Object Handler. Unload all necessary cataloged objects into a work file of type `PORTABLE`.

   Error messages, if needed, can also be unloaded to the work file.

   Important: The specified work file type must be of type `PORTABLE`. `PORTABLE` performs an automatic endian conversion of a work file when it is transferred to a different machine. See also the information on the work file type in the description of the `DEFINE WORK FILE` statement in the *Statements* documentation.

2 Transfer the work file to the target environment. Depending on the transfer mechanism (network, CD, diskette, tape, email, download, etc.), the use of a compressed archive such as a ZIP file or encoding with UUENCODE/UUDECODE or similar may make sense. Copying via FTP requires binary transfer type.

Note: According to the transfer method used, it may be necessary to adjust the record format and attributes or block size of the transferred work file depending on the specific target platform, before continuing with the load function. The work file should have the same format and attributes on the target platform as a work file of the same type that was generated on the target platform itself. Use operating system tools if an adaptation is necessary.

3 Start the Natural Object Handler in the target environment. Select `PORTABLE` as work file type. Load the Natural objects and error messages from the work file.

For more details on how to use the Natural Object Handler, refer to *Object Handler* in the *Utilities* documentation.

Beside the aforementioned preferred method, there are various other ways of “moving” or copying single Natural generated objects or even entire libraries or parts thereof, using operating system tools and different transfer methods. In all of these cases, to make the objects executable by Natural, they have to be imported into the Natural system file `FUSER` so that the *FILEDIR.SAG* structure is adapted. For information on the `FNAT` or `FUSER` directory, see *System Files FNAT and FUSER* in the *Operations* documentation.

This can be done with either of the following methods:

- Using the Import function of the `SYSMAIN` utility.
- Using the `FTOUCH` utility. This utility can be used without entering Natural.

The same applies when direct access is possible from a target platform to the generated objects in the source environment, for example, via NSF, network file server, etc. In this case, the objects have to be imported, too.

Portable FILEDIR.SAG and Error Message Files

As of Natural Version 6.2, the file *FILEDIR.SAG* and the error message files are platform independent. Hence, it is possible to share common `FUSER` system files among different Open Systems platforms. For example, it is possible to copy sets of Natural libraries from one Open Systems platform to another with operating system copy procedures. However, it is not recommended to share `FNAT` system files. For more information about the portable *FILEDIR.SAG*, refer to *Portable Natural System Files* in the *Operations* documentation.

X Application User Interfaces

The user interface of an application, that is, the way an application presents itself to the user, is a key consideration when writing an application.

This part provides information on the various possibilities Natural offers for designing character-based user interfaces that are uniform in presentation and provide powerful mechanisms for user guidance and interaction.

When designing user interfaces, standards and standardization are key factors.

Using Natural, you can offer the end user common functionality across various hardware and operating systems.

This includes the general screen layout (information, data and message areas), function-key assignment and the layout of windows.

This part covers the following topics:

58 Screen Design

Control of Function-Key Lines - Terminal Command %Y

With the terminal command `%Y` you can define how and where the Natural function-key lines are to be displayed.

Below is information on:

- Format of Function-Key Lines
- Positioning of Function-Key Lines
- Cursor-Sensitivity

Format of Function-Key Lines

The following terminal commands are available for defining the format of function-key lines:

%YN
The function-key lines are displayed in tabular Software AG format:

```
 Command ===>
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
      Help        Exit                                                     Canc
```

%YS
The function-key lines display the keys sequentially and only show those keys to which names have been assigned (PF1=*value*, PF2=*value*, etc.):

```
Command ===>                                                                  ↩

PF1=Help,PF3=Exit,PF12=Canc
```

%YP
The function-key lines are displayed in PC-like format, that is, sequentially and only showing those keys to which names have been assigned (F1=*value*, F2=*value*, etc.):

```
Command ===>                                                                  ↩

F1=Help,F3=Exit,F12=Canc
```

Other Display Options

Various other command options are available for function-key lines, such as:

- single- and double-line display,
- intensified display,
- reverse video display,
- color display.

For details on these options, see *%Y - Control of PF-Key Lines* in the *Terminal Commands* documentation.

Positioning of Function-Key Lines

%YB
The function-key lines are displayed at the bottom of the screen.

%YT
The function-key lines are displayed at the top of the screen.

%Y*nn*
The function-key lines are displayed on line *nn* of the screen.

Cursor-Sensitivity

%YC
This command makes the function-key lines cursor-sensitive. This means that they act like an action bar on a PC screen: you just move the cursor to the desired function-key number or name and press Enter, and Natural reacts as if the corresponding function key had been pressed.

To switch cursor-sensitivity off, you enter `%YC` again (toggle switch).

By using `%YC` in conjunction with tabular display format (`%YN`) and having only the function-key names displayed (`%YH`), you can equip your applications with very comfortable action bar processing: the user merely has to select a function name with the cursor and press Enter, and the function is executed.

Control of the Message Line - Terminal Command %M

Various options of the terminal command %M are available for defining how and where the Natural message line is to be displayed.

Below is information on:

- Positioning the Message Line
- Message Line Color

Positioning the Message Line

%MB
The message line is displayed at the bottom of the screen.

%MT
The message line is displayed at the top of the screen.

Other options for the positioning of the message line are described in *%M - Control of Message Line* in the *Terminal Commands* documentation.

Message Line Color

%M=*color-code*
The message line is displayed in the specified color (for an explanation of color codes, see the session parameter CD as described in the *Parameter Reference*).

Assigning Colors to Fields - Terminal Command %=

You can use the terminal command %= to assign colors to field attributes for programs that were originally not written for color support. The command causes all fields/text defined with the specified attributes to be displayed in the specified color.

If predefined color assignments are not suitable for your terminal type, you can use this command to override the original assignments with new ones.

You can also use the %= terminal command within Natural editors, for example to define color assignments dynamically during map creation.

Codes	Description
blank	Clear color translate table.
F	Newly defined colors are to override colors assigned by the program.
N	Color attributes assigned by program are not to be modified.
O	Output field.
M	Modifiable field (output and input).
T	Text constant.
B	Blinking
C	Italic
D	Default
I	Intensified
U	Underlined
V	Reverse video
BG	Background
BL	Blue
GR	Green
NE	Neutral
PI	Pink
RE	Red
TU	Turquoise
YE	Yellow

Example:

```
%=TI=RE,OB=YE
```

This example assigns the color red to all intensified text fields and yellow to all blinking output fields.

Infoline - Terminal Command %X

The terminal command %X controls the display of the Natural infoline.

For further information, see the description of the terminal command `%X` in the *Terminal Commands* documentation.

Windows

Below is information on:

What is a Window?

A *window* is that segment of a logical page, built by a program, which is displayed on the terminal screen.

A *logical page* is the output area for Natural; in other words the logical page contains the current report/map produced by the Natural program for display. This logical page may be larger than the physical screen.

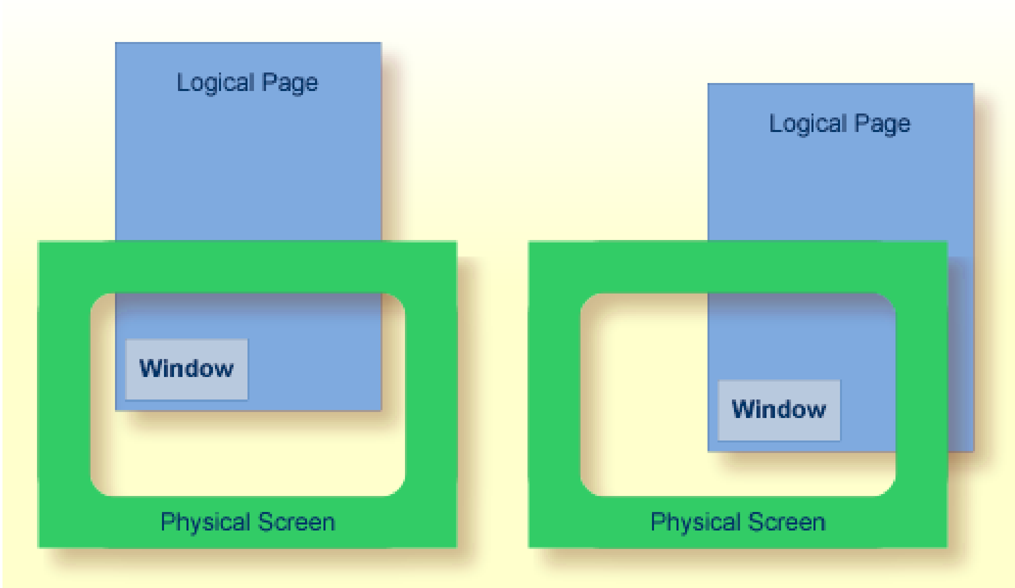
There is always a window present, although you may not be aware of its existence. Unless specified differently (by a `DEFINE WINDOW` statement), the size of the window is identical to the physical size of your terminal screen.

You can manipulate a window in two ways:

- You can control the size and position of the window on the *physical screen*.
- You can control the position of the window on the *logical page*.

Positioning on the Physical Screen

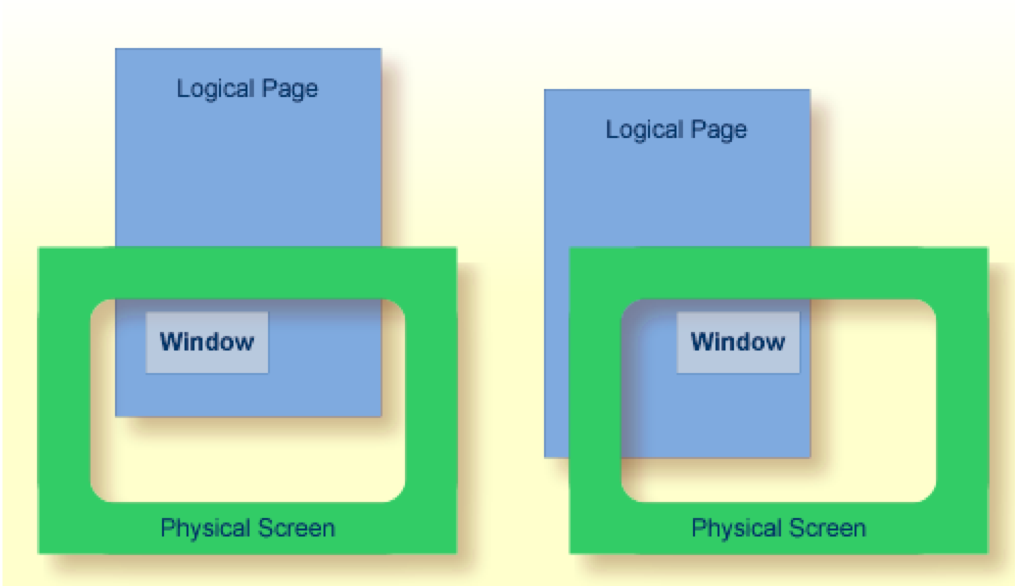
The figure below illustrates the positioning of a window on the physical screen. Note that the same section of the logical page is displayed in both cases, only the position of the window on the screen has changed.



Positioning on the Logical Page

The figure below illustrates the positioning of a window on the logical page.

When you change the position of the window on the *logical page*, the size and position of the window on the *physical screen* will remain unchanged. In other words, the window is not moved over the page, but the page is moved “underneath” the window.



DEFINE WINDOW Statement

You use the `DEFINE WINDOW` statement to specify the size, position and attributes of a window on the *physical screen*.

A `DEFINE WINDOW` statement does not activate a window; this is done with a `SET WINDOW` statement or with the `WINDOW` clause of an `INPUT` statement.

Various options are available with the `DEFINE WINDOW` statement. These are described below in the context of the following example.

The following program defines a window on the physical screen.

```
** Example 'WINDX01': DEFINE WINDOW
************************************************************************
DEFINE DATA LOCAL
1 COMMAND (A10)
END-DEFINE
*
DEFINE WINDOW TEST
       SIZE 5*25
       BASE 5/40
       TITLE 'Sample Window'
       CONTROL WINDOW
       FRAMED POSITION SYMBOL BOTTOM LEFT
*
INPUT WINDOW='TEST' WITH TEXT 'message line'
      COMMAND (AD=I'_') /
      'dataline 1' /
      'dataline 2' /
      'dataline 3' 'long data line'
*
IF COMMAND = 'TEST2'
  FETCH 'WINDX02'
ELSE
  IF COMMAND = '.'
    STOP
  ELSE
    REINPUT 'invalid command'
  END-IF
END-IF
END
```

The window-name identifies the window. The name may be up to 32 characters long. For a window name, the same naming conventions apply as for user-defined variables. Here the name is `TEST`.

The window size is set with the `SIZE` option. Here the window is 5 lines high and 25 columns (positions) wide.

The position of the window is set by the `BASE` option. Here the top left-hand corner of the window is positioned on line 5, column 40.

With the `TITLE` option, you can define a title that is to be displayed in the window frame (of course, only if you have defined a frame for the window).

With the `CONTROL` clause, you determine whether the PF-key lines, the message line and the statistics line are displayed in the window or on the full physical screen. Here `CONTROL WINDOW` causes the message line to be displayed inside the window. `CONTROL SCREEN` would cause the lines to be displayed on the full physical screen outside the window. If you omit the `CONTROL` clause, `CONTROL WINDOW` applies by default.

With the `FRAMED` option, you define that the window is to be framed. This frame is then cursor-sensitive. Where applicable, you can page forward, backward, left or right within the window by simply placing the cursor over the appropriate symbol (<, -, +, or >; see `POSITION` clause) and then pressing Enter. In other words, you are moving the *logical page* underneath the window on the physical screen. If no symbols are displayed, you can page backward and forward within the window by placing the cursor in the top frame line (for backward positioning) or bottom frame line (for forward positioning) and then pressing Enter.

With the `POSITION` clause of the `FRAMED` option, you define that information on the position of the window on the logical page is to be displayed in the frame of the window. This applies only if the logical page is larger than the window; if it is not, the `POSITION` clause will be ignored. The position information indicates in which directions the logical page extends above, below, to the left and to the right of the current window.

If the `POSITION` clause is omitted, `POSITION SYMBOL TOP RIGHT` applies by default.

`POSITION SYMBOL` causes the position information to be displayed in form of symbols: “More: < - + >”. The information is displayed in the top and/or bottom frame line.

`TOP`/`BOTTOM` determines whether the position information is displayed in the top or bottom frame line.

`LEFT`/`RIGHT` determines whether the position information is displayed in the left or right part of the frame line.

You can define which characters are to be used for the frame with the terminal command `%F=`*`chv`*.

c	The first character will be used for the four *corners* of the window frame.
h	The second character will be used for the *horizontal* frame lines.
v	The third character will be used for the *vertical* frame lines.

Example:

```
%F=+-!
```

The above command makes the window frame look like this:

```
+-----------------------+
!                       !
!                       !
!                       !
!                       !
+-----------------------+
```

INPUT WINDOW Statement

The `INPUT WINDOW` statement activates the window defined in the `DEFINE WINDOW` statement. In the example, the window TEST is activated. Note that if you wish to output data in a window (for example, with a `WRITE` statement), you use the `SET WINDOW` statement.

When the above program is run, the window is displayed with one input field `COMMAND`. The session parameter `AD` is used to define that the value of the field is displayed intensified and an underscore is used as filler character.

Output of Program WINDX01:

```
> r                                    > +  Program     WINDX01  Lib SYSEXPG
Top    ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
  0010 ** Example 'WINDX01': DEFINE WINDOW
  0020 ****************************** +----Sample Window-----+ ***************
  0030 DEFINE DATA LOCAL               ! message line         !
  0040 1 COMMAND (A10)                 ! COMMAND __________   !
  0050 END-DEFINE                      ! dataline 1           !
  0060 *                               +More:     + >---------+
  0070 DEFINE WINDOW TEST
  0080        SIZE 5*25
  0090        BASE 5/40
  0100        TITLE 'Sample Window'
  0110        CONTROL WINDOW
  0120        FRAMED POSITION SYMBOL BOTTOM LEFT
  0130 *
  0140 INPUT WINDOW='TEST' WITH TEXT 'message line'
  0150       COMMAND (AD=I'_') /
  0160       'dataline 1' /
  0170       'dataline 2' /
  0180       'dataline 3' 'long data line'
  0190 *
  0200 IF COMMAND = 'TEST2'
       ....+....1....+....2....+....3....+....4....+....5....+... S 29   L 1
```

In the bottom frame line, the position information `More: + >` indicates that there is more information on the logical page than is displayed in the window.

To see the information that is further down on the logical page, you place the cursor in the bottom frame line on the plus (+) sign and press Enter.

The window is now moved downwards. Note that the text `long data line` does not fit in the window and is consequently not fully visible.

```
> r                                    > +  Program     WINDX01  Lib SYSEXPG
Top    ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
  0010 ** Example 'WINDX01': DEFINE WINDOW
  0020 ****************************** +----Sample Window-----+ ***************
  0030 DEFINE DATA LOCAL              ! message line         !
  0040 1 COMMAND (A10)                ! dataline 3 long data !
  0050 END-DEFINE                     ! dataline 2           !
  0060 *                              +More:   -   >---------+
  0070 DEFINE WINDOW TEST
  0080        SIZE 5*25
  0090        BASE 5/40
  0100        TITLE 'Sample Window'
  0110        CONTROL WINDOW
  0120        FRAMED POSITION SYMBOL BOTTOM LEFT
  0130 *
  0140 INPUT WINDOW='TEST' WITH TEXT 'message line'
  0150       COMMAND (AD=I'_') /
  0160       'dataline 1' /
  0170       'dataline 2' /
  0180       'dataline 3' 'long data line'
  0190 *
  0200 IF COMMAND = 'TEST2'
       ....+....1....+....2....+....3....+....4....+....5....+... S 29   L 1
```

To see this hidden information to the right, you place the cursor in the bottom frame line on the less-than symbol (>) and press Enter. The window is now moved to the right on the logical page and displays the previously invisible word `line`:

```
> r                                    > +  Program     WINDX01  Lib SYSEXPG
Top    ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
  0010 ** Example 'WINDX01': DEFINE WINDOW
  0020 ****************************** +----Sample Window-----+ ***************
  0030 DEFINE DATA LOCAL              ! message line          !
  0040 1 COMMAND (A10)                !  line                 ! <==
  0050 END-DEFINE                     !                       !
  0060 *                              +More: < -    ---------+
  0070 DEFINE WINDOW TEST
  0080        SIZE 5*25
  0090        BASE 5/40
  0100        TITLE 'Sample Window'
  0110        CONTROL WINDOW
  0120        FRAMED POSITION SYMBOL BOTTOM LEFT
  0130 *
  0140 INPUT WINDOW='TEST' WITH TEXT 'message line'
  0150       COMMAND (AD=I'_') /
  0160       'dataline 1' /
  0170       'dataline 2' /
  0180       'dataline 3' 'long data line'
  0190 *
  0200 IF COMMAND = 'TEST2'
       ....+....1....+....2....+....3....+....4....+....5....+... S 29   L 1
```

Multiple Windows

You can, of course, open multiple windows. However, only one Natural window is active at any one time, that is, the most recent window. Any previous windows may still be visible on the screen, but are no longer active and are ignored by Natural. You may enter input only in the most recent window. If there is not enough space to enter input, the window size must be adjusted first.

When `TEST2` is entered in the `COMMAND` field, the program `WINDX02` is executed.

```
** Example 'WINDX02': DEFINE WINDOW
************************************************************************
DEFINE DATA LOCAL
1 COMMAND (A10)
END-DEFINE
*
DEFINE WINDOW TEST2
       SIZE 5*30
       BASE 15/40
       TITLE 'Another Window'
       CONTROL SCREEN
       FRAMED POSITION SYMBOL BOTTOM LEFT
*
INPUT WINDOW='TEST2' WITH TEXT 'message line'
      COMMAND (AD=I'_') /
      'dataline 1' /
```

```
      'dataline 2' /
      'dataline 3' 'long data line'
*
IF COMMAND = 'TEST'
  FETCH 'WINDX01'
ELSE
  IF COMMAND = '.'
    STOP
  ELSE
    REINPUT 'invalid command'
  END-IF
END-IF
END
```

A second window is opened. The other window is still visible, but it is inactive.

```
message line
> r                                     > +  Program     WINDX01  Lib SYSEXPG
Top    ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
  0010 ** Example 'WINDX01': DEFINE WINDOW
  0020 ****************************** +----Sample Window-----+ ***************
  0030 DEFINE DATA LOCAL               ! message line         !  Inactive
  0040 1 COMMAND (A10)                 ! COMMAND TEST2_____   !  Window
  0050 END-DEFINE                      ! dataline 1           !  <==
  0060 *                               +More:     + >---------+
  0070 DEFINE WINDOW TEST
  0080        SIZE 5*25
  0090        BASE 5/40
  0100        TITLE 'Sample Window'
  0110        CONTROL WINDOW
  0120       FRAMED POSITION SYMBOL B +------Another Window-------+  Currently   ↩

  0130 *                               ! COMMAND __________        !  Active
  0140 INPUT WINDOW='TEST' WITH TEXT ' ! dataline 1                !  Window
  0150       COMMAND (AD=I'_') /       ! dataline 2                !  <==
  0160       'dataline 1' /            +More:   +------------------+
  0170       'dataline 2' /
  0180       'dataline 3' 'long data line'
  0190 *
  0200 IF COMMAND = 'TEST2'                                                    ↩
```

Note that for the new window the message line is now displayed on the full physical screen (at the top) and not in the window. This was defined by the `CONTROL SCREEN` clause in the `WINDX02` program.

For further details on the statements `DEFINE WINDOW`, `INPUT WINDOW` and `SET WINDOW`, see the corresponding descriptions in the *Statements* documentation.

Standard and Dynamic Map Layouts

A standard layout can be defined in the map editor. This layout guarantees a uniform appearance for all maps that reference it throughout the application.

When a map that references a standard layout is initialized, the standard layout becomes a fixed part of the map. This means that if this standard layout is modified, all affected maps must be re-cataloged before the changes take effect.

In contrast to a standard layout, a dynamic layout does not become a fixed part of a map that references it, rather it is executed at runtime.

This means that if you define the map layout as dynamic in the map editor, any modifications to the layout map are also carried out on all maps that reference it. The maps need not be re-cataloged.

For details on defining standard and dynamic map layouts, see *Map Profile* in the *Editors* documentation.

Multilingual User Interfaces

Using Natural, you can create multilingual applications for international use.

Maps, helproutines, error messages, programs, functions, subprograms and copycodes can be defined in up to 60 different languages (including languages with double-byte character sets).

Below is information on:

- Language Codes
- Defining the Language of a Natural Object
- Defining the User Language
- Referencing Multilingual Objects
- Programs
- Error Messages

- Edit Masks for Date and Time Fields

Language Codes

In Natural, each language has a *language code* (from 1 to 60). The table below is an extract from the full table of language codes. For a complete overview, refer to the description of the system variable `*LANGUAGE` in the *System Variables* documentation.

Language Code	Language	Map Code in Object Names
1	English	1
2	German	2
3	French	3
4	Spanish	4
5	Italian	5
6	Dutch	6
7	Turkish	7
8	Danish	8
9	Norwegian	9
10	Albanian	A
11	Portuguese	B

The language code (left column) is the code that is contained in the system variable `*LANGUAGE`. This code is used by Natural internally. It is the code you use to define the user language (see ***Defining the User Language*** below). The code you use to identify the language of a Natural object is the *map code* in the right-hand column of the table.

Example:

The language code for Portuguese is "11". The code you use when cataloging a Portuguese Natural object is "B".

For the full table of language codes, see the system variable `*LANGUAGE` as described in the *System Variables* documentation.

Defining the Language of a Natural Object

To define the language of a Natural object (map, helproutine, program, function, subprogram or copycode), you add the corresponding map code to the object name. Apart from the map code, the name of the object must be identical for all languages.

In the example below, a map has been created in English and in German. To identify the languages of the maps, the map code that corresponds to the respective language has been included in the map name.

Example of Map Names for a Multilingual Application

`DEMO1` = English map (map code 1)

`DEMO2` = German map (map code 2)

Defining Languages with Alphabetical Map Codes

Map codes are in the range 1-9, A-Z or a-y. The alphabetical map codes require special handling.

Normally, it is not possible to catalog an object with a lower-case letter in the name - all characters are automatically converted into capitals.

This is however necessary, if for example you wish to define an object for Kanji (Japanese) which has the language code 59 and the map code `x`.

To catalog such an object, you first set the correct language code (here 59) using the terminal command `%L=`*`nn`*, where *`nn`* is the language code.

You then catalog the object using the ampersand (&) character instead of the actual map code in the object name. So to have a Japanese version of the map `DEMO`, you stow the map under the name `DEMO&`.

If you now look at the list of Natural objects, you will see that the map is correctly listed as `DEMOx`.

Objects with language codes 1-9 and upper case A-Z can be cataloged directly without the use of the ampersand (&) notation.

In the example list below, you can see the three maps `DEMO1`, `DEMO2` and `DEMOx`. To delete the map `DEMOx`, you use the same method as when creating it, that is, you set the correct language with the terminal command `%L=59` and then confirm the deletion with the ampersand (&) notation (`DEMO&`).

Defining the User Language

You define the language to be used per user - as defined in the system variable *LANGUAGE - with the profile parameter ULANG (which is described in the *Parameter Reference*) or with the terminal command %L=*nn* (where *nn* is the language code).

Referencing Multilingual Objects

To reference multilingual objects in a program, you use the ampersand (&) character in the name of the object.

The program below uses the maps DEMO1 and DEMO2. The ampersand (&) character at the end of the map name stands for the map code and indicates that the map with the current language as defined in the *LANGUAGE system variable is to be used.

```
DEFINE DATA LOCAL
1 PERSONNEL VIEW OF EMPLOYEES
  2 NAME (A20)
  2 PERSONNEL-ID (A8)
1 CAR VIEW OF VEHICLES
   2 REG-NUM (A15)
1 #CODE (N1)
END-DEFINE
*
INPUT USING MAP 'DEMO&' /* <--- INVOKE MAP WITH CURRENT LANGUAGE CODE
...
```

When this program is run, the English map (DEMO1) is displayed. This is because the current value of *LANGUAGE is 1 = English.

```
                                                            MAP DEMO1

                                SAMPLE MAP



                    Please select a function!



                    1.) Employee information

                    2.) Vehicle information



                    Enter code here: _
```

In the example below, the language code has been switched to `2` = German with the terminal command `%L=2`.

When the program is now run, the German map (`DEMO2`) is displayed.

```
                                BEISPIEL-MAP



                    Bitte wählen Sie eine Funktion!



                    1.) Mitarbeiterdaten

                    2.) Fahrzeugdaten



                 Code hier eingeben: _
```

Programs

For some applications it may be useful to define multilingual programs. For example, a standard invoicing program might use different subprograms to handle various tax aspects, depending on the country where the invoice is to be written.

Multilingual programs are defined with the same technique as described above for maps.

Error Messages

Using the Natural utility SYSERR, you can translate Natural error messages into up to 60 languages, and also define your own error messages.

Which message language a user sees, depends on the `*LANGUAGE` system variable.

For further information on error messages, see *SYSERR Utility* in the *Utilities* documentation.

Edit Masks for Date and Time Fields

The language used for date and time fields defined with edit masks also depends on the system variable `*LANGUAGE`.

For details on edit masks, see the session parameter `EM` as described in the *Parameter Reference*.

Skill-Sensitive User Interfaces

Users with varying levels of skill may wish to have different maps (of varying detail) while using the same application.

If your application is not for international use by users speaking different languages, you can use the techniques for multilingual maps to define maps of varying detail.

For example, you could define language code 1 as corresponding to the skill of the beginner, and language code 2 as corresponding to the skill of the advanced user. This simple but effective technique is illustrated below.

The following map (PERS1) includes instructions for the end user on how to select a function from the menu. The information is very detailed. The name of the map contains the map code 1:

```
                                                               MAP PERS1

                                     SAMPLE MAP

                    Please select a function


                    1.) Employee information  _

                    2.) Vehicle information   _

                        Enter code:  _

  To select a function, do one of the following:

  - place the cursor on the input field next to desired function and press ENTER
  - mark the input field next to desired function with an X and press ENTER
  - enter the desired function code (1 or 2) in the 'Enter code' field and press
ENTER
```

The same map, but without the detailed instructions is saved under the same name, but with map code 2.

```
                                                           MAP PERS2


                                     SAMPLE MAP

                    Please select a function


                    1.) Employee information  _

                    2.) Vehicle information   _

                        Enter code:  _
```

In the example above, the map with the detailed instructions is output, if the `ULANG` profile parameter has the value 1, the map without the instructions if the value is 2. See also the description of the profile parameter `ULANG` (in the *Parameter Reference*).

59 Dialog Design

This chapter tells you how you can design character-based user interfaces that make user interaction with the application simple and flexible.

Field-Sensitive Processing

*CURS-FIELD and POS(field-name)

Using the system variable `*CURS-FIELD` together with the system function `POS(`*`field-name`*`)`, you can define processing based on the field where the cursor is positioned at the time the user presses Enter.

`*CURS-FIELD` contains the internal identification of the field where the cursor is currently positioned; it cannot be used by itself, but only in conjunction with `POS(`*`field-name`*`)`.

You can use `*CURS-FIELD` and `POS(`*`field-name`*`)`, for example, to enable a user to select a function simply by placing the cursor on a specific field and pressing Enter.

The example below illustrates such an application:

```
DEFINE DATA LOCAL
1 #EMP (A1)
1 #CAR (A1)
1 #CODE (N1)
END-DEFINE
*
INPUT USING MAP 'CURS'
*
DECIDE FOR FIRST CONDITION
  WHEN *CURS-FIELD = POS(#EMP) OR #EMP = 'X' OR #CODE = 1
    FETCH 'LISTEMP'
  WHEN *CURS-FIELD = POS(#CAR) OR #CAR = 'X' OR #CODE = 2
    FETCH 'LISTCAR'
   WHEN NONE
    REINPUT 'PLEASE MAKE A VALID SELECTION'
END-DECIDE
END
```

And the result:

```
                                  SAMPLE MAP

                    Please select a function


                    1.) Employee information  _
                    2.) Vehicle information   _  <== Cursor positioned
                                                      on field

                        Enter code:  _

  To select a function, do one of the following:

  - place the cursor on the input field next to desired function and press Enter
  - mark the input field next to desired function with an X and press Enter
  - enter the desired function code (1 or 2) in the 'Enter code' field and press
Enter
```

If the user places the cursor on the input field (`#EMP`) next to Employee information, and presses Enter, the program LISTEMP displays a list of employee names:

```
 Page      1                                                2001-01-22  09:39:32

         NAME
 --------------------

 ABELLAN
 ACHIESON
 ADAM
 ADKINSON
 ADKINSON
 ADKINSON
 ADKINSON
 ADKINSON
 ADKINSON
 ADKINSON
 ADKINSON
 AECKERLE
 AFANASSIEV
 AFANASSIEV
 AHL
 AKROYD
```

Notes:

1. In Natural for Ajax applications, `*CURS-FIELD` identifies the operand that represents the value of the control that has the input focus. You may use `*CURS-FIELD` in conjunction with the `POS` function to check for the control that has the input focus and perform processing depending on that condition.

2. The values of `*CURS-FIELD` and `POS(`*`field-name`*`)` serve for internal identification of the fields only. They cannot be used for arithmetical operations.

Simplifying Programming

System Function POS

The Natural system function `POS(`*`field-name`*`)` contains the internal identification of the field whose name is specified with the system function.

`POS(`*`field-name`*`)` may be used to identify a specific field, regardless of its position in a map. This means that the sequence and number of fields in a map may be changed, but `POS(`*`field-name`*`)` will still uniquely identify the same field. With this, for example, you need only a single `REINPUT` statement to make the field to be `MARK`ed dependent on the program logic.

Note: The value `POS(`*`field-name`*`)` serves for internal identification of the fields only. It cannot be used for arithmetical operations.

Example:

```
...
DECIDE ON FIRST VALUE OF ...
   VALUE ...
      COMPUTE #FIELDX = POS(FIELD1)
   VALUE ...
      COMPUTE #FIELDX = POS(FIELD2)
   ...
END-DECIDE
...
REINPUT ... MARK #FIELDX
...
```

Full details on `*CURS-FIELD` and `POS(`*`field-name`*`)` are described in the *System Variables* and *System Functions* documentation.

Line-Sensitive Processing

System Variable *CURS-LINE

Using the system variable `*CURS-LINE`, you can make processing dependent on the line where the cursor is positioned at the time the user presses Enter.

Using this variable, you can make user-friendly menus. With the appropriate programming, the user merely has to place the cursor on the line of the desired menu option and press Enter to execute the option.

The cursor position is defined within the current active window, regardless of its physical placement on the screen.

Note: The message line, function-key lines and statistics line/infoline are not counted as data lines on the screen.

The example below demonstrates line-sensitive processing using the `*CURS-LINE` system variable. When the user presses Enter on the map, the program checks if the cursor is positioned on line 8 of the screen which contains the option `Employee information`. If this is the case, the program that lists the names of employees LISTEMP is executed.

```
DEFINE DATA LOCAL
1 #EMP (A1)
1 #CAR (A1)
1 #CODE (N1)
END-DEFINE
*
INPUT USING MAP 'CURS'
*
DECIDE FOR FIRST CONDITION
  WHEN *CURS-LINE = 8
     FETCH 'LISTEMP'
   WHEN NONE
     REINPUT 'PLACE CURSOR ON LINE OF OPTION YOU WISH TO SELECT'
END-DECIDE
END
```

Output:

```
                              Company Information


                  Please select a function


               [] 1.) Employee information

                  2.) Vehicle information









   Place the cursor on the line of the option you wish to select and press
   Enter
```

The user places the cursor indicated by square brackets [] on the line of the desired option and presses Enter and the corresponding program is executed.

Column-Sensitive Processing

System Variable *CURS-COL

The system variable `*CURS-COL` can be used in a similar way to `*CURS-LINE` described above. With `*CURS-COL` you can make processing dependent on the column where the cursor is positioned on the screen.

Processing Based on Function Keys

System Variable *PF-KEY

Frequently you may wish to make processing dependent on the function key a user presses.

This is achieved with the statement `SET KEY`, the system variable `*PF-KEY` and a modification of the default map settings (`Standard Keys = Y`).

The `SET KEY` statement assigns functions to function keys during program execution. The system variable `*PF-KEY` contains the identification of the last function key the user pressed.

The example below illustrates the use of `SET KEY` in combination with `*PF-KEY`.

```
...
SET KEY PF1
*
INPUT USING MAP 'DEMO&'
IF *PF-KEY = 'PF1'
  WRITE 'Help is currently not active'
END-IF
...
```

The `SET KEY` statement activates PF1 as a function key.

The `IF` statement defines what action is to be taken when the user presses PF1. The system variable `*PF-KEY` is checked for its current content; if it contains PF1, the corresponding action is taken.

Further details regarding the statement `SET KEY` and the system variable `*PF-KEY` are described in the *Statements* and the *System Variables* documentation respectively.

Processing Based on Function-Key Names

System Variable *PF-NAME

When defining processing based on function keys, further comfort can be added by using the system variable `*PF-NAME`. With this variable you can make processing dependent on the name of a function, not on a specific key.

The variable `*PF-NAME` contains the name of the last function key the user pressed (that is, the name as assigned to the key with the `NAMED` clause of the `SET KEY` statement).

For example, if you wish to allow users to invoke help by pressing either PF3 or PF12, you assign the same name (in the example below: `INFO`) to both keys. When the user presses either one of the keys, the processing defined in the `IF` statement is performed.

```
...
SET KEY PF3  NAMED 'INFO'
        PF12 NAMED 'INFO'
INPUT USING MAP 'DEMO&'
IF *PF-NAME = 'INFO'
  WRITE 'Help is currently not active'
END-IF
...
```

The function names defined with `NAMED` appear in the function-key lines:

Processing Data Outside an Active Window

Below is information on:

- System Variable *COM
- Example Usage of *COM
- Positioning the Cursor to *COM - the %T* Terminal Command

System Variable *COM

As stated in the section *Screen Design* - ***Windows***, only *one* window is active at any one time. This normally means that input is only possible within that particular window.

Using the `*COM` system variable, which can be regarded as a communication area, it is possible to enter data outside the current window.

The prerequisite is that a map contains `*COM` as a modifiable field. This field is then available for the user to enter data when a window is currently on the screen. Further processing can then be made dependent on the content of `*COM`.

This allows you to implement user interfaces as already used, for example, by Con-nect, Software AG's office system, where a user can always enter data in the command line, even when a window with its own input fields is active.

Note that `*COM` is only cleared when the Natural session is ended.

Example Usage of *COM

In the example below, the program ADD performs a simple addition using the input data from a map. In this map, `*COM` has been defined as a modifiable field (at the bottom of the map) with the length specified in the `AL` field of the Extended Field Editing. The result of the calculation is displayed in a window. Although this window offers no possibility for input, the user can still use the `*COM` field in the map outside the window.

Program ADD:

```
DEFINE DATA LOCAL
1 #VALUE1 (N4)
1 #VALUE2 (N4)
1 #SUM3 (N8)
END-DEFINE
*
DEFINE WINDOW EMP
  SIZE 8*17
  BASE 10/2
```

```
  TITLE 'Total of Add'
  CONTROL SCREEN
  FRAMED POSITION SYMBOL BOT LEFT
*
INPUT USING MAP 'WINDOW'
*
COMPUTE #SUM3 = #VALUE1 + #VALUE2
*
SET WINDOW 'EMP'
INPUT (AD=O) / 'Value 1 +' /
               'Value 2 =' //
               ' ' #SUM3
*
IF *COM = 'M'
  FETCH 'MULTIPLY' #VALUE1 #VALUE2
END-IF
END
```

Output of Program ADD:

```
                   Map to Demonstrate Windows with *COM

                                    CALCULATOR

                        Enter values you wish to calculate


                        Value 1:  12__
                        Value 2:  12__
   +-Total of Add-+
   !              !
   ! Value 1 +    !
   ! Value 2 =    !
   !              !
   !         24   !
   !              !
   +--------------+


    Next line is input field (*COM) for input outside the window:
```

In this example, by entering the value `M`, the user initiates a multiplication function; the two values from the input map are multiplied and the result is displayed in a second window:

```
                  Map to Demonstrate Windows with *COM

                                  CALCULATOR

                       Enter values you wish to calculate


                       Value 1:  12__
                       Value 2:  12__
   +-Total of Add-+                             +--------------+
   !              !                             !              !
   ! Value 1 +    !                             ! Value 1 x    !
   ! Value 2 =    !                             ! Value 2 =    !
   !              !                             !              !
   !         24   !                             !      144     !
   !              !                             !              !
   +--------------+                             +--------------+


    Next line is input field (*COM) for input outside the window:
               M
```

Positioning the Cursor to *COM - the %T* Terminal Command

Normally, when a window is active and the window contains no input fields (AD=M or AD=A), the cursor is placed in the top left corner of the window.

With the terminal command %T*, you can position the cursor to a *COM system variable outside the window when the active window contains no input fields.

By using %T* again, you can switch back to standard cursor placement.

Example:

```
...
INPUT USING MAP 'WINDOW'
*
COMPUTE #SUM3 = #VALUE1 + #VALUE2
*
SET CONTROL 'T*'
SET WINDOW 'EMP'
INPUT (AD=O) / 'Value 1 +' /
               'Value 2 =' //
               ' ' #SUM3
...
```

Copying Data from a Screen

Below is information on:

Terminal Commands %CS and %CC

With these terminal commands, you can copy parts of a screen into the Natural stack (`%CS`) or into the system variable `*COM` (`%CC`). The protected data from a specific screen line are copied field by field.

The full options of these terminal commands are described in the *Terminal Commands* documentation.

Once copied to the stack or `*COM`, the data are available for further processing. Using these commands, you can make user-friendly interfaces as in the example below.

Selecting a Line from Report Output for Further Processing

In the following example, the program COM1 lists all employee names from Abellan to Alestia.

Program COM1:

```
DEFINE DATA LOCAL
1 EMP VIEW OF EMPLOYEES
  2 NAME(A20)
  2 MIDDLE-NAME (A20)
  2 PERSONNEL-ID (A8)
END-DEFINE
*
READ EMP BY NAME STARTING FROM 'ABELLAN' THRU 'ALESTIA'
  DISPLAY NAME
END-READ
FETCH 'COM2'
END
```

Output of Program COM1:

```
 Page      1                                                2006-08-12  09:41:21

         NAME
 --------------------

 ABELLAN
 ACHIESON
 ADAM
 ADKINSON
 ADKINSON
 ADKINSON
 ADKINSON
 ADKINSON
 ADKINSON
 ADKINSON
 ADKINSON
 AECKERLE
 AFANASSIEV
 AFANASSIEV
 AHL
 AKROYD
 ALEMAN
 ALESTIA
 MORE
```

Control is now passed to the program COM2.

Program COM2:

```
DEFINE DATA LOCAL
1 EMP VIEW OF EMPLOYEES
  2 NAME(A20)
  2 MIDDLE-NAME (A20)
  2 PERSONNEL-ID (A8)
1 SELECTNAME (A20)
END-DEFINE
*
SET KEY PF5 = '%CCC'
*
INPUT NO ERASE 'SELECT FIELD WITH CURSOR AND PRESS PF5'
*
MOVE *COM TO SELECTNAME
FIND EMP WITH NAME = SELECTNAME
  DISPLAY NAME PERSONNEL-ID
END-FIND
END
```

In this program, the terminal command `%CCC` is assigned to PF5. The terminal command copies all protected data from the line where the cursor is positioned to the system variable `*COM`. This information is then available for further processing. This further processing is defined in the program lines shown in boldface.

The user can now position the cursor on the name that interests him; when he/she now presses PF5, further employee information is supplied.

```
SELECT FIELD WITH CURSOR AND PRESS PF5                     2006-08-12  09:44:25

          NAME
  --------------------

  ABELLAN
  ACHIESON
  ADAM  <==  Cursor positioned on name for which more information is required
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  AECKERLE
  AFANASSIEV
  AFANASSIEV
  AHL
  AKROYD
  ALEMAN
  ALESTIA
```

In this case, the personnel ID of the selected employee is displayed:

```
  Page      1                                              2006-08-12  09:44:52

          NAME         PERSONNEL
                          ID
  -------------------- ---------

  ADAM                 50005800
```

Statements REINPUT/REINPUT FULL

If you wish to return to and re-execute an `INPUT` statement, you use the `REINPUT` statement. It is generally used to display a message indicating that the data input as a result of the previous `INPUT` statement were invalid.

If you specify the `FULL` option in a `REINPUT` statement, the corresponding `INPUT` statement will be re-executed fully:

- With an ordinary `REINPUT` statement (without `FULL` option), the contents of variables that were changed between the `INPUT` and `REINPUT` statement will not be displayed; that is, all variables on the screen will show the contents they had when the `INPUT` statement was originally executed.
- With a `REINPUT FULL` statement, all changes that have been made after the initial execution of the `INPUT` statement will be applied to the `INPUT` statement when it is re-executed; that is, all variables on the screen contain the values they had when the `REINPUT` statement was executed.
- If you wish to position the cursor to a specified field, you can use the `MARK` option, and to position to a particular position within a specified field, you use the `MARK POSITION` option.

The example below illustrates the use of `REINPUT FULL` with `MARK POSITION`.

```
DEFINE DATA LOCAL
1 #A (A10)
1 #B (N4)
1 #C (N4)
END-DEFINE
*
INPUT (AD=M) #A #B #C
IF #A = ' '
  COMPUTE #B = #B + #C
  RESET #C
  REINPUT FULL 'Enter a value' MARK POSITION 5 IN *#A
END-IF
END
```

The user enters `3` in field `#B` and `3` in field `#C` and presses Enter.

```
  #A              #B      3 #C      3
```

The program requires field `#A` to be non-blank. The `REINPUT FULL` statement with `MARK POSITION 5 IN *#A` returns the input screen; the now modified variable `#B` contains the value 6 (after the `COMPUTE` calculation has been performed). The cursor is positioned to the 5th position in field `#A` ready for new input.

```
Enter name of field
  #A    _       #B      6 #C      0



Enter a value
```

This is the screen that would be returned by the same statement, without the `FULL` option. Note that the variables `#B` and `#C` have been reset to their status at the time of execution of the `INPUT` statement (each field contains the value 3).

```
  #A    _       #B      3 #C      3
```

Object-Oriented Processing - The Natural Command Processor

The Natural Command Processor is used to define and control navigation within an application. It consists of two parts: The development part and the run-time part.

- The development part is the utility SYSNCP. With this utility, you define commands and the actions to be performed in response to the execution of these commands. From your definitions, SYSNCP generates decision tables which determine what happens when a user enters a command.
- The run-time part is the statement `PROCESS COMMAND`. This statement is used to invoke the Command Processor within a Natural program. In the statement you specify the name of the `SYSNCP` table to be used to handle the data input by a user at that point.

For further information regarding the Natural Command Processor, see *SYSNCP Utility* in the *Utilities* documentation and the statement `PROCESS COMMAND` as described in the *Statements* documentation.

XI Natural Native Interface

This part covers the following topics:

60 Introduction

The Natural Native Interface enables an application to execute Natural code in its own process context through function calls according to the C calling convention. The interface consists of a shared library that contains a set of interface functions. These functions include initialization and uninitialization of a Natural session, logging on to a specific Natural library and execution of individual Natural modules. The calling application loads the interface library dynamically with operating system calls and then locates and calls the interface functions.

An example C program *nnisample.c* that shows the usage of the interface is contained in `<install-dir>`*/natural\samples/sysexnni*.

The Natural modules called by the C program *nnisample.c* are contained in the Natural library `SYSEXNNI`.

As outlined above, using the Natural Native Interface is a completely different approach to run a Natural session.

Using the Natural Native Interface, the C program drives the Natural session via the interface calls. Alternatively or additionally, the user may enter a program (sequence) via the `STACK` parameter.

So it is not necessary, to fill `CMOBJIN` and `CMSYNIN` and in fact does not have any effect.

Running in batch mode when the Natural Native Interface is not used, all commands and input data for a batch session are provided in `CMOBJIN` and `CMSYNIN`. Natural starts (in batch mode) and is then driven by these commands. This is not applicable/possible using the Natural Native Interface.

61 Interface Library and Location

The interface consists of a shared library that exports a set of functions. The individual functions are described in ***Interface Functions***. The shared library is called *libnatural.so* and is contained in the Natural *bin* directory. In a Natural Security environment the library name is *libnatsec.so*.

When executing a program that uses the Natural Native Interface, the Natural *bin* directory must be defined in the environment variable `LD_LIBRARY_PATH` (or `LIBPATH` respectively), so that the calling program can locate the interface library and all dependent libraries.

Note: Depending on the Linux platform, the file extension may be *.sl* instead of *.so*.

62 Interface Versioning

The Natural Native Interface might change in future versions of Natural. Natural versions that provide a modified interface will support previous interface versions in parallel, until a point in time that is determined by Software AG and is announced in time. To access an instance of a specific version of the interface, the application calls the function `nni_get_interface`. The application passes the required interface version number to the function and receives a structure with function pointers in return. The application may also request the most recent interface version, without specifying the interface version explicitly.

63 Interface Access

In order to access the interface, an application loads the interface library using a platform dependent system call.

Then the application locates the address of the function `nni_get_interface`, again using a platform dependent system call. Once the application has located the central function `nni_get_interface`, it requests an instance of the interface by calling the function `nni_get_interface` and specifying the desired interface version. The resulting structure contains the interface function pointers.

After having finished using the interface functions, the application unloads the interface library using a platform dependent system call.

The sample program *nnisample.c* demonstrates the interface. Also the platform dependent mechanism of loading the interface library and the access to the function `nni_get_interface` is illustrated by this sample program.

64 Interface Instances and Natural Sessions

The function `nni_get_interface` returns a pointer to an instance of the Natural Native Interface. One interface instance can host one Natural session at a time. An application initializes a Natural session by calling the function `nni_initialize` on a given interface instance. It uninitializes the Natural session by calling `nni_uninitialize` on that interface instance. After that it can initialize a new Natural session on the same interface instance.

It is implementation dependent if multiple interface instances and thus multiple Natural sessions can be maintained per process or per thread. In the current implementation of Natural for Windows or Linux, one process can host one Natural session at a time. Consequently, every call to `nni_get_interface` in one process yields the same interface instance. However, this unique interface instance can be used alternating by several concurrently running threads. The thread synchronization is implicitly performed by the interface functions themselves. Optionally it can be performed by the application explicitly. The interface provides the required synchronization functions `nni_enter`, `nni_try_enter` and `nni_leave`.

65 Interface Functions

nni_get_interface

Syntax

```
int nni_get_interface( int iVersion, void** ppnni_func );
```

The function returns an instance of the Natural Native Interface.

An application calls this function after having retrieved and loaded the interface library with platform depending system calls. The function returns a pointer to a structure that contains function pointers to the individual interface functions. The functions returned in the structure may differ between interface versions.

Instead of a specific interface version, the caller can also specify the constant `NNI_VERSION_CURR`, which always refers to the most recent interface version. The interface version number belonging to a given Natural version is defined in the header file *natni.h* that is delivered with that version. In Natural Version *n.n*, the interface version number is defined as `NNI_VERSION_`*nn*. `NNI_VERSION_CURR` is also defined as `NNI_VERSION_`*nn*. If the Natural version against which the function is called does not support the requested interface version, the error code `NNI_RC VERSION_ERROR` is returned. Otherwise the return code is `NNI_RC_OK`.

The pointer returned by the function represents one instance of the interface. In order to use this interface instance, the application holds on to that pointer and passes it to subsequent interface calls.

Usually the application will subsequently initialize a Natural session by calling `nni_initialize` on the given instance. After the application has finished using that Natural session, it calls `nni_uninitialize` on that instance. After that it can initialize a different Natural session on the same interface instance. After the application has finished using the interface instance entirely, it calls `nni_free_interface` on that instance.

Parameters

Parameter	Meaning
`iVersion`	Interface version number. (`NNI_VERSION_`*nn* or `NNI_VERSION_CURR`).
`ppnni_func`	Points to an NNI interface instance on return.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_PARM_ERROR	
NNI_RC_VERSION_ERROR	

nni_free_interface

Syntax

```
int nni_free_interface(void* pnni_func);
```

An application calls this function after it has finished using the interface instance and has uninitialized the Natural session it hosts. The function frees the resources occupied by that interface instance.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	

nni_initialize

Syntax

```
int nni_initialize(void* pnni_func, const char* szCmdLine, void*, void*);
```

The function initializes a Natural session with a given command line. The syntax and semantics of the command line is the same as when Natural is started interactively. If a Natural session has already been initialized on the given interface instance, that session is implicitly uninitialized before the new session is initialized.

The command line must be specified in the way that the Natural initialization can be completed without user interaction. This means especially that if a program is passed on the stack or a startup program is specified, that program must not perform an `INPUT` statement that is not satisfied from the stack. Otherwise the subsequent behavior of the Natural session is undetermined.

The Natural session is initialized as batch session and in server mode. This means that the usage of certain statements and commands in the executed Natural modules is restricted.

When initializing a Natural session under Natural Security, the command line must contain a `LOGON` command to a freely chosen default library under which the session will be started, and an appropriate user ID and password.

Example:

```
int iRes =
pnni_func->nni_initialize( pnni_func, "STACK=(LOGON,MYLIB,MYUSER,MYPASS)", 0, 0);
```

If the application later calls `nni_logon` to a different library with a different user ID and afterwards calls `nni_logoff`, the Natural session will be reset to the library and user ID that was passed during `nni_initialize`.

Parameters

Parameter	Meaning
`pnni_func`	Pointer to an NNI interface instance.
`szCmdLine`	Natural command line. May be a null pointer.
`void*`	For future use. Must be a null pointer.
`void*`	For future use. Must be a null pointer.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
`NNI_RC_OK`	
`NNI_RC_PARM_ERROR`	
rc, where *rc* < `NNI_RC_SERR_OFFSET`	Natural startup error. The real Natural startup error number as documented in *Natural Startup Errors* (which is part of the *Operations* documentation) can be determined by the following calculation: *startup-error-nr* = - (*rc* - `NNI_RC_SERR_OFFSET`) Warnings that occur during session initialization are ignored.
> 0	Natural error number.

nni_is_initialized

Syntax

```
int nni_is_initialized( void* pnni_func, int* piIsInit );
```

The function checks if the interface instance contains an initialized Natural session.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
piIsInit	Returns 0, if no Natural session is initialized, a non-zero value otherwise.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_PARM_ERROR	

nni_uninitialize

Syntax

```
int nni_uninitialize(void* pnni_func);
```

The function uninitializes the Natural session hosted by the given interface instance.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	

nni_enter

Syntax

```
int nni_enter(void* pnni_func);
```

The function lets the current thread wait for exclusive access to the interface instance and the Natural session it hosts. A thread calls this function if it wants to issue a series of interface calls that may not be interrupted by other threads. The thread releases the exclusive access to the interface instance by calling `nni_leave`.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	

nni_try_enter

Syntax

```
int nni_try_enter(void* pnni_func);
```

The function behaves like `nni_enter` except that it does not block the thread and instead always returns immediately. If a different thread already has exclusive access to the interface instance, the function returns `NNI_RC_LOCKED`.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_LOCKED	

nni_leave

Syntax

```
int nni_leave(void* pnni_func);
```

The function releases exclusive access to the interface instance and allows other threads to access that instance and the Natural session it hosts.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	

nni_logon

Syntax

```
int nni_logon(void* pnni_func, const char* szLibrary, const char* szUser, const ↩
char* szPassword);
```

The function performs a `LOGON` to the specified Natural library.

Parameters

Parameter	Meaning
`pnni_func`	Pointer to an NNI interface instance.
`szLibrary`	Name of the Natural library.
`szUser`	Name of the Natural user. May be a null pointer, if the Natural session is not running under Natural Security or if `AUTO=ON` was used during initialization.
`szPassword`	Password of that user. May be a null pointer, if the Natural session is not running under Natural Security or if `AUTO=ON` was used during initialization..

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
`NNI_RC_OK`	
`NNI_RC_NOT_INIT`	
`NNI_RC_PARM_ERROR`	
> 0	Natural error number.

nni_logoff

Syntax

```
int nni_logoff(void* pnni_func);
```

The function performs a `LOGOFF` from the current Natural library. This corresponds to a `LOGON` to the previously active library and user ID.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
> 0	Natural error number.

nni_callnat

Syntax

```
int nni_callnat(void* pnni_func, const char* szName, int iParm, struct ↩
parameter_description* rgDesc, struct natural_exception* pExcep);
```

The function calls a Natural subprogram.

The function receives its parameters as an array of `parameter_description` structures. The caller creates these structures using NNI functions in the following way:

- Use one the functions `create_parm` or `create_module_parm` to create an appropriate parameter set for the subprogram.
- If you have used `create_parm`, use the functions `init_parm_*` to initialize each parameter to the appropriate Natural data format. If you have used `create_module_parm`, the parameters are already initialized to the appropriate Natural data format.
- Assign a value to each parameter, using one the functions `nni_put_parm` or `nni_put_parm_array`.
- Call `nni_get_parm` on each parameter in the set. This fills the `parameter_description` structures.
- Pass the array of `parameter_description` structures to the function `nni_callnat`.
- After the call has been executed, extract the modified parameter values from the parameter set using the function `nni_get_parm` or `nni_get_parm_array`.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
szName	Name of the Natural subprogram.
iParm	Number of parameters. Indicates the number of occurrences of the array rgDesc.
rgDesc	An array of parm_description structures containing the parameters for the subprogram. If the subprogram does not expect parameters, the caller passes a null pointer.
pExcep	Pointer to a natural_exception structure. If a Natural error occurs during execution of the subprogram, this structure is filled with Natural error information. The caller may specify a null pointer. In this case no extended error information is returned.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_NO_MEMORY	
> 0	Natural error number.

nni_create_object

Syntax

```
int nni_create_object(void* pnni_func, const char* szName, int iParm, struct ↩
parameter_description* rgDesc, struct natural_exception* pExcep); ↩
```

Creates a Natural object (an instance of a Natural class).

The function receives its parameters as a one-element array of parameter_description structures. The caller creates the structures using NNI functions in the following way:

- Use the function nni_create_parm to create parameter set with one element.
- Use the function nni_init_parm_s to initialize the parameter with the type HANDLE OF OBJECT.
- Call nni_get_parm_info on this parameter. This fills the parameter_description structure.
- Pass the parameter_description structure to the function nni_create_object.
- After the call has been executed, extract the modified parameter value from the parameter set using one the function nni_get_parm.

The parameters passed in `rgDesc` have the following meaning:

- The first (and only) parameter must be initialized with the data type `HANDLE OF OBJECT` and contains on return the Natural object handle of the newly created object.

Parameters

Parameter	Meaning
`pnni_func`	Pointer to an NNI interface instance.
`szName`	Name of the class.
`iParm`	Number of parameters. Indicates the number of occurrences of the array `rgDesc`.
`rgDesc`	An array of `parm_description` structures containing the parameters for the object creation. The caller always passes one parameter, which will contain the object handle on return.
`pExcep`	Pointer to a `natural_exception` structure. If a Natural error occurs during object creation, this structure is filled with Natural error information. The caller may specify a null pointer. In this case no extended exception information is returned.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
`NNI_RC_OK`	
`NNI_RC_NOT_INIT`	
`NNI_RC_PARM_ERROR`	
`NNI_RC_NO_MEMORY`	
> 0	Natural error number.

nni_send_method

Syntax

```
int nni_send_method(void* pnni_func, const char* szName, int iParm, struct ↩
parameter_description* rgDesc, struct natural_exception* pExcep);
```

Sends a method call to a Natural object (an instance of a Natural class).

The function receives its parameters as an array of `parameter_description` structures. The caller creates these structures using NNI functions in the following way:

- Use the function `nni_create_parm` or `nni_create_method_parm` to create a matching parameter set.

- If you have used create_parm, use the functions init_parm_* to initialize each parameter to the appropriate Natural data format. If you have used nni_create_method_parm, the parameters are already initialized to the appropriate Natural data format.
- Assign a value to each parameter using one the functions nni_put_parm or nni_put_parm_array.
- Call nni_get_parm_info on each parameter in the set. This fills the parameter_description structures.
- Pass the array of parameter_description structures to the function nni_send_method.
- After the call has been executed, extract the modified parameter values from the parameter set using one of the nni_get_parm functions.

The parameters passed in rgDesc have the following meaning:

- The first parameter contains the object handle.
- The second parameter must be initialized to the data type of the method return value. If the method does not have a return value, the second parameter remains not initialized. On return from the method call, this parameter contains the return value of the method.
- The remaining parameters are the method parameters.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
szName	Name of the method.
iParm	Number of parameters. Indicates the number of occurrences of the array rgDesc. This is always 2 + the number of method parameters.
rgDesc	An array of parm_description structures containing the parameters for the method. If the method does not expect parameters, the caller still passes two parameters, the first for the object handle and the second for the return value.
pExcep	Pointer to a natural_exception structure. If a Natural error occurs during execution of the method, this structure is filled with Natural error information. The caller may specify a null pointer. In this case no extended exception information is returned.

Return Codes

The meaning of the return codes is explained in the section *Return Codes*.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_NO_MEMORY	
> 0	Natural error number.

nni_get_property

Syntax

```
int nni_get_property(void* pnni_func, const char* szName, int iParm, struct ↩
parameter_description* rgDesc, struct natural_exception* pExcep); ↩
```

Retrieves a property value of a Natural object (an instance of a Natural class).

The function receives its parameters as an array of `parameter_description` structures. The caller creates these structures using NNI functions in the following way:

- Use the function `nni_create_parm` or `nni_create_method_parm` to create a matching parameter set.
- If you have used `create_parm`, use the functions `init_parm_*` to initialize each parameter to the appropriate Natural data format. If you have used `create_method_parm`, the parameters are already initialized to the appropriate Natural data format.
- Assign a value to each parameter using one the functions `nni_put_parm` or `nni_put_parm_array`.
- Call `nni_get_parm_info` on each parameter in the set. This fills the `parameter_description` structures.
- Pass the array of `parameter_description` structures to the function `nni_send_method`.
- After the call has been executed, extract the modified parameter values from the parameter set using one of the `nni_get_parm` functions.

The parameters passed in `rgDesc` have the following meaning:

- The first parameter contains the object handle.
- The second parameter is initialized to the data type of the property. On return from the property access, this parameter contains the property value.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
szName	Name of the property.
iParm	Number of parameters. Indicates the number of occurrences of the array rgDesc. This is always 2.
rgDesc	An array of parm_description structures containing the parameters for the property access. The caller always passes two parameters, the first for the object handle and the second for the returned property value.
pExcep	Pointer to a natural_exception structure. If a Natural error occurs during property access, this structure is filled with Natural error information. The caller may specify a null pointer. In this case no extended exception information is returned.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_NO_MEMORY	
> 0	Natural error number.

nni_set_property

Syntax

```
int nni_set_property(void* pnni_func, const char* szName, int iParm, struct ↩
parameter_description* rgDesc, struct natural_exception* pExcep);
```

Assigns a property value to a Natural object (an instance of a Natural class).

The function receives its parameters as an array of parameter_description structures. The caller creates these structures using NNI functions in the following way:

- Use the function nni_create_parm or nni_create_prop_parm to create a matching parameter set.
- If you have used create_parm, use the functions init_parm_* to initialize each parameter to the appropriate Natural data format. If you have used create_prop_parm, the parameters are

already initialized to the appropriate Natural data format. Assign a value to each parameter using one of the `nni_put_parm` functions.

- Assign a value to each parameter using one the functions `nni_put_parm` or `nni_put_parm_array`.
- Call `nni_get_parm_info` on each parameter in the set. This fills the `parameter_description` structures.
- Pass the array of `parameter_description` structures to the function `nni_set_property`.

The parameters passed in `rgDesc` have the following meaning:

- The first parameter contains the object handle.
- The second parameter contains the property value.

Parameters

Parameter	Meaning
`pnni_func`	Pointer to an NNI interface instance.
`szName`	Name of the property.
`iParm`	Number of parameters. Indicates the number of occurrences of the array `rgDesc`. This is always 2.
`rgDesc`	An array of `parm_description` structures containing the parameters for the property access. The caller always passes two parameters, the first for the object handle and the second for the property value.
`pExcep`	Pointer to a `natural_exception` structure. If a Natural error occurs during property access, this structure is filled with Natural error information. The caller may specify a null pointer. In this case no extended exception information is returned.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
`NNI_RC_OK`	
`NNI_RC_NOT_INIT`	
`NNI_RC_PARM_ERROR`	
`NNI_RC_NO_MEMORY`	
> 0	Natural error number.

nni_delete_object

Syntax

```
int nni_delete_object(void* pnni_func, int iParm, struct parameter_description* ↩
rgDesc, struct natural_exception* pExcep);
```

Deletes a Natural object (an instance of a Natural class) created with `nni_create_object`.

The function receives its parameters as a one-element array of `parameter_description` structures. The caller creates the structures using NNI functions in the following way:

- Use the function `nni_create_parm` to create parameter set with one element.
- Use the function `nni_init_parm_s` to initialize the parameter with the type `HANDLE OF OBJECT`.
- Assign a value to the parameter using one the functions `nni_put_parm`.
- Call `nni_get_parm_info` on this parameter. This fills the `parameter_description` structure.
- Pass the `parameter_description` structure to the function `nni_delete_object`.

The parameters passed in `rgDesc` have the following meaning:

- The first (and only) parameter must be initialized with the data type `HANDLE OF OBJECT` and contains the Natural object handle of the object to be deleted.

Parameters

Parameter	Meaning
`pnni_func`	Pointer to an NNI interface instance.
`szName`	Name of the class.
`iParm`	Number of parameters. Indicates the number of occurrences of the array `rgDesc`. This is always 1.
`rgDesc`	An array of `parm_description` structures containing the parameters for the object creation. The caller always passes one parameter, which contains the object handle.
`pExcep`	Pointer to a `natural_exception` structure. If a Natural error occurs during object creation, this structure is filled with Natural error information. The caller may specify a null pointer. In this case no extended exception information is returned.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_NO_MEMORY	
> 0	Natural error number.

nni_create_parm

Syntax

```
int nni_create_parm(void* pnni_func, int iParm, void** pparmhandle);
```

Creates a set of parameters that can be passed to a Natural module.

The parameters contained in the set are not yet initialized to specific Natural data types. Before using the parameter set in a call to `nni_callnat`, `nni_create_object`, `nni_send_method`, `nni_set_property` or `nni_get_property`:

- Initialize each parameter to the required Natural data type using one of the functions `nni_init_parm_s`, `nni_init_parm_sa`, `nni_init_parm_d` or `nni_init_parm_da`.
- Assign a value to each parameter using one of the functions `nni_put_parm` or `nni_put_parm_array`.
- Turn each parameter into a `parm_description` structure using the function `nni_get_parm_info`.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iParm	Requested number of parameters. The maximum number of parameters is 32767.
pparmhandle	Points a to a pointer to a parameter set on return.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
> 0	Natural error number.

nni_create_module_parm

Syntax

```
int nni_create_module_parm(void* pnni_func, char chType, const char* szName, void** ↩
pparmhandle); ↩
```

Creates a set of parameters that can be used in a call to nni_callnat. The function enables an application to dynamically explore the signature of a callable Natural module.

The parameters contained in the returned set are already initialized to Natural data types according to the parameter data area of the specified module. Before using the parameter set in a call to nni_callnat:

- Assign a value to each parameter using one of the functions nni_put_parm or nni_put_parm_array.
- Turn each parameter into a parm_description structure using the function nni_get_parm_info.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
chType	Type of the Natural module. Always N (for subprogram).
szName	Name of the Natural module.
pparmhandle	Points a to a pointer to a parameter set on return.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
> 0	Natural error number.

nni_create_method_parm

Syntax

```
int nni_create_method_parm( void* pnni_func, const char* szClass, const char* ↩
szMethod, void** pparmhandle ); ↩
```

Creates a set of parameters that can be used in a call to `nni_send_method`. The function enables an application to dynamically explore the signature of a method of a Natural class.

The returned parameter set contains not only the method parameters, but also the other parameters required by `nni_send_method`. This means: If the method has n parameters, the parameter set contains $n + 2$ parameters.

- The first parameter in the set is initialized to the data type `HANDLE OF OBJECT`.
- The second parameter in the set is initialized to the data type of the method return value. If the method does not have a return value, the second parameter is not initialized.
- The remaining parameters in the set are initialized to the data types of the method parameters.

Before using the parameter set in a call to `nni_send_method`:

- Assign a value to each parameter using one of the functions `nni_put_parm` or `nni_put_parm_array`.
- Turn each parameter into a `parm_description` structure using the function `nni_get_parm_info`.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
szClass	Name of the Natural class.
szMethod	Name of the Natural method.
pparmhandle	Points a to a pointer to a parameter set on return.

Return Codes

The meaning of the return codes is explained in the section *Return Codes*.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
> 0	Natural error number.

nni_create_prop_parm

Syntax

```
int nni_create_prop_parm(void* pnni_func, const char* szClass, const char* ↩
szProp,void** pparmhandle); ↩
```

Creates a set of parameters that can be used in a call to `nni_get_property` or `nni_set_property`. The returned parameter set contains all parameters required by `nni_get_property` or `nni_set_property`. The function enables an application to determine the data type of a property of a Natural class.

- The first parameter in the set is initialized to the data type `HANDLE OF OBJECT`.
- The second parameter in the set is initialized to the data type of the property.

Before using the parameter set in a call to `nni_get_property` or `nni_set_property`:

- Assign a value to each parameter using one of the functions `nni_put_parm` or `nni_put_parm_array`.
- Turn each parameter into a `parm_description` structure using the function `nni_get_parm_info`.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
szClass	Name of the Natural class.
szProp	Name of the Natural property.
pparmhandle	Points a to a pointer to a parameter set on return.

Return Codes

The meaning of the return codes is explained in the section *Return Codes*.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
> 0	Natural error number.

nni_parm_count

Syntax

```
int nni_parm_count( void* pnni_func, void* parmhandle, int* piParm )
```

The function retrieves the number of parameters in a parameter set.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
parmhandle	Pointer to a parameter set.
piParm	Returns the number of parameters in the parameter set.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	

nni_init_parm_s

Syntax

```
int nni_init_parm_s(void* pnni_func, int iParm, void* parmhandle, char chFormat, ↩
int iLength, int iPrecision, int iFlags); ↩
```

Initializes a parameter in a parameter set to a static data type.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iParm	Index of the parameter. The first parameter in the set has the index 0.
parmhandle	Pointer to a parameter set.
chFormat	Natural data type of the parameter.
iLength	Natural length of the parameter.
iPrecision	Number of decimal places (NNI_TYPE_NUM and NNI_TYPE_PACK only).
iFlags	Parameter flags. The following flags can be used: NNI_FLG_PROTECTED

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_NO_MEMORY	
NNI_RC_BAD_FORMAT	
NNI_RC_BAD_LENGTH	

nni_init_parm_sa

Syntax

```
int nni_init_parm_sa (void* pnni_func, int iParm, void* parmhandle, char chFormat, ↩
int iLength, int iPrecision, int iDim, int* rgiOcc, int iFlags); ↩
```

Initializes a parameter in a parameter set to an array of a static data type.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iParm	Index of the parameter. The first parameter in the set has the index 0.
parmhandle	Pointer to a parameter set.
chFormat	Natural data type of the parameter.
iLength	Natural length of the parameter.
iPrecision	Number of decimal places (NNI_TYPE_NUM and NNI_TYPE_PACK only).
iDim	Array dimension of the parameter.
rgiOcc	Three dimensional array of int values, indicating the occurrence count for each dimension. The occurrence count for unused dimensions must be specified as 0.
iFlags	Parameter flags. The following flags can be used: NNI_FLG_PROTECTED NNI_FLG_LBVAR_0 NNI_FLG_UBVAR_0 NNI_FLG_LBVAR_1 NNI_FLG_UBVAR_1 NNI_FLG_LBVAR_2 NNI_FLG_UBVAR_2 If one of the NNI_FLG_*VAR* flags is set, the array is an x-array. In each dimension only the lower bound or the upper bound (not both) can be variable. Therefore for instance the flag IF4_FLG_LBVAR_0 may not be combined with IF4_FLG_UBVAR_0.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_NO_MEMORY	
NNI_RC_BAD_FORMAT	
NNI_RC_BAD_LENGTH	
NNI_RC_BAD_DIM	
NNI_RC_BAD_BOUNDS	

nni_init_parm_d

Syntax

```
int nni_init_parm_d(void* pnni_func, int iParm, void* parmhandle, char chFormat, ↩
int iFlags); ↩
```

Initializes a parameter in a parameter set to a dynamic data type.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iParm	Index of the parameter. The first parameter in the set has the index 0.
parmhandle	Pointer to a parameter set.
chFormat	Natural data type of the parameter (NNI_TYPE_ALPHA or NNI_TYPE_BIN).
iFlags	Parameter flags. The following flags can be used: NNI_FLG_PROTECTED

Return Codes

The meaning of the return codes is explained in the section *Return Codes*.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_NO_MEMORY	
NNI_RC_BAD_FORMAT	

nni_init_parm_da

Syntax

```
int nni_init_parm_da (void* pnni_func, int iParm, void* parmhandle, char chFormat, ↩
int iDim, int* rgiOcc, int iFlags); ↩
```

Initializes a parameter in a parameter set to an array of a dynamic data type.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iParm	Index of the parameter. The first parameter in the set has the index 0.
parmhandle	Pointer to a parameter set.
chFormat	Natural data type of the parameter (NNI_TYPE_ALPHA or NNI_TYPE_BIN).
iDim	Array dimension of the parameter.
rgiOcc	Three dimensional array of int values, indicating the occurrence count for each dimension. The occurrence count for unused dimensions must be specified as 0.
iFlags	Parameter flags. The following flags can be used: NNI_FLG_PROTECTED NNI_FLG_LBVAR_0 NNI_FLG_UBVAR_0 NNI_FLG_LBVAR_1 NNI_FLG_UBVAR_1 NNI_FLG_LBVAR_2 NNI_FLG_UBVAR_2 If one of the NNI_FLG_*VAR* flags is set, the array is an x-array. In each dimension only the lower bound or the upper bound (not both) can be variable. Therefore for instance the flag IF4_FLG_LBVAR_0 may not be combined with IF4_FLG_UBVAR_0.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_NO_MEMORY	
NNI_RC_BAD_FORMAT	
NNI_RC_BAD_DIM	
NNI_RC_BAD_BOUNDS	

nni_get_parm_info

Syntax

```
int nni_get_parm_info (void* pnni_func, int iParm, void* parmhandle, struct ↩
parameter_description* pDesc); ↩
```

Returns detailed information about a specific parameter in a parameter set.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iParm	Index of the parameter. The first parameter in the set has the index 0.
parmhandle	Pointer to a parameter set.
pDesc	Parameter description structure.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	

nni_get_parm

Syntax

```
int nni_get_parm(void* pnni_func, int iParm, void* parmhandle, int iBufferLength, ↩
void* pBuffer); ↩
```

Returns the value of a specific parameter in a parameter set. The value is returned in the buffer at the address specified in `pBuffer`, with the size specified in `iBufferLength`. On successful return, the buffer contains the data in Natural internal format. See ***Natural Data Types*** on how to interpret the contents of the buffer.

If the length of the parameter according to the Natural data type is greater than `iBufferLength`, Natural truncates the data to the given length and returns the code `NNI_RC_DATA_TRUNC`. The caller can use the function `nni_get_parm_info` to request the length of the parameter value in advance.

If the length of the parameter according to the Natural data type is smaller than `iBufferLength`, Natural fills the buffer according to the length of the parameter and returns the length of the copied data in the return code.

If the parameter is an array, the function returns the whole array in the buffer. This makes sense only for fixed size arrays of fixed size elements, because in other cases the caller cannot interpret the contents of the buffer. In order to retrieve an individual occurrence of an arbitrary array use the function `nni_get_parm_array`.

If no memory of the size specified in `iBufferLength` is allocated at the address specified in `pBuffer`, the results of the operation are unpredictable. Natural only checks that `pBuffer` is not null.

Parameters

Parameter	Meaning
`pnni_func`	Pointer to an NNI interface instance.
`iParm`	Index of the parameter. The first parameter in the set has the index 0.
`parmhandle`	Pointer to a parameter set.
`iBufferLength`	Length of the buffer specified in `pBuffer`.
`pBuffer`	Buffer in which the value is returned.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
`NNI_RC_OK`	
`NNI_RC_NOT_INIT`	
`NNI_RC_PARM_ERROR`	
`NNI_RC_ILL_PNUM`	
`NNI_RC_DATA_TRUNC`	
= *n*, where *n* > 0	Successful operation, but only *n* bytes were returned in the buffer.

nni_get_parm_array

Syntax

```
int nni_get_parm_array(void* pnni_func, int parmnum, void* parmhandle, int ↩
iBufferLength, void* pBuffer, int* rgiInd);
```

Returns the value of a specific occurrence of a specific array parameter in a parameter set. The only difference to `nni_get_parm` is that array indices can be specified. The indices for unused dimensions must be specified as 0.

Parameters

Parameter	Meaning
`pnni_func`	Pointer to an NNI interface instance.
`iParm`	Index of the parameter. The first parameter in the set has the index 0.
`parmhandle`	Pointer to a parameter set.
`iBufferLength`	Length of the buffer specified in `pBuffer`.
`pBuffer`	Buffer in which the value is returned.
`rgiInd`	Three dimensional array of int values, indicating a specific array occurrence. The indices start with 0.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
`NNI_RC_OK`	
`NNI_RC_NOT_INIT`	
`NNI_RC_PARM_ERROR`	
`NNI_RC_ILL_PNUM`	
`NNI_RC_DATA_TRUNC`	
`NNI_RC_NOT_ARRAY`	
`NNI_RC_BAD_INDEX_0`	
`NNI_RC_BAD_INDEX_1`	
`NNI_RC_BAD_INDEX_2`	
= *n*, where $n > 0$	Successful operation, but only *n* bytes were returned.

nni_get_parm_array_length

Syntax

```
int nni_get_parm_array_length(void* pnni_func, int iParm, void* parmhandle, int* ↩
piLength, int* rgiInd);
```

Returns the length of a specific occurrence of a specific array parameter in a parameter set.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iParm	Index of the parameter. The first parameter in the set has the index 0.
parmhandle	Pointer to a parameter set.
piLength	Pointer to an int in which the length of the value is returned.
rgiInd	Three dimensional array of int values, indicating a specific array occurrence. The indices start with 0.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_ILL_PNUM	
NNI_RC_DATA_TRUNC	
NNI_RC_NOT_ARRAY	
NNI_RC_BAD_INDEX_0	
NNI_RC_BAD_INDEX_1	
NNI_RC_BAD_INDEX_2	

nni_put_parm

Syntax

```
int nni_put_parm(void* pnni_func, int iParm, void* parmhandle, int iBufferLength, ↩
const void* pBuffer); ↩
```

Assigns a value to a specific parameter in a parameter set. The value is passed to the function in the buffer at the address specified in `pBuffer`, with the size specified in `iBufferLength`. See ***Natural Data Types*** on how to prepare the contents of the buffer.

If the length of the parameter according to the Natural data type is smaller than the given buffer length, the data will be truncated to the length of the parameter. The rest of the buffer will be ignored. If the length of the parameter according to the Natural data type is greater than the given buffer length, the data will copied only to the given buffer length, the rest of the parameter value stays unchanged. See ***Natural Data Types*** on the internal length of Natural data types.

If the parameter is a dynamic variable, it is automatically resized according to the given buffer length.

If the parameter is an array, the function expects the whole array in the buffer. This makes sense only for fixed size arrays of fixed size elements, because in other cases the caller cannot provide the correct contents of the buffer. In order to assign a value to an individual occurrence of an arbitrary array use the function `nni_put_parm_array`.

Parameters

Parameter	Meaning
`pnni_func`	Pointer to an NNI interface instance.
`iParm`	Index of the parameter. The first parameter in the set has the index 0.
`parmhandle`	Pointer to a parameter set.
`iBufferLength`	Length of the buffer specified in `pBuffer`.
`pBuffer`	Buffer in which the value is passed.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_WRT_PROT	
NNI_RC_DATA_TRUNC	
NNI_RC_NO_MEMORY	
= *n*, where $n > 0$	Successful operation, but only *n* bytes of the buffer were used.

nni_put_parm_array

Syntax

```
int nni_put_parm_array(void* pnni_func, int iParm, void* parmhandle, int ↩
iBufferLength, const void* pBuffer, int* rgiInd);
```

Assigns a value to a specific occurrence of a specific array parameter in a parameter set. The only difference to `nni_get_parm` is that array indices can be specified. The indices for unused dimensions must be specified as 0.

Parameters

Parameter	Meaning
`pnni_func`	Pointer to an NNI interface instance.
`iParm`	Index of the parameter. The first parameter in the set has the index 0.
`parmhandle`	Pointer to a parameter set.
`iBufferLength`	Length of the buffer specified in `pBuffer`.
`pBuffer`	Buffer in which the value is passed.
`rgiInd`	Three dimensional array of int values, indicating a specific array occurrence. The indices start with 0.

Return Codes

The meaning of the return codes is explained in the section *Return Codes*.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_WRT_PROT	
NNI_RC_DATA_TRUNC	
NNI_RC_NO_MEMORY	
NNI_RC_NOT_ARRAY	
NNI_RC_BAD_INDEX_0	
NNI_RC_BAD_INDEX_1	
NNI_RC_BAD_INDEX_2	
= *n*, where *n* > 0	Successful operation, but only *n* bytes of the buffer were used.

nni_resize_parm_array

Syntax

```
int nni_resize_parm_array(void* pnni_func, int iParm, void* parmhandle, int* rgiOcc); ↩
```

Changes the occurrence count of a specific x-array parameter in a parameter set. For an *n*-dimensional array an occurrence count must be specified for all *n* dimensions. If the dimension of the array is less than 3, the value 0 must be specified for the not used dimensions.

The function tries to resize the occurrence count of each dimension either by changing the lower bound or the upper bound, whatever is appropriate for the given x-array.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iParm	Index of the parameter. The first parameter in the set has the index 0.
parmhandle	Pointer to a parameter set.
rgiOcc	Three dimensional array of int values, indicating the new occurrence count of the array.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_WRT_PROT	
NNI_RC_DATA_TRUNC	
NNI_RC_NO_MEMORY	
NNI_RC_NOT_ARRAY	
NNI_RC_NOT_RESIZABLE	
> 0	Natural error number.

nni_delete_parm

Syntax

```
int nni_delete_parm(void* pnni_func, void* parmhandle);
```

Deletes the specified parameter set.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
parmhandle	Pointer to a parameter set.

Return Codes

The meaning of the return codes is explained in the section ***Return Codes***.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	

nni_from_string

Syntax

```
int nni_from_string(void* pnni_func, const char* szString, char chFormat, int ↩
iLength, int iPrecision, int iBufferLength, void* pBuffer); ↩
```

Converts the string representation of a Natural P, N, D or T value into the internal representation of the value, as it is used in the functions `nni_get_parm`, `nni_get_parm_array`, `nni_put_parm` and `nni_put_parm_array`.

The string representations of these Natural data types look like this:

Format	String representation
P, N	For example, `-3.141592`, where the decimal character defined in the `DC` parameter is used.
D	Date format as defined in the `DTFORM` parameter, (e. g. “2004-07-06”, if `DTFORM=I`).
T	Date format as defined in the `DTFORM` parameter, combined with a Time value in the form hh:ii:ss:t (e. g. `2004-07-06 11:30:42:7`, if `DTFORM=I`) or Time value in the form hh:ii:ss:t (e. g. `11:30:42:7`).

Parameters

Parameter	Meaning
`pnni_func`	Pointer to an NNI interface instance.
`szString`	String representation of the value.
`chFormat`	Natural data type of the value.
`iLength`	Natural length of the value. The total number of significant digits in the case of `NNI_TYPE_NUM` and `NNI_TYPE_PACK`, 0 otherwise.
`iPrecision`	Number of decimal places in the case of `NNI_TYPE_NUM` and `NNI_TYPE_PACK`, 0 otherwise.
`iBufferLength`	Length of the buffer provided in `pBuffer`.
`pBuffer`	Buffer that contains the internal representation of the value on return. The buffer must be large enough to hold the internal Natural representation of the value. The required sizes are documented in *Format and Length of User-Defined Variables*.

Return Codes

The meaning of the return codes is explained in the section *Return Codes*.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
> 0	Natural error number

nni_to_string

Syntax

```
int nni_to_string(void* pnni_func, int iBufferLength, const void* pBuffer, char ↩
chFormat, int iLength, int iPrecision, int iStringLength, char* szString);
```

Converts the internal representation of a Natural P, N, D or T value, as it is used in the functions `nni_get_parm`, `nni_get_parm_array`, `nni_put_parm` and `nni_put_parm_array`, into a the string representation.

The string representations of these Natural data types look as described with the function `nni_from_string`.

Parameters

Parameter	Meaning
pnni_func	Pointer to an NNI interface instance.
iBufferLength	Length of the buffer provided in pBuffer.
pBuffer	Buffer that contains the internal representation of the value. The required sizes are documented in *Format and Length of User-Defined Variables*.
chFormat	Natural data type of the value.
iLength	Natural length of the value. The total number of significant digits in the case of NNI_TYPE_NUM and NNI_TYPE_PACK, 0 otherwise.
iPrecision	Number of decimal places in the case of NNI_TYPE_NUM and NNI_TYPE_PACK, 0 otherwise.
iStringLength	Length of the string buffer provided in szString including the terminating zero.
szString	String buffer that contains the string representation of the value on return. The string buffer must be large enough to hold the external representation including the terminating zero.

Return Codes

The meaning of the return codes is explained in the section *Return Codes*.

Return Code	Remark
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
> 0	Natural error number

66 Parameter Description Structure

The interface provides information about the parameters of a Natural subprogram or method in a structure named `parameter_description`. The structure is defined in the header file *natuser.h*. This file is contained in the directory `<install-dir>`*/natural/samples/sysexnni*.

An array of `parameter_description` structures is passed to the interface with each call to `nni_callnat` and similar functions. A `parameter_description` structure is created from a parameter in a parameter set using the function `nni_get_parm_info`.

The relevant elements of the structure contain the following information. All elements not listed in this table are for internal use only.

Format	Element Name	Content
void*	address	Address of the parameter value. Must not be reallocated or freed. The address element is a null pointer for arrays of dynamic variables and for x-arrays. In these cases, the array data cannot be accessed as a whole, but can only be accessed elementwise through the parameter access function `nni_get_parm`.
int	format	Natural data type of the parameter. Refer to ***Natural Data Types*** for further information.
int	length	Natural length of the parameter value. In the case of the data types `NNI_TYPE_ALPHA` and `NNI_TYPE_UNICODE`, the number of characters. In the case of the data types `NNI_TYPE_PACK` and `NNI_TYPE_NUM`, the number of digits before the decimal character. In the case of an array, the length of a single occurrence. In the case of an array of dynamic variables, the length is indicated with 0. The length of an individual occurrence must then be determined with the function `nni_get_parm_array_length`.
int	precision	In the case of the data types `NNI_TYPE_PACK` and `NNI_TYPE_NUM` the number of digits after the decimal character, 0 otherwise.
int	byte_length	Length of the parameter value in bytes. In the case of an array the byte length of a single occurrence. In the case of an array of dynamic variables the byte length is indicated with 0. The length of an individual occurrence must then be determined with the function `nni_get_parm_array_length`.

Format	Element Name	Content
int	dimensions	Number of dimensions. 0 in the case of a scalar. The maximum number of dimensions is 3.
int	length_all	Total length of the parameter value in bytes. In the case of an array the byte length of the whole array. In the case of an array of dynamic variables the total length is indicated with 0. The length of an individual occurrence must then be determined with the function `nni_get_parm_array_length`.
int	flags	Parameter flags, see ***Flags***.
int	occurrences[10]	Number of occurrences in each dimension. Only the first three occurrences are used.
int	indexfactors[10]	Array index factors for each dimension. Only the first three occurrences are used.

In the case of arrays with fixed bounds of variables with fixed length, the array contents can be accessed directly using the structure element `address`. In these cases the following applies:

- The address of the element (i,j,k) of a three dimensional array is computed as follows:

  elementaddress = address + i * indexfactors[0] + j * indexfactors[1] + k * indexfactors[2]

- The address of the element (i,j) of a two dimensional array is computed as follows:

  elementaddress = address + i * indexfactors[0] + j * indexfactors[1]

- The address of the element (i) of a one dimensional array is computed as follows:

  elementaddress = address + i * indexfactors[0]

67 Natural Data Types

Some of the parameter access functions (like `nni_get_parm`, `nni_put_parm`) use a buffer that contains a parameter value in the correct representation. The length of the buffer depends on the Natural data type. The data format of the buffer is defined according to the following table:

Natural Data Type	Buffer Format
A	char[ ]
B	byte[ ]
C	short
F4	float
F8	double
I1	signed char
I2	short
I4	int
L	`NNI_L_TRUE` or `NNI_L_FALSE`, see *natni.h*
HANDLE OF OBJECT	byte[8]
P, N, D, T	The buffer content should be created from a string representation with the function `nni_from_string`. It can be transformed to a string representation with the function `nni_to_string`.
U	An array of UTF-16 characters. On Windows and on those Linux platforms where a wchar corresponds to a UTF-16 character, this is a wchar[].

Some of the parameter access functions (like `nni_get_parm`, and `nni_put_parm`) require a Natural data type to be specified. In these cases the following constants should be used. The constants are defined in the header file *natni.h*. This file is contained in the directory `<install-dir>`*/natural/samples/sysexnni*.

Natural Data Type	Constant
A	NNI_TYPE_ALPHA
B	NNI_TYPE_BIN
C	NNI_TYPE_CV
D	NNI_TYPE_DATE
F	NNI_TYPE_FLOAT
I	NNI_TYPE_INT
L	NNI_TYPE_LOG
N	NNI_TYPE_NUM
HANDLE OF OBJECT	NNI_TYPE_OBJECT
P	NNI_TYPE_PACK
T	NNI_TYPE_TIME
U	NNI_TYPE_UNICODE

68 Flags

The structure parameter_description has an element flags that contains information about the status of the parameter. Also the functions nni_init_parm* allow specifying some of these flags when initializing a parameter. The individual flags can be combined with a logical OR in the element flags. The following flags are defined in the header file *natni.h*. This file is contained in the directory *<install-dir>/natural/samples/sysexnni*.

Return Code	Meaning
NNI_FLG_PROTECTED	Parameter is write protected.
NNI_FLG_DYNAMIC (*)	Parameter is dynamic (variable length or x-array).
NNI_FLG_NOT_CONTIG (*)	Array is not contiguous.
NNI_FLG_AIV (*)	Parameter is an AIV or INDEPENDENT variable.
NNI_FLG_DYNVAR (*)	Parameter has variable length.
NNI_FLG_XARRAY (*)	Parameter is an x-array.
NNI_FLG_LBVAR_0	Lower bound of dimension 0 is variable.
NNI_FLG_UBVAR_0	Upper bound of dimension 0 is variable.
NNI_FLG_LBVAR_1	Lower bound of dimension 1 is variable.
NNI_FLG_UBVAR_1	Upper bound of dimension 1 is variable.
NNI_FLG_LBVAR_2	Lower bound of dimension 2 is variable.
NNI_FLG_UBVAR_2	Upper bound of dimension 2 is variable.

Only the flags marked with (*) can be explicitly set in the functions nni_init_parm*. The other flags are automatically set by the interface according to the type of the parameter.

If one of the NNI_FLG_*VAR* flags is set, the array is an x-array. In each dimension of an x-array only the lower bound or the upper bound, not both, can be variable. Therefore for instance the flag NNI_FLG_LBVAR_0 may not be combined with NNI_FLG_UBVAR_0.

If NNI_FLG_DYNAMIC is on, also NNI_FLG_DYNVAR, NNI_FLG_XARRAY or both are on. If both are on, the parameter is an x-array with elements of variable length.

69 Return Codes

The interface functions return the following return codes. The constants are defined in the header file *natni.h*. This file is contained in the directory `<install-dir>`*/natural/samples/sysexnni*.

Return Code	Meaning
`NNI_RC_OK`	Successful execution.
`NNI_RC_ILL_PNUM`	Invalid parameter number.
`NNI_RC_INT_ERROR`	Internal error.
`NNI_RC_DATA_TRUNC`	Data has been truncated during parameter value access.
`NNI_RC_NOT_ARRAY`	Parameter is not an array.
`NNI_RC_WRT_PROT`	Parameter is write protected.
`NNI_RC_NO_MEMORY`	Memory allocation failed.
`NNI_RC_BAD_FORMAT`	Invalid Natural data type.
`NNI_RC_BAD_LENGTH`	Invalid length or precision.
`NNI_RC_BAD_DIM`	Invalid dimension count.
`NNI_RC_BAD_BOUNDS`	Invalid x-array bound definition.
`NNI_RC_NOT_RESIZABLE`	Array cannot be resized in the requested way.
`NNI_RC_BAD_INDEX_0`	Index for array dimension 0 out of range.
`NNI_RC_BAD_INDEX_1`	Index for array dimension 1 out of range.
`NNI_RC_BAD_INDEX_2`	Index for array dimension 2 out of range.
`NNI_RC_VERSION_ERROR`	Requested interface version not supported.
`NNI_RC_NOT_INIT`	No Natural session initialized in this interface instance.
`NNI_RC_NOT_IMPL`	Function not implemented in this interface version.
`NNI_RC_PARM_ERROR`	Mandatory parameter not specified.
`NNI_RC_LOCKED`	Interface instance is locked by another thread.
rc, where *rc* < `NNI_RC_SERR_OFFSET`	Natural startup error occurred. The Natural startup error number as documented in *Natural Startup Errors* (which is part of the

Return Code	Meaning
	Operations documentation) can be determined from the return code by the following calculation: `startup-error-nr = - (rc - NNI_RC_SERR_OFFSET)`
> 0	Natural error number.

70 Natural Exception Structure

The interface functions that execute Natural code (such as `nni_callnat`) return a structure named `natural_exception` that contains further information about a Natural error that might have occurred. The structure is defined in the header file *natni.h*. This file is contained in the directory `<install-dir>`*/natural/samples/sysexnni*.

The elements of the structure contain the following information.

Format	Element Name	Content
int	natMessageNumber	Natural message number.
char	natMessageText[NNI_LEN_TEXT+1]	Natural message text with all replacements.
char	natLibrary[NNI_LEN_LIBRARY+1]	Natural library name.
char	natMember[NNI_LEN_MEMBER+1]	Natural member name.
char	natName[NNI_LEN_NAME+1];	Natural function, subroutine or class name.
char	natMethod[NNI_LEN_NAME+1];	Natural method or property name.
int	int natLine;	Natural code line where the error occurred.

71 Interface Usage

The interface is typically used in the following way (example: Call a Natural subprogram):

1. Determine the location of the Natural binaries.
2. Load the Natural Native Interface library.
3. Call `nni_get_interface` to retrieve an interface instance.
4. Call `nni_initialize` to initialize a Natural session.
5. Call `nni_logon` to logon to a specific Natural library.
6. Call `nni_create_parm` or a related function to create a set of parameters.
7. For each parameter
   - Call one of the `nni_init_parm` functions to initialize the parameter to the correct type.
   - Call one of the `nni_put_parm` functions to assign a value to the parameter.
   - Call `nni_get_parm_info` to create the `parameter_description` structure.
8. Call `nni_callnat` to call the subprogram.
9. For each modifiable parameter.
   - Call one of the `nni_get_parm` functions to retrieve the parameter value.
10. Call `nni_delete_parm` to free the parameter structures.
11. Call `nni_uninitialize` to uninitialize the Natural session.
12. Call `nni_logoff` to return to the previous library.
13. Call `nni_free_interface` to free the interface instance.

An example C program *nnisample.c* that shows the usage of the interface is contained in `<install-dir>`*/natural/samples/sysexnni*.

72 Threading Issues

A Natural process on Windows or Linux always contains only one thread that executes Natural code. Thus in an interactively started Natural session, it can never occur that several threads try to execute Natural code in parallel. The situation is different when a client program that runs several threads in parallel uses the Natural Native Interface.

The Natural Native Interface can be used by multithreaded applications. The interface functions are thread safe. As long as a given thread T is executing one of the interface functions, other threads of the same process that call one of the interface functions are blocked until T has left the interface function. Effectively the parallel executing threads of the process are serialized as far as the usage of the interface functions is concerned. It is not necessary to serialize interface access among the threads of different processes, because each different process that uses the NNI runs its own Natural session.

The calling application can also control the multithreaded access to the NNI explicitly. This can make sense if a thread wants to execute a series of NNI calls without being interrupted by another thread. To achieve this, the thread calls `nni_enter`, which lets the thread wait until all other threads have left the NNI. Then the thread does its work and calls NNI functions at will. After having finished its work, the thread calls `nni_leave` to allow other threads to access the NNI.

A multithreaded application that uses the NNI must follow these rules:

- The functions `nni_initialize` and `nni_uninitialize` must be called at least once per process.
- The function `nni_uninitialize` must be called on the same thread as the corresponding call to `nni_initialize`.
- The function `nni_uninitialize` must not be called before the last thread that uses the NNI has terminated.

XII NaturalX

This part describes how to develop object-based applications.

The following topics are covered:

73 Introduction to NaturalX

This chapter contains a short introduction to component-based programming involving the use of the NaturalX interface and a dedicated set of Natural statements.

Why NaturalX?

Software applications that are based on component architecture offer many advantages over traditional designs. These include the following:

- Faster development. Programmers can build applications faster by assembling software from prebuilt components.
- Reduced development costs. Having a common set of interfaces for programs means less work integrating the components into complete solutions.
- Improved flexibility. It is easier to customize software for different departments within a company by just changing some of the components that constitute the application.
- Reduced maintenance costs. In the case of an upgrade, it is often sufficient to change some of the components instead of having to modify the entire application.

Using NaturalX you can create component-based applications.

You can use NaturalX to apply a component-based programming style. However, on this platform the components cannot be distributed and can only run in a local Natural session.

74 Developing NaturalX Applications

This chapter describes how to develop an application by defining and using classes.

Development Environments

- **Developing Classes on Windows Platforms**
  On Windows platforms, Natural provides the Class Builder as the tool to develop Natural classes. The Class Builder shows a Natural class in a structured hierarchical order and allows the user to manage the class and its components efficiently. If you use the Class Builder, no knowledge or only a basic knowledge of the syntax elements described below is required.
- **Developing Classes Using SPoD**
  In a Natural Single Point of Development (SPoD) environment that includes a Mainframe and/or Linux remote development server, you can use the Class Builder available with the Natural Studio front-end to develop classes on Mainframe and/or Linux platforms. In this case, no knowledge or only a basic knowledge of the syntax elements described below is required.
- **Developing Classes on Mainframe or Linux Platforms**
  If you do not use SPoD, you develop classes on these platforms using the Natural program editor. In this case, you should know the syntax of class definition described below.

Defining Classes

When you define a class, you must create a Natural class module, within which you create a `DEFINE CLASS` statement. Using the `DEFINE CLASS` statement, you assign the class an externally usable name and define its interfaces, methods and properties. You can also assign an object data area to the class, which describes the layout of an instance of the class.

This section covers the following topics:

- Implementing Methods

Creating a Natural Class Module

To create a Natural class module

- Use the `CREATE OBJECT` statement to create a Natural object of type Class.

Specifying a Class

The `DEFINE CLASS` statement defines the name of the class, the interfaces the class supports and the structure of its objects.

To specify a class

- Use the `DEFINE CLASS` statement as described in the *Statements* documentation.

Defining an Interface

Each interface of a class is specified with an `INTERFACE` statement inside the class definition. An `INTERFACE` statement specifies the name of the interface and a number of properties and methods. For classes that are to be registered as COM classes, it specifies also the globally inique ID of the interface.

A class can have one or several interfaces. For each interface, one `INTERFACE` statement is coded in the class definition. Each `INTERFACE` statement contains one or several `PROPERTY` and `METHOD` clauses. Usually the properties and methods contained in one interface are related from either a technical or a business point of view.

The `PROPERTY` clause defines the name of a property and assigns a variable from the object data area to the property. This variable is used to store the value of the property.

The `METHOD` clause defines the name of a method and assigns a subprogram to the method. This subprogram is used to implement the method.

To define an interface

- Use the `INTERFACE` statement as described in the *Statements* documentation.

Assigning an Object Data Variable to a Property

The `PROPERTY` statement is used only when several classes are to implement the same interface in different ways. In this case, the classes share the same interface definition and include it from a Natural **copycode**. The `PROPERTY` statement is then used to assign a variable from the object data area to a property, *outside* the interface definition. Like the `PROPERTY` clause of the `INTERFACE` statement, the `PROPERTY` statement defines the name of a property and assigns a variable from the object data area to the property. This variable is used to store the value of the property.

To assign an object data variable to a property

- Use the `PROPERTY` statement as described in the *Statements* documentation.

Assigning a Subprogram to a Method

The `METHOD` statement is used only when several classes are to implement the same interface in different ways. In this case, the classes share the same interface definition and include it from a Natural **copycode**. The `METHOD` statement is then used to assign a subprogram to the method, *outside* the interface definition. Like the `METHOD` clause of the `INTERFACE` statement, the `METHOD` statement defines the name of a method and assigns a subprogram to the method. This subprogram is used to implement the method.

To assign a subprogram to a method

- Use the `METHOD` statement as described in the *Statements* documentation.

Implementing Methods

A method is implemented as a Natural subprogram in the following general form:

```
DEFINE DATA statement
*
* Implementation code of the method
*
END
```

For information on the `DEFINE DATA` statement see the *Statements* documentation.

All clauses of the `DEFINE DATA` statement are optional.

It is recommended that you use data areas instead of inline data definitions to ensure data consistency.

If a `PARAMETER` clause is specified, the method can have parameters and/or a return value.

Parameters that are marked `BY VALUE` in the parameter data area are input parameters of the method.

Parameters that are not marked `BY VALUE` are passed “by reference” and are input/output parameters. This is the default.

The first parameter that is marked `BY VALUE RESULT` is returned as the return value for the method. If more than one parameter is marked in this way, the others will be treated as input/output parameters.

Parameters that are marked `OPTIONAL` need not be specified when the method is called. They can be left unspecified by using the *n*`X` notation in the `SEND METHOD` statement.

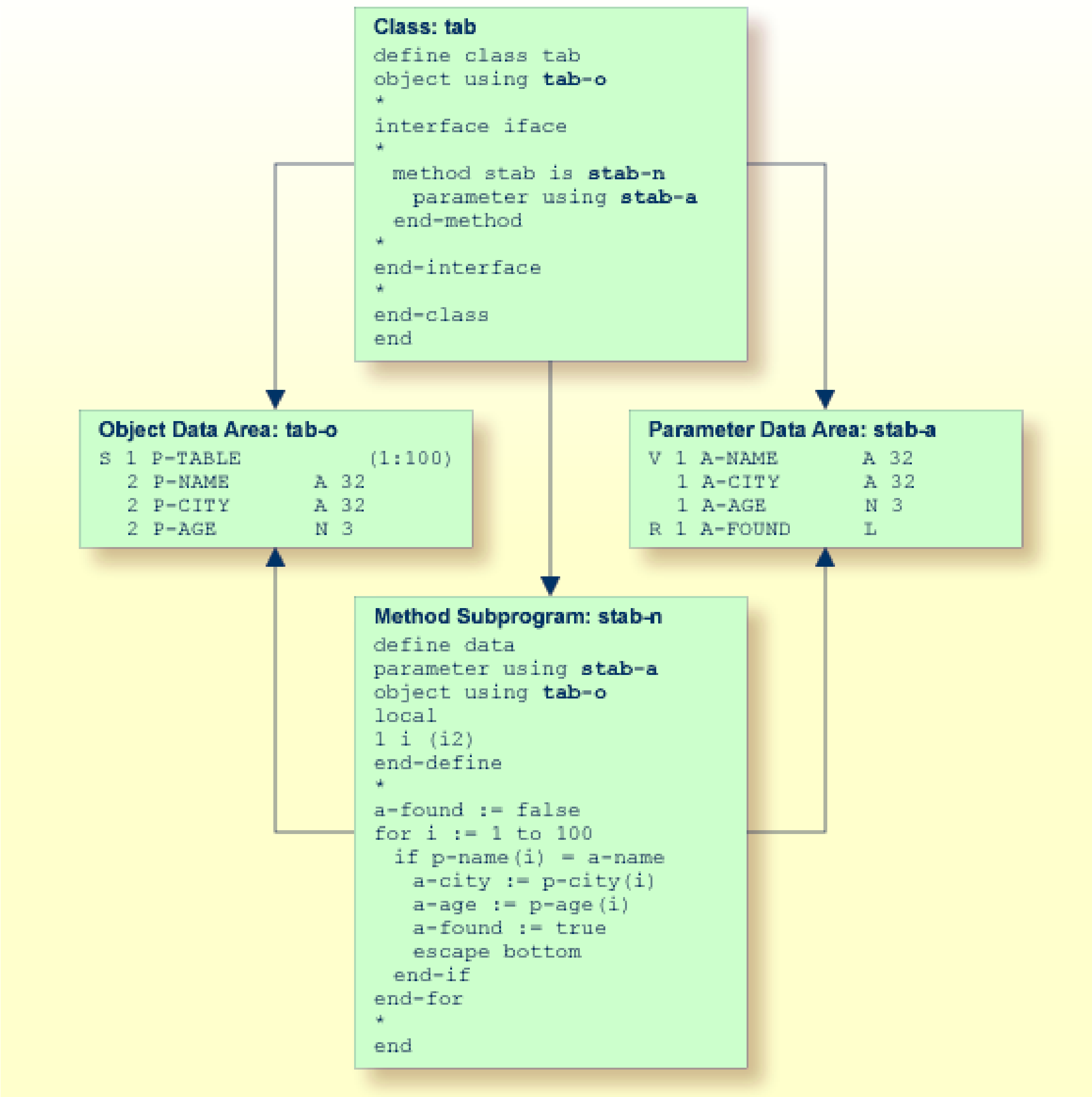
To make sure that the method subprogram accepts exactly the same parameters as specified in the corresponding `METHOD` statement in the class definition, use a parameter data area instead of inline data definitions. Use the same parameter data area as in the corresponding `METHOD` statement.

To give the method subprogram access to the object data structure, the `OBJECT` clause can be specified. To make sure that the method subprogram can access the object data correctly, use a local data area instead of inline data definitions. Use the same local data area as specified in the `OBJECT` clause of the `DEFINE CLASS` statement.

The `GLOBAL`, `LOCAL` and `INDEPENDENT` clauses can be used as in any other Natural program.

While technically possible, it is usually not meaningful to use a `CONTEXT` clause in a method subprogram.

The following example retrieves data about a given person from a table. The search key is passed as a `BY VALUE` parameter. The resulting data is returned through “by reference” parameters (“by reference” is the default definition). The return value of the method is defined by the specification `BY VALUE RESULT`.



Using Classes and Objects

Objects created in a local Natural session can be accessed by other modules in the same Natural session.

The statement `CREATE OBJECT` is used to create an object (also known as an instance) of a given class.

To reference objects in Natural programs, object handles have to be defined in the `DEFINE DATA` statement. Methods of an object are invoked with the statement `SEND METHOD`. Objects can have properties, which can be accessed using the normal assignment syntax.

These steps are described below:

- Defining Object Handles
- Creating an Instance of a Class
- Invoking a Particular Method of an Object
- Accessing Properties

Defining Object Handles

To reference objects in Natural programs, object handles have to be defined as follows in the `DEFINE DATA` statement:

```
DEFINE DATA
  level-handle-name [(array-definition)] HANDLE OF OBJECT
  ...
END-DEFINE
```

Example:

```
DEFINE DATA LOCAL
1 #MYOBJ1 HANDLE OF OBJECT
1 #MYOBJ2 (1:5) HANDLE OF OBJECT
END-DEFINE
```

Creating an Instance of a Class

To create an instance of a class

- Use the `CREATE OBJECT` statement as described in the *Statements* documentation.

Invoking a Particular Method of an Object

To invoke a particular method of an object

- Use the `SEND METHOD` statement as described in the *Statements* documentation.

Accessing Properties

Properties can be accessed using the `ASSIGN` (or `COMPUTE`) statement as follows:

```
ASSIGN operand1.property-name = operand2
ASSIGN operand2 = operand1.property-name
```

Object Handle - *`operand1`*

`operand1` must be defined as an object handle and identifies the object whose property is to be accessed. The object must already exist.

`operand2`

As *`operand2`*, you specify an operand whose format must be data transfer-compatible to the format of the property. Please refer to the **data transfer compatibility rules** for further information.

`property-name`

The name of a property of the object.

If the property name conforms to Natural identifier syntax, it can be specified as follows

```
create object #o1 of class "Employee"
  #age := #o1.Age
```

If the property name does not conform to Natural identifier syntax, it must be enclosed in angle brackets:

```
create object #o1 of class "Employee"
  #salary := #o1.<<%Salary>>
```

The property name can also be qualified with an interface name. This is necessary if the object has more than one interface containing a property with the same name. In this case, the qualified property name must be enclosed in angle brackets:

```
create object #o1 of class "Employee"
  #age := #o1.<<PersonalData.Age>>
```

Example:

```
define data
  local
  1 #i          (i2)
  1 #o          handle of object
  1 #p          (5) handle of object
  1 #q          (5) handle of object
  1 #salary     (p7.2)
  1 #history    (p7.2/1:10)
  end-define
  * ...
  * Code omitted for brevity.
  * ...
  * Set/Read the Salary property of the object #o.
  #o.Salary := #salary
  #salary := #o.Salary
  * Set/Read the Salary property of
  * the second object of the array #p.
  #p.Salary(2) := #salary
  #salary := #p.Salary(2)
  *
  * Set/Read the SalaryHistory property of the object #o.
  #o.SalaryHistory := #history(1:10)
  #history(1:10) := #o.SalaryHistory
  * Set/Read the SalaryHistory property of
  * the second object of the array #p.
  #p.SalaryHistory(2) := #history(1:10)
  #history(1:10) := #p.SalaryHistory(2)
  *
  * Set the Salary property of each object in #p to the same value.
  #p.Salary(*) := #salary
  * Set the SalaryHistory property of each object in #p
  * to the same value.
  #p.SalaryHistory(*) := #history(1:10)
  *
  * Set the Salary property of each object in #p to the value
  * of the Salary property of the corresponding object in #q.
  #p.Salary(*) := #q.Salary(*)
  * Set the SalaryHistory property of each object in #p to the value
  * of the SalaryHistory property of the corresponding object in #q.
  #p.SalaryHistory(*) := #q.SalaryHistory(*)
  *
  end
```

In order to use arrays of object handles and properties that have arrays as values correctly, it is important to know the following:

A property of an occurrence of an array of object handles is addressed with the following index notation:

```
#p.Salary(2) := #salary
```

A property that has an array as value is always accessed as a whole. Therefore no index notation is necessary with the property name:

```
#o.SalaryHistory := #history(1:10)
```

A property of an occurrence of an array of object handles which has an array as value is therefore addressed as follows:

```
#p.SalaryHistory(2) := #history(1:10)
```

XIII

75 Natural Reserved Keywords

This chapter contains a list of all keywords that are reserved in the Natural programming language.

Important: To avoid any naming conflicts, you are strongly recommended not to use Natural reserved keywords as names for variables.

Alphabetical List of Natural Reserved Keywords

The following list is an overview of Natural reserved keywords and is for general information only. In case of doubt, use the **keyword check** function of the compiler.

[**A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z**]

- A -

ABS
ABSOLUTE
ACCEPT
ACTION
ACTIVATION
AD
ADD
AFTER
AL
ALARM
ALL
ALPHA
ALPHABETICALLY
AND
ANY
APPL
APPLICATION
ARRAY
AS
ASC
ASCENDING
ASSIGN
ASSIGNING
ASYNC
AT
ATN
ATT
ATTRIBUTES
AUTH

AUTHORIZATION
AUTO
AVER
AVG

- B -

BACKOUT
BACKWARD
BASE
BEFORE
BETWEEN
BLOCK
BOT
BOTTOM
BREAK
BROWSE
BUT
BX
BY

- C -

CABINET
CALL
CALLDBPROC
CALLING
CALLNAT
CAP
CAPTIONED
CASE
CC
CD
CDID
CF
CHAR
CHARLENGTH
CHARPOSITION
CHILD
CIPH
CIPHER
CLASS
CLOSE
CLR
COALESCE
CODEPAGE

COMMAND
COMMIT
COMPOSE
COMPRESS
COMPUTE
CONCAT
CONDITION
CONST
CONSTANT
CONTEXT
CONTROL
CONVERSATION
COPIES
COPY
COS
COUNT
COUPLED
CS
CURRENT
CURSOR
CV

- D -

DATA
DATAAREA
DATE
DAY
DAYS
DC
DECIDE
DECIMAL
DEFINE
DEFINITION
DELETE
DELIMITED
DELIMITER
DELIMITERS
DESC
DESCENDING
DIALOG
DIALOG-ID
DIGITS
DIRECTION
DISABLED

DISP
DISPLAY
DISTINCT
DIVIDE
DL
DLOGOFF
DLOGON
DNATIVE
DNRET
DO
DOCUMENT
DOEND
DOWNLOAD
DU
DY
DYNAMIC

- E -

EDITED
EJ
EJECT
ELSE
EM
ENCODED
END
END-ALL
END-BEFORE
END-BREAK
END-BROWSE
END-CLASS
END-DECIDE
END-DEFINE
END-ENDDATA
END-ENDFILE
END-ENDPAGE
END-ERROR
END-FILE
END-FIND
END-FOR
END-FUNCTION
END-HISTOGRAM
ENDHOC
END-IF
END-INTERFACE

END-LOOP
END-METHOD
END-NOREC
END-PARAMETERS
END-PARSE
END-PROCESS
END-PROPERTY
END-PROTOTYPE
END-READ
END-REPEAT
END-RESULT
END-SELECT
END-SORT
END-START
END-SUBROUTINE
END-TOPPAGE
END-WORK
ENDING
ENTER
ENTIRE
ENTR
EQ
EQUAL
ERASE
ERROR
ERRORS
ES
ESCAPE
EVEN
EVENT
EVERY
EXAMINE
EXCEPT
EXISTS
EXIT
EXP
EXPAND
EXPORT
EXTERNAL
EXTRACTING

- F -

FALSE
FC

FETCH
FIELD
FIELDS
FILE
FILL
FILLER
FINAL
FIND
FIRST
FL
FLOAT
FOR
FORM
FORMAT
FORMATTED
FORMATTING
FORMS
FORWARD
FOUND
FRAC
FRAMED
FROM
FS
FULL
FUNCTION
FUNCTIONS

- G -

GC
GE
GEN
GENERATED
GET
GFID
GIVE
GIVING
GLOBAL
GLOBALS
GREATER
GT
GUI

- H -

HANDLE

HAVING
HC
HD
HE
HEADER
HEX
HISTOGRAM
HOLD
HORIZ
HORIZONTALLY
HOUR
HOURS
HW

- I -

IA
IC
ID
IDENTICAL
IF
IGNORE
IM
IMMEDIATE
IMPORT
IN
INC
INCCONT
INCDIC
INCDIR
INCLUDE
INCLUDED
INCLUDING
INCMAC
INDEPENDENT
INDEX
INDEXED
INDICATOR
INIT
INITIAL
INNER
INPUT
INSENSITIVE
INSERT
INT

INTEGER
INTERCEPTED
INTERFACE
INTERFACE4
INTERMEDIATE
INTERSECT
INTO
INVERTED
INVESTIGATE
IP
IS
ISN

- J -

JOIN
JSON
JUST
JUSTIFIED

- K -

KD
KEEP
KEY
KEYS

- L -

LANGUAGE
LAST
LC
LE
LEAVE
LEAVING
LEFT
LENGTH
LESS
LEVEL
LIB
LIBPW
LIBRARY
LIBRARY-PASSWORD
LIKE
LIMIT
LINDICATOR

LINES
LISTED
LOCAL
LOCKS
LOG
LOG-LS
LOG-PS
LOGICAL
LOOP
LOWER
LS
LT

- M -

MACROAREA
MAP
MARK
MASK
MAX
MC
MCG
MESSAGES
METHOD
MGID
MICROSECOND
MIN
MINUTE
MODAL
MODIFIED
MODULE
MONTH
MORE
MOVE
MOVING
MP
MS
MT
MULTI-FETCH
MULTIPLY

- N -

NAME
NAMED
NAMESPACE

NATIVE
NAVER
NC
NCOUNT
NE
NEWPAGE
NL
NMIN
NO
NODE
NOHDR
NONE
NORMALIZE
NORMALIZED
NOT
NOTIT
NOTITLE
NULL
NULL-HANDLE
NUMBER
NUMERIC

- O -

OBJECT
OBTAIN
OCCURRENCES
OF
OFF
OFFSET
OLD
ON
ONCE
ONLY
OPEN
OPTIMIZE
OPTIONAL
OPTIONS
OR
ORDER
OUTER
OUTPUT

- P -

PACKAGESET

PAGE
PARAMETER
PARAMETERS
PARENT
PARSE
PASS
PASSW
PASSWORD
PATH
PATTERN
PA1
PA2
PA3
PC
PD
PEN
PERFORM
PF*n* (*n* = 1 to 9)
PF*nn*(*nn* = 10 to 99)
PGDN
PGUP
PGM
PHYSICAL
PM
POLICY
POS
POSITION
PREFIX
PRINT
PRINTER
PROCESS
PROCESSING
PROFILE
PROGRAM
PROPERTY
PROTOTYPE
PRTY
PS
PT
PW

- Q -

QUARTER
QUERYNO

- R -

RD
READ
READONLY
REC
RECORD
RECORDS
RECURSIVELY
REDEFINE
REDUCE
REFERENCED
REFERENCING
REINPUT
REJECT
REL
RELATION
RELATIONSHIP
RELEASE
REMAINDER
REPEAT
REPLACE
REPORT
REPORTER
REPOSITION
REQUEST
REQUIRED
RESET
RESETTING
RESIZE
RESPONSE
RESTORE
RESULT
RET
RETAIN
RETAINED
RETRY
RETURN
RETURNS
REVERSED
RG
RIGHT
ROLLBACK
ROUNDED
ROUTINE

ROW
ROWS
RR
RS
RULEVAR
RUN

- S -

SA
SAME
SCAN
SCREEN
SCROLL
SECOND
SELECT
SELECTION
SEND
SENSITIVE
SEPARATE
SEQUENCE
SERVER
SET
SETS
SETTIME
SF
SG
SGN
SHORT
SHOW
SIN
SINGLE
SIZE
SKIP
SL
SM
SOME
SORT
SORTED
SORTKEY
SOUND
SPACE
SPECIFIED
SQL
SQLID

SQRT
STACK
START
STARTING
STATEMENT
STATIC
STATUS
STEP
STOP
STORE
SUBPROGRAM
SUBPROGRAMS
SUBROUTINE
SUBSTR
SUBSTRING
SUBTRACT
SUM
SUPPRESS
SUPPRESSED
SUSPEND
SYMBOL
SYNC
SYSTEM

- T -

TAN
TC
TERMINATE
TEXT
TEXTAREA
TEXTVARIABLE
THAN
THEM
THEN
THRU
TIME
TIMESTAMP
TIMEZONE
TITLE
TO
TOP
TOTAL
TP
TR

TRAILER
TRANSACTION
TRANSFER
TRANSLATE
TREQ
TRUE
TS
TYPE
TYPES

- U -

UC
UNDERLINED
UNION
UNIQUE
UNKNOWN
UNTIL
UPDATE
UPLOAD
UPPER
UR
USED
USER
USING

- V -

VAL
VALUE
VALUES
VARGRAPHIC
VARIABLE
VARIABLES
VERT
VERTICALLY
VIA
VIEW

- W -

WH
WHEN
WHERE
WHILE
WINDOW

WITH
WORK
WRITE
WITH_CTE

- X -

XML

- Y -

YEAR

- Z -

ZD
ZP

Performing a Check for Natural Reserved Keywords

There is a subset of Natural keywords which, when used as names for variables, would be ambiguous. These are in particular keywords which identify Natural statements (`ADD`, `FIND`, etc.) or system functions (`ABS`, `SUM`, etc.). If you use such a keyword as the name of a variable, you cannot use this variable in the context of optional operands (with `CALLNAT`, `WRITE`, etc.).

Example:

```
DEFINE DATA LOCAL
1 ADD (A10)
END-DEFINE
CALLNAT 'MYSUB' ADD 4     /* ADD is regarded as ADD statement
END
```

To check variable names in a Natural object against such Natural reserved keywords, you can use the Natural profile parameter `KCHECK` or the `KCHECK` option of the `COMPOPT` system command.

The following table contains a list of Natural reserved keywords that are checked by `KC` or `KCHECK`.

A - D	E - F	G - P	R - S	T - W
ABS	EJECT	GET	READ	TAN
ACCEPT	ELSE	HISTOGRAM	REDEFINE	TERMINATE
ADD	END	IF	REDUCE	TOP
ALL	END-ALL	IGNORE	REINPUT	TOTAL
ANY	END-BEFORE	IMPORT	REJECT	TRANSFER
ASSIGN	END-BREAK	INCCONT	RELEASE	TRUE
AT	END-BROWSE	INCDIC	REPEAT	UNTIL
ATN	END-DECIDE	INCDIR	REQUEST	UPDATE
AVER	END-ENDDATA	INCLUDE	RESET	UPLOAD
BACKOUT	END-ENDFILE	INCMAC	RESIZE	VAL
BEFORE	END-ENDPAGE	INPUT	RESTORE	VALUE
BREAK	END-ERROR	INSERT	RET	VALUES
BROWSE	END-FILE	INT	RETRY	WASTE
CALL	END-FIND	INVESTIGATE	RETURN	WHEN
CALLDBPROC	END-FOR	LIMIT	ROLLBACK	WHILE
CALLNAT	END-FUNCTION	LOG	ROUNDED	WITH_CTE
CLOSE	END-HISTOGRAM	LOOP	RULEVAR	WRITE
COMMIT	ENDHOC	MAP	RUN	
COMPOSE	END-IF	MAX	SELECT	
COMPRESS	END-LOOP	MIN	SEND	
COMPUTE	END-NOREC	MOVE	SEPARATE	
COPY	END-PARSE	MULTIPLY	SET	
COS	END-PROCESS	NAVER	SETTIME	
COUNT	END-READ	NCOUNT	SGN	
CREATE	END-REPEAT	NEWPAGE	SHOW	
DECIDE	END-RESULT	NMIN	SIN	
DEFINE	END-SELECT	NONE	SKIP	
DELETE	END-SORT	NULL-HANDLE	SORT	
DISPLAY	END-START	OBTAIN	SORTKEY	
DIVIDE	END-SUBROUTINE	OLD	SQRT	
DLOGOFF	END-TOPPAGE	ON	STACK	
DLOGON	END-WORK	OPEN	START	
DNATIVE	ENTIRE	OPTIONS	STOP	
DO	ESCAPE	PARSE	STORE	
DOEND	EXAMINE	PASSW	SUBSTR	
DOWNLOAD	EXP	PERFORM	SUBSTRING	
	EXPAND	POS	SUBTRACT	
	EXPORT	PRINT	SUM	
	FALSE	PROCESS	SUSPEND	
	FETCH			
	FIND			
	FOR			
	FORMAT			
	FRAC			

By default, no keyword check is performed.

76 Referenced Example Programs

This chapter contains some additional example programs that are referenced in the *Programming Guide*.

Notes:

1. All example programs shown in the *Programming Guide* are also provided as source objects in the Natural library `SYSEXPG`. The example programs use data from the files `EMPLOYEES` and `VEHICLES`, which are supplied by Software AG for demonstration purposes. The Natural library `SYSEXPG` also includes example programs for Natural **functions**.
2. Further example programs of using Natural statements are provided in the Natural library `SYSEXSYN` and are documented in the section *Referenced Example Programs* in the *Statements* documentation.
3. Please ask your Natural administrator about the availability of the libraries `SYSEXPG` and `SYSEXSYN` at your site.
4. To use any Natural example program to access an Adabas database, the Adabas nucleus parameter `OPTIONS` must be set to `TRUNCATION`.

READ Statement

The following example is referenced in the section ***Statements for Database Access***.

READX03 - READ statement (with LOGICAL clause)

```
** Example 'READX03': READ (with LOGICAL clause)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 PERSONNEL-ID
  2 JOB-TITLE
END-DEFINE
*
LIMIT 8
READ EMPLOY-VIEW LOGICAL BY PERSONNEL-ID
  DISPLAY NOTITLE  *ISN      NAME
                  'PERS-NO'  PERSONNEL-ID
                  'POSITION' JOB-TITLE
END-READ
END
```

Output of Program READX03:

```
   ISN             NAME         PERS-NO         POSITION
----------- -------------------- -------- -------------------------

        204 SCHINDLER            11100102 PROGRAMMIERER
        205 SCHIRM               11100105 SYSTEMPROGRAMMIERER
        206 SCHMITT              11100106 OPERATOR
        207 SCHMIDT              11100107 SEKRETAERIN
        208 SCHNEIDER            11100108 SACHBEARBEITER
        209 SCHNEIDER            11100109 SEKRETAERIN
        210 BUNGERT              11100110 SYSTEMPROGRAMMIERER
        211 THIELE               11100111 SEKRETAERIN
```

FIND Statement

The following examples are referenced in the section ***Statements for Database Access***.

FINDX07 - FIND statement (with several clauses)

```
** Example 'FINDX07': FIND  (with several clauses)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 SALARY    (1)
  2 CURR-CODE (1)
END-DEFINE
*
FIND EMPLOY-VIEW WITH  PHONETIC-NAME = 'JONES' OR = 'BECKR'
                 AND   CITY          = 'BOSTON' THRU 'NEW YORK'
                 BUT NOT               'CHAPEL HILL'
                 SORTED BY NAME
                 WHERE SALARY (1)    < 28000
  DISPLAY NOTITLE NAME FIRST-NAME CITY SALARY (1)
END-FIND
END
```

Output of Program FINDX07:

```
       NAME                 FIRST-NAME                CITY          ANNUAL
                                                                    SALARY
-------------------- -------------------- -------------------- ----------

BAKER                PAULINE              DERBY                      4450
JONES                MARTHA               KALAMAZOO                 21000
JONES                KEVIN                DERBY                      7000
```

FINDX08 - FIND statement (with LIMIT)

```
** Example 'FINDX08': FIND (with LIMIT)
**                    Demonstrates FIND statement with LIMIT option to
**                    terminate program with an error message if the
**                    number of records selected exceeds a specified
**                    limit (no output).
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
END-DEFINE
*
FIND EMPLOY-VIEW WITH LIMIT (5) JOB-TITLE = 'SALES PERSON'
  DISPLAY NAME JOB-TITLE
END-FIND
END
```

Runtime Error Caused by Program FINDX08:

```
NAT1005 More records found than specified in search limit.
```

FINDX09 - FIND statement (using *NUMBER, *COUNTER, *ISN)

```
** Example 'FINDX09': FIND (using *NUMBER, *COUNTER, *ISN)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 DEPT
  2 NAME
END-DEFINE
*
FIND EMPLOY-VIEW WITH CITY  = 'BOSTON'
                 WHERE DEPT = 'TECH00' THRU 'TECH10'
  DISPLAY NOTITLE
         'COUNTER' *COUNTER NAME DEPT 'ISN' *ISN
  AT START OF DATA
    WRITE '(TOTAL NUMBER IN BOSTON:' *NUMBER ')' /
  END-START
```

```
END-FIND
END
```

Output of Program FINDX09:

```
  COUNTER           NAME         DEPARTMENT     ISN
                                    CODE
----------- -------------------- ---------- -----------

(TOTAL NUMBER IN BOSTON:           7 )

          1 STANWOOD             TECH10             782
          2 PERREAULT            TECH10             842
```

FINDX10 - FIND statement (combined with READ)

```
** Example 'FINDX10': FIND (combined with READ)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
1 VEHIC-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 15
*
EMP. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  VEH. FIND VEHIC-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (EMP.)
    IF NO RECORDS FOUND
      MOVE '*** NO CAR ***' TO MAKE
    END-NOREC
    DISPLAY NOTITLE
            NAME (EMP.) (IS=ON)
            FIRST-NAME (EMP.) (IS=ON)
            MAKE (VEH.)
  END-FIND
END-READ
END
```

Output of Program FINDX10:

```
       NAME             FIRST-NAME             MAKE
-------------------- -------------------- --------------------

JONES                VIRGINIA             CHRYSLER
                     MARSHA               CHRYSLER
                                          CHRYSLER
                     ROBERT               GENERAL MOTORS
                     LILLY                FORD
                                          MG
                     EDWARD               GENERAL MOTORS
                     MARTHA               GENERAL MOTORS
                     LAUREL               GENERAL MOTORS
                     KEVIN                DATSUN
                     GREGORY              FORD
JOPER                MANFRED              *** NO CAR ***
JOUSSELIN            DANIEL               RENAULT
JUBE                 GABRIEL              *** NO CAR ***
JUNG                 ERNST                *** NO CAR ***
JUNKIN               JEREMY               *** NO CAR ***
KAISER               REINER               *** NO CAR ***
```

FINDX11 - FIND NUMBER statement (with *NUMBER)

```
** Example 'FINDX11': FIND NUMBER (with *NUMBER)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 CITY
  2 JOB-TITLE
  2 SALARY        (1)
*
1 #PERCENT        (N.2)
1 REDEFINE #PERCENT
  2 #WHOLE-NBR    (N2)
1 #ALL-BOST       (N3.2)
1 #SECR-ONLY      (N3.2)
1 #BOST-NBR       (N3)
1 #SECR-NBR       (N3)
END-DEFINE
*
F1. FIND NUMBER EMPLOY-VIEW WITH CITY = 'BOSTON'
F2. FIND NUMBER EMPLOY-VIEW WITH CITY = 'BOSTON'
                            AND JOB-TITLE = 'SECRETARY'
*
MOVE *NUMBER(F1.) TO #ALL-BOST  #BOST-NBR
MOVE *NUMBER(F2.) TO #SECR-ONLY #SECR-NBR
DIVIDE #ALL-BOST INTO #SECR-ONLY GIVING #PERCENT
*
```

```
WRITE TITLE LEFT JUSTIFIED UNDERLINED
  'There are' #BOST-NBR 'employees in the Boston offices.' /
  #SECR-NBR '(=' #WHOLE-NBR (EM=99%')') 'are secretaries.'
*
SKIP 1
FIND EMPLOY-VIEW WITH CITY      = 'BOSTON'
                 AND  JOB-TITLE = 'SECRETARY'
  DISPLAY NAME FIRST-NAME JOB-TITLE SALARY (1)
END-FIND
END
```

Output of Program FINDX11:

```
There are    7 employees in the Boston offices.
   3 (= 42%) are secretaries.
-------------------------------------------------------------------------------

        NAME                 FIRST-NAME               CURRENT             ANNUAL
                                                     POSITION             SALARY
-------------------- -------------------- ------------------------- ----------

SHAW                 LESLIE               SECRETARY                      18000
CREMER               WALT                 SECRETARY                      20000
COHEN                JOHN                 SECRETARY                      16000
```

Nested READ and FIND Statements

The following examples are referenced in the section *Database Processing Loops*.

READX04 - READ statement (in combination with FIND and the system variables *NUMBER and *COUNTER)

```
** Example 'READX04': READ (in combination with FIND and the system
**                     variables *NUMBER and *COUNTER)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
1 VEHIC-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 10
```

```
RD. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  FD. FIND VEHIC-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      ENTER
    END-NOREC
    /*
    DISPLAY NOTITLE
            *COUNTER (RD.)(NL=8) NAME         (AL=15) FIRST-NAME (AL=10)
            *NUMBER  (FD.)(NL=8) *COUNTER (FD.)(NL=8) MAKE
  END-FIND
END-READ
END
```

Output of Program READX04:

```
  CNT          NAME       FIRST-NAME   NMBR      CNT            MAKE
--------- --------------- ---------- --------- --------- --------------------

        1 JONES           VIRGINIA           1         1 CHRYSLER
        2 JONES           MARSHA             2         1 CHRYSLER
        2 JONES           MARSHA             2         2 CHRYSLER
        3 JONES           ROBERT             1         1 GENERAL MOTORS
        4 JONES           LILLY              2         1 FORD
        4 JONES           LILLY              2         2 MG
        5 JONES           EDWARD             1         1 GENERAL MOTORS
        6 JONES           MARTHA             1         1 GENERAL MOTORS
        7 JONES           LAUREL             1         1 GENERAL MOTORS
        8 JONES           KEVIN              1         1 DATSUN
        9 JONES           GREGORY            1         1 FORD
       10 JOPER           MANFRED            0         0
```

LIMITX01 - LIMIT statement (for READ, FIND loop processing)

```
** Example 'LIMITX01': LIMIT  (for READ, FIND loop processing)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
1 VEH-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 4
*
READ EMPLOY-VIEW BY NAME STARTING FROM 'A'
  FIND VEH-VIEW WITH PERSONNEL-ID = EMPLOY-VIEW.PERSONNEL-ID
```

```
    IF NO RECORDS FOUND
      MOVE 'NO CAR' TO MAKE
    END-NOREC
    DISPLAY PERSONNEL-ID NAME FIRST-NAME MAKE
  END-FIND
END-READ
END
```

Output of Program LIMITX01:

```
Page      1                                                  04-12-13  14:01:57

PERSONNEL-ID         NAME                FIRST-NAME              MAKE
------------ -------------------- -------------------- --------------------

             ABELLAN              KEPA                 NO CAR
30000231     ACHIESON             ROBERT               FORD
             ADAM                 SIMONE               NO CAR
20008800     ADKINSON             JEFF                 GENERAL MOTORS
```

ACCEPT and REJECT Statements

The following examples are referenced in the section *Selecting Records Using ACCEPT/REJECT*.

ACCEPX04 - ACCEPT IF ... LESS THAN ... statement

```
** Example 'ACCEPX04': ACCEPT IF ... LES THAN ...
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 SALARY (1)
END-DEFINE
*
LIMIT 20
READ EMPLOY-VIEW BY PERSONNEL-ID = '20017000'
  ACCEPT IF SALARY (1) LESS THAN 38000
  DISPLAY NOTITLE PERSONNEL-ID NAME JOB-TITLE SALARY (1)
END-READ
END
```

Output of Program ACCEPX04:

```
PERSONNEL         NAME                  CURRENT           ANNUAL
   ID                                  POSITION           SALARY
--------- -------------------- ------------------------- ----------

20017000  CREMER               ANALYST                        34000
20017100  MARKUSH              TRAINEE                        22000
20017400  NEEDHAM              PROGRAMMER                     32500
20017500  JACKSON              PROGRAMMER                     33000
20017600  PIETSCH              SECRETARY                      22000
20017700  PAUL                 SECRETARY                      23000
20018000  FARRIS               PROGRAMMER                     30500
20018100  EVANS                PROGRAMMER                     31000
20018200  HERZOG               PROGRAMMER                     31500
20018300  LORIE                SALES PERSON                   28000
20018400  HALL                 SALES PERSON                   30000
20018500  JACKSON              MANAGER                        36000
20018800  SMITH                SECRETARY                      24000
20018900  LOWRY                SECRETARY                      25000
```

ACCEPX05 - ACCEPT IF ... AND ... statement

```
** Example 'ACCEPX05': ACCEPT IF ... AND ...
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 JOB-TITLE
  2 SALARY (1:2)
END-DEFINE
*
LIMIT 6
READ EMPLOY-VIEW PHYSICAL WHERE SALARY(2) > 0
  ACCEPT IF  SALARY(1) > 10000
         AND SALARY(1) < 50000
  DISPLAY (AL=15) 'SALARY I' SALARY (1) 'SALARY II'  SALARY (2)
                   NAME JOB-TITLE  CITY
END-READ
END
```

Output of Program ACCEPX05:

```
Page      1                                                04-12-13  14:05:28

 SALARY I  SALARY II       NAME           CURRENT          CITY
                                         POSITION
---------- ---------- -------------- -------------- ---------------

     48000      46000 SPENGLER       SACHBEARBEITER  DARMSTADT
     45000      40000 SPECK          SACHBEARBEITER  DARMSTADT
     48000      46000 SCHINDLER      PROGRAMMIERER   HEPPENHEIM
     36000      32000 SCHMIDT        SEKRETAERIN     HEPPENHEIM
```

ACCEPX06 - REJECT IF ... OR ... statement

```
** Example 'ACCEPX06': REJECT IF ... OR ...
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 SALARY (1)
  2 JOB-TITLE
  2 CITY
  2 NAME
END-DEFINE
*
LIMIT 20
READ EMPLOY-VIEW LOGICAL BY PERSONNEL-ID = '20017000'
  REJECT IF SALARY (1) < 20000
         OR SALARY (1) > 26000
  DISPLAY NOTITLE SALARY (1) NAME JOB-TITLE CITY
END-READ
END
```

Output of Program ACCEPX06:

```
  ANNUAL           NAME                 CURRENT                 CITY
  SALARY                               POSITION
---------- -------------------- ------------------------- --------------------

     22000 MARKUSH              TRAINEE                   LOS ANGELES
     22000 PIETSCH              SECRETARY                 VISTA
     23000 PAUL                 SECRETARY                 NORFOLK
     24000 SMITH                SECRETARY                 SILVER SPRING
     25000 LOWRY                SECRETARY                 LEXINGTON
```

AT START OF DATA and AT END OF DATA Statements

The following examples are referenced in the section *AT START/END OF DATA Statements*.

ATENDX01 - AT END OF DATA statement

```
** Example 'ATENDX01': AT END OF DATA
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
END-DEFINE
*
READ (6) EMPLOY-VIEW BY PERSONNEL-ID FROM '20017000'
  DISPLAY NOTITLE NAME JOB-TITLE
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD(NAME)
  END-ENDDATA
END-READ
END
```

Output of Program ATENDX01:

```
        NAME                      CURRENT
                                 POSITION
-------------------- -------------------------

CREMER               ANALYST
MARKUSH              TRAINEE
GEE                  MANAGER
KUNEY                DBA
NEEDHAM              PROGRAMMER
JACKSON              PROGRAMMER

LAST PERSON SELECTED: JACKSON
```

ATSTAX02 - AT START OF DATA statement

```
** Example 'ATSTAX02': AT START OF DATA
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 SALARY    (1)
  2 CURR-CODE (1)
  2 BONUS     (1,1)
END-DEFINE
*
LIMIT 3
FIND EMPLOY-VIEW WITH CITY = 'MADRID'
  DISPLAY NAME FIRST-NAME SALARY(1) BONUS(1,1) CURR-CODE (1)
  /*
  AT START OF DATA
    WRITE NOTITLE *DAT4E /
  END-START
END-FIND
END
```

Output of Program ATSTAX02:

```
        NAME              FIRST-NAME        ANNUAL     BONUS    CURRENCY
                                            SALARY                CODE
-------------------- -------------------- ---------- ---------- --------

13/12/2004

DE JUAN              JAVIER                  1988000          0 PTA
DE LA MADRID         ANSELMO                 3120000          0 PTA
PINERO               PAULA                   1756000          0 PTA
```

WRITEX09 - WRITE statement (in combination with AT END OF DATA)

```
** Example 'WRITEX09': WRITE (in combination with AT END OF DATA )
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 BIRTH
  2 JOB-TITLE
  2 DEPT
END-DEFINE
```

```
*
READ (3) EMPLOY-VIEW BY CITY
  DISPLAY NOTITLE NAME BIRTH (EM=YYYY-MM-DD) JOB-TITLE
  WRITE 38T 'DEPT CODE:' DEPT
  /*
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD(NAME)
  END-ENDDATA
  SKIP 1
END-READ
END
```

Output of Program WRITEX09:

```
        NAME              DATE              CURRENT
                           OF              POSITION
                         BIRTH
-------------------- ---------- -------------------------

SENKO                1971-09-11 PROGRAMMER
                                     DEPT CODE: TECH10

GODEFROY             1949-01-09 COMPTABLE
                                     DEPT CODE: COMP02

CANALE               1942-01-01 CONSULTANT
                                     DEPT CODE: TECH03


LAST PERSON SELECTED: CANALE
```

DISPLAY and WRITE Statements

The following examples are referenced in the section *Statements DISPLAY and WRITE*.

DISPLX13 - DISPLAY statement (compare with WRITEX08 using WRITE)

```
** Example 'DISPLX13': DISPLAY (compare with WRITEX08 using WRITE)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 SALARY (2)
  2 BONUS  (1,1)
```

```
  2 CITY
END-DEFINE
*
LIMIT 2
READ EMPLOY-VIEW WITH CITY = 'CHAPEL HILL' WHERE BONUS(1,1) NE 0
  /*
  DISPLAY 'PERS/ID' PERSONNEL-ID  NAME / FIRST-NAME
          '**' '=' SALARY(1:2) 'BONUS' BONUS(1,1) CITY (AL=15)
  /*
  SKIP 1
END-READ
END
```

Output of Program DISPLX13:

```
Page      1                                                  04-12-13  14:11:28

  PERS           NAME            ANNUAL       BONUS         CITY
   ID         FIRST-NAME         SALARY
-------- -------------------- ------------- ---------- ---------------

20027000 CUMMINGS             **      41000       1500 CHAPEL HILL
         PUALA                        38900

20000200 WOOLSEY              **      26000       3000 CHAPEL HILL
         LOUISE                       24700
```

WRITEX08 - WRITE statement (compare with DISPLX13 using DISPLAY)

```
** Example 'WRITEX08': WRITE  (compare with DISPLX13 using DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 SALARY (2)
  2 BONUS  (1,1)
  2 CITY
END-DEFINE
*
LIMIT 2
READ EMPLOY-VIEW WITH CITY = 'CHAPEL HILL' WHERE BONUS(1,1) NE 0
  /*
  WRITE 'PERS/ID' PERSONNEL-ID  NAME / FIRST-NAME
        '**' '=' SALARY(1:2) 'BONUS' BONUS(1,1) CITY (AL=15)
  /*
  SKIP 1
```

```
END-READ
END
```

Output of Program WRITEX08:

```
Page      1                                                   04-12-13  14:12:43

PERS/ID 20027000 CUMMINGS
PUALA                ** ANNUAL SALARY:      41000      38900 BONUS       1500
CHAPEL HILL

PERS/ID 20000200 WOOLSEY
LOUISE               ** ANNUAL SALARY:      26000      24700 BONUS       3000
CHAPEL HILL
```

DISPLX14 - DISPLAY statement (with AL, SF and *n*X)

```
** Example 'DISPLX14': DISPLAY  (with AL, SF and nX)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 ADDRESS-LINE (1)
  2 TELEPHONE
    3 AREA-CODE
    3 PHONE
  2 CITY
END-DEFINE
*
READ (3) EMPLOY-VIEW BY NAME STARTING FROM 'W'
  DISPLAY (AL=15 SF=5) NAME CITY / ADDRESS-LINE(1) 2X TELEPHONE
  SKIP 1
END-READ
END
```

Output of Program DISPLX14:

```
Page      1                                                   04-12-13  14:14:00

     NAME                CITY                     TELEPHONE
                        ADDRESS
                                          AREA              TELEPHONE
                                          CODE
---------------     ---------------  ---------------     ---------------

WABER               HEIDELBERG       06221               456452
                    ERBACHERSTR. 78
```

```
WADSWORTH           DERBY            0332                515365
                    56 PINECROFT CO

WAGENBACH           FRANKFURT        069                 983218
                    BECKERSTR. 4
```

WRITEX09 - WRITE statement (in combination with AT END OF DATA)

```
** Example 'WRITEX09': WRITE (in combination with AT END OF DATA )
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 BIRTH
  2 JOB-TITLE
  2 DEPT
END-DEFINE
*
READ (3) EMPLOY-VIEW BY CITY
  DISPLAY NOTITLE NAME BIRTH (EM=YYYY-MM-DD) JOB-TITLE
  WRITE 38T 'DEPT CODE:' DEPT
  /*
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD(NAME)
  END-ENDDATA
  SKIP 1
END-READ
END
```

Output of Program WRITEX09:

```
        NAME            DATE             CURRENT
                         OF             POSITION
                       BIRTH
-------------------- ---------- -------------------------

SENKO                1971-09-11 PROGRAMMER
                                     DEPT CODE: TECH10

GODEFROY             1949-01-09 COMPTABLE
                                     DEPT CODE: COMP02

CANALE               1942-01-01 CONSULTANT
                                     DEPT CODE: TECH03


LAST PERSON SELECTED: CANALE
```

DISPLAY Statement

The following example is referenced in the section *Page Titles, Page Breaks, Blank Lines*.

DISPLX21 DISPLAY statement (with slash '/' and compare with WRITE)

```
** Example 'DISPLX21': DISPLAY (usage of slash '/' in DISPLAY and WRITE)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
  2 ADDRESS-LINE (1)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED
       *TIME
   5X  'PEOPLE LIVING IN SALT LAKE CITY'
   21X 'PAGE:' *PAGE-NUMBER /
   15X 'AS OF' *DAT4E //
*
WRITE TRAILER UNDERLINED 'REGISTER OF' / 'SALT LAKE CITY'
*
READ (2) EMPLOY-VIEW WITH CITY = 'SALT LAKE CITY'
  DISPLAY  NAME /
           FIRST-NAME
           'HOME/CITY' CITY
           'STREET/OR BOX NO.' ADDRESS-LINE (1)
  SKIP 1
END-READ
END
```

Output of Program DISPLX21:

```
14:15:50.1     PEOPLE LIVING IN SALT LAKE CITY                     PAGE:      1
               AS OF 13/12/2004


-------------------------------------------------------------------------------
        NAME                 HOME                STREET
     FIRST-NAME              CITY              OR BOX NO.
-------------------- -------------------- --------------------


ANDERSON             SALT LAKE CITY       3701 S. GEORGE MASON
JENNY
```

```
SAMUELSON           SALT LAKE CITY      7610 W. 86TH STREET
MARTIN

                                  REGISTER OF
                                SALT LAKE CITY
-------------------------------------------------------------------------------
```

Column Headers

The following example is referenced in the section *Column Headers*.

DISPLX15 - DISPLAY statement (with FC, UC)

```
** Example 'DISPLX15': DISPLAY  (with FC, UC)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 ADDRESS-LINE (1)
  2 CITY
  2 TELEPHONE
    3 AREA-CODE
    3 PHONE
END-DEFINE
*
FORMAT AL=12 GC== UC=%
*
READ (3) EMPLOY-VIEW BY NAME STARTING FROM 'R'
  DISPLAY NOTITLE (FC=*)
          NAME FIRST-NAME CITY  (FC=- UC=-) /
          ADDRESS-LINE(1) TELEPHONE
  SKIP 1
END-READ
END
```

Output of Program DISPLX15:

```
****NAME**** *FIRST-NAME* ----CITY---- ========TELEPHONE========
                          **ADDRESS***
                                       ****AREA**** *TELEPHONE**
                                       ****CODE****
%%%%%%%%%%%% %%%%%%%%%%%% ------------ %%%%%%%%%%%% %%%%%%%%%%%%

RACKMANN     MARIAN       FRANKFURT    069          375849
                          FINKENSTR. 1
```

```
RAMAMOORTHY  TY           SEPULVEDA    209         175-1885
                          12018 BROOKS

RAMAMOORTHY  TIMMIE       SEATTLE      206         151-4673
                          921-178TH PL
```

DISPLX16 - DISPLAY statement (with '/', '*text*', '*text/text*')

```
** Example 'DISPLX16': DISPLAY  (with '/', 'text', 'text/text')
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 ADDRESS-LINE (1)
  2 CITY
  2 TELEPHONE
    3 AREA-CODE
    3 PHONE
END-DEFINE
*
READ (5) EMPLOY-VIEW BY NAME STARTING FROM 'E'
  DISPLAY NOTITLE
    '/'          NAME       (AL=12) /* suppressed header
    'FIRST/NAME' FIRST-NAME (AL=10) /* two-line user-defined header
    'ADDRESS'    CITY /             /* user-defined header
    ' '          ADDRESS-LINE(1)    /* 'blank' header
                 TELEPHONE (HC=L)   /* default header
  SKIP 1
END-READ
END
```

Output of Program DISPLX16:

```
               FIRST          ADDRESS        TELEPHONE
                NAME
                                             AREA   TELEPHONE
                                             CODE
             ---------- -------------------- ------ ---------------

EAVES        TREVOR     DERBY                0332   657623
                        17 HARTON ROAD

ECKERT       KARL       OBERRAMSTADT         06154  99722
                        FORSTWEG 22

ECKHARDT     RICHARD    DARMSTADT
                        BRESLAUERPL. 4
```

```
EDMUNDSON    LES        TULSA                918    945-4916
                        2415 ALSOP CT.

EGGERT       HERMANN    STUTTGART            0711   981237
                        RABENGASSE 8
```

Field-Output-Relevant Parameters

The following examples are referenced in the section *Parameters to Influence the Output of Fields*.

They are provided to demonstrate the use of the parameters `LC`, `IC`, `TC`, `AL`, `NL`, `IS`, `ZP` and `ES`, and the `SUSPEND IDENTICAL SUPPRESS` statement:

DISPLX17 - DISPLAY statement (with NL, AL, IC, LC, TC)

```
** Example 'DISPLX17': DISPLAY (with NL, AL, IC, LC, TC)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 SALARY (1)
  2 BONUS  (1,1)
END-DEFINE
*
READ (3) EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  DISPLAY NOTITLE (IS=ON NL=15)
                    NAME
    '-' '='         FIRST-NAME (AL=12)
    'ANNUAL SALARY' SALARY(1)  (LC=USD   TC=.00)   /
    '+ BONUSES'     BONUS(1,1) (IC='+  ' TC=.00)
  SKIP 1
END-READ
END
```

Output of Program DISPLX17:

```
        NAME           FIRST-NAME       ANNUAL SALARY
                                          + BONUSES
-------------------- -------------- ----------------------

JONES                - VIRGINIA     USD           46000.00
                                                +  9000.00

                     - MARSHA       USD           50000.00
```

```
                                               +  0.00

                     - ROBERT       USD           31000.00
                                               +  0.00
```

DISPLX18 - DISPLAY statement (using default settings for SF, AL, UC, LC, IC, TC and compare with DISPLX19)

```
** Example 'DISPLX18': DISPLAY  (using default settings for SF, AL, UC,
**                     LC, IC, TC and compare with DISPLX19)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 SALARY    (1)
  2 BONUS     (1,1)
END-DEFINE
*
FIND (6) EMPLOY-VIEW WITH CITY = 'CHAPEL HILL'
  DISPLAY NAME FIRST-NAME SALARY(1) BONUS(1,1)
END-FIND
END
```

Output of Program DISPLX18:

```
Page      1                                              04-12-13  14:20:48

        NAME                FIRST-NAME        ANNUAL     BONUS
                                              SALARY
-------------------- -------------------- ---------- ----------

KESSLER              CLARE                     41000          0
ADKINSON             DAVID                     24000          0
GEE                  TOMMIE                    39500          0
HERZOG               JOHN                      31500          0
QUILLION             TIMOTHY                   30500          0
CUMMINGS             PUALA                     41000       1500
```

DISPLX19 - DISPLAY statement (with SF, AL, LC, IC, TC and compare with DISPLX18)

```
** Example 'DISPLX19': DISPLAY  (with SF, AL, LC, IC, TC and compare
**                     with DISPLX19)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 SALARY (1)
  2 BONUS  (1,1)
END-DEFINE
*
FORMAT SF=3 AL=15 UC==
*
FIND (6) EMPLOY-VIEW WITH CITY = 'CHAPEL HILL'
  DISPLAY (NL=10)
    NAME
    FIRST-NAME  (LC='- ' UC=-)
    SALARY (1)  (LC=USD)
    BONUS (1,1) (IC='*** ' TC=' ***')
END-FIND
END
```

Output of Program DISPLX19:

```
Page      1                                                04-12-13  14:21:57

     NAME            FIRST-NAME          ANNUAL             BONUS
                                         SALARY
===============   -----------------   ==============   ===================

KESSLER           - CLARE             USD      41000             *** 0 ***
ADKINSON          - DAVID             USD      24000             *** 0 ***
GEE               - TOMMIE            USD      39500             *** 0 ***
HERZOG            - JOHN              USD      31500             *** 0 ***
QUILLION          - TIMOTHY           USD      30500             *** 0 ***
CUMMINGS          - PUALA             USD      41000          *** 1500 ***
```

SUSPEX01 - SUSPEND IDENTICAL SUPPRESS statement (in conjunction with parameters IS, ES, ZP in DISPLAY)

```
** Example 'SUSPEX01': SUSPEND IDENTICAL SUPPRESS (in conjunction with
**                     parameters IS, ES, ZP in DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 CITY
1 VEH-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 15
RD. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  SUSPEND IDENTICAL SUPPRESS
  FD. FIND VEH-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      MOVE '******' TO MAKE
    END-NOREC
    DISPLAY NOTITLE (ES=OFF IS=ON ZP=ON AL=15)
            NAME       (RD.)
            FIRST-NAME (RD.)
            MAKE       (FD.) (IS=OFF)
  END-FIND
END-READ
END
```

Output of Program SUSPEX01:

```
     NAME         FIRST-NAME         MAKE
--------------- --------------- ---------------

JONES           VIRGINIA        CHRYSLER
JONES           MARSHA          CHRYSLER
                                CHRYSLER
JONES           ROBERT          GENERAL MOTORS
JONES           LILLY           FORD
                                MG
JONES           EDWARD          GENERAL MOTORS
JONES           MARTHA          GENERAL MOTORS
JONES           LAUREL          GENERAL MOTORS
JONES           KEVIN           DATSUN
JONES           GREGORY         FORD
JOPER           MANFRED         ******
```

```
JOUSSELIN       DANIEL          RENAULT
JUBE            GABRIEL         ******
JUNG            ERNST           ******
JUNKIN          JEREMY          ******
KAISER          REINER          ******
```

SUSPEX02 - SUSPEND IDENTICAL SUPPRESS statement (in conjunction with parameters IS, ES, ZP in DISPLAY) Identical to SUSPEX01, but with IS=OFF.

```
** Example 'SUSPEX02': SUSPEND IDENTICAL SUPPRESS (in conjunction with
**                     parameters IS, ES, ZP in DISPLAY)
**                     Identical to SUSPEX01, but with IS=OFF.
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 CITY
1 VEH-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 15
RD. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  SUSPEND IDENTICAL SUPPRESS
  FD. FIND VEH-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      MOVE '******' TO MAKE
    END-NOREC
    DISPLAY NOTITLE (ES=OFF IS=OFF ZP=ON AL=15)
            NAME       (RD.)
            FIRST-NAME (RD.)
            MAKE       (FD.) (IS=OFF)
  END-FIND
END-READ
END
```

Output of Program SUSPEX02:

```
     NAME         FIRST-NAME         MAKE
--------------- --------------- ---------------

JONES           VIRGINIA        CHRYSLER
JONES           MARSHA          CHRYSLER
JONES           MARSHA          CHRYSLER
JONES           ROBERT          GENERAL MOTORS
JONES           LILLY           FORD
```

```
JONES           LILLY           MG
JONES           EDWARD          GENERAL MOTORS
JONES           MARTHA          GENERAL MOTORS
JONES           LAUREL          GENERAL MOTORS
JONES           KEVIN           DATSUN
JONES           GREGORY         FORD
JOPER           MANFRED         ******
JOUSSELIN       DANIEL          RENAULT
JUBE            GABRIEL         ******
JUNG            ERNST           ******
JUNKIN          JEREMY          ******
KAISER          REINER          ******
```

COMPRX03 - COMPRESS statement

```
** Example 'COMPRX03': COMPRESS (using parameters LC and TC)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 SALARY     (1)
  2 CURR-CODE  (1)
  2 LEAVE-DUE
  2 NAME
  2 FIRST-NAME
  2 JOB-TITLE
*
1 #SALARY      (N9)
1 #FULL-SALARY (A25)
1 #VACATION    (A11)
END-DEFINE
*
READ (3) EMPLOY-VIEW WITH CITY = 'BOSTON'
  MOVE SALARY(1) TO #SALARY
  COMPRESS 'SALARY  :' CURR-CODE(1) #SALARY INTO #FULL-SALARY
  COMPRESS 'VACATION:' LEAVE-DUE            INTO #VACATION
  /*
  DISPLAY NOTITLE NAME FIRST-NAME
          'JOB DESCRIPTION' JOB-TITLE  (LC='JOB     : ')  /
          '/'               #FULL-SALARY                  /
          '/'               #VACATION  (TC='DAYS')
  SKIP 1
END-READ
END
```

Output of Program COMPRX03:

```
       NAME                FIRST-NAME                JOB DESCRIPTION
-------------------- -------------------- ----------------------------------

SHAW                 LESLIE               JOB     : SECRETARY
                                          SALARY  : USD 18000
                                          VACATION: 2DAYS

STANWOOD             VERNON               JOB     : PROGRAMMER
                                          SALARY  : USD 31000
                                          VACATION: 1DAYS

CREMER               WALT                 JOB     : SECRETARY
                                          SALARY  : USD 20000
                                          VACATION: 3DAYS
```

Edit Masks

The following examples are referenced in the section *Edit Masks - EM Parameter*.

EDITMX03 - Edit mask (different EM for alpha-numeric fields)

```
** Example 'EDITMX03': Edit mask (different EM for alpha-numeric fields)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 CITY
  2 SALARY(1)
END-DEFINE
*
LIMIT 3
READ EMPLOY-VIEW BY PERSONNEL-ID FROM '20018000'
                 WHERE SALARY(1) = 28000 THRU 30000
  DISPLAY 'N A M E'  NAME       (EM=X^^X^^X^^X^^X^^X^^X^^X^^X^^X) /
          'NAME HEX' NAME       (EM=H^H^H^H^H^H^H^H^H^H)
                     FIRST-NAME (EM=' - 'X(15)*)
                     CITY       (EM=X..X(10))
  SKIP 1
END-READ
END
```

Output of Program EDITMX03:

```
Page      1                                             04-12-13  14:26:57

            N A M E                  FIRST-NAME         CITY
            NAME HEX
-------------------------------- ----------------- -------------

L  O  R  I  E                    - JEAN-PAUL      * C..LEVELAND
D3 D6 D9 C9 C5 40 40 40 40 40 40

H  A  L  L                       - ARTHUR         * A..NN ARBER
C8 C1 D3 D3 40 40 40 40 40 40 40

V  A  S  W  A  N  I              - TOMMIE         * M..ONTERREY
E5 C1 E2 E6 C1 D5 C9 40 40 40 40
```

EDITMX04 - Edit mask (different EM for numeric fields)

```
** Example 'EDITMX04': Edit mask (different EM for numeric fields)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 SALARY (1)
  2 BONUS  (1,1)
  2 LEAVE-DUE
END-DEFINE
*
LIMIT 2
READ EMPLOY-VIEW BY PERSONNEL-ID = '20018000'
                 WHERE SALARY(1) = 28000 THRU 30000
  DISPLAY (SF=4)
          'N A M E'    NAME
          'SALARY'     SALARY(1)  (EM=*USD^ZZZ,999)
          'BONUS (ZZ)' BONUS(1,1) (EM=S*ZZZ,999) /
          'BONUS (Z9)' BONUS(1,1) (EM=SZ99,999+) /
          '->' '='     BONUS(1,1) (EM=-999,999)
          'VAC/DUE'    LEAVE-DUE  (EM=+999)
  SKIP 1
END-READ
END
```

Output of Program EDITMX04:

```
Page      1                                                  04-12-13  14:27:43

      N A M E            SALARY       BONUS (ZZ)     VAC
                                      BONUS (Z9)     DUE
                                         BONUS
--------------------    -----------    -----------    ---

LORIE                   USD *28,000    +**4,000       +13
                                       + 04,000+
                                       ->  004,000

HALL                    USD *30,000    +**5,000       +14
                                       + 05,000+
                                       ->  005,000
```

EDITMX05 - Edit mask (EM for date and time system variables)

```
** Example 'EDITMX05': Edit mask (EM for date and time system variables)
************************************************************************
WRITE NOTITLE //
  'DATE INTERNAL :' *DATX (DF=L) /
  '              :' *DATX (EM=N(9)' 'ZW.'WEEK 'YYYY)  /
  '              :' *DATX (EM=ZZJ'.DAY 'YYYY)        /
  '     ROMAN    :' *DATX (EM=R)  /
  '     AMERICAN :' *DATX (EM=MM/DD/YYYY)      12X 'OR  ' *DAT4U /
  '     JULIAN   :' *DATX (EM=YYYYJJJ)         15X 'OR  ' *DAT4J /
  '     GREGORIAN:' *DATX (EM=ZD.''L(10)''YYYY) 5X 'OR  ' *DATG ///
  'TIME INTERNAL :' *TIMX                      14X 'OR  ' *TIME /
  '              :' *TIMX (EM=HH.II.SS.T) /
  '              :' *TIMX (EM=HH.II.SS' 'AP) /
  '              :' *TIMX (EM=HH)
END
```

Output of Program EDITMX05:

```



DATE INTERNAL : 2004-12-13
              : Monday 51.WEEK 2004
              : 348.DAY 2004
     ROMAN    : MMIV
     AMERICAN : 12/13/2004           OR   12/13/2004
     JULIAN   : 2004348              OR   2004348
     GREGORIAN: 13.December2004      OR   13December 2004


TIME INTERNAL : 14:28:49             OR   14:28:49.1
              : 14.28.49.1
```

```
              : 02.28.49 PM
              : 14
```

DISPLAY VERT with WRITE Statement

WRITEX10 - WRITE statement (with *n*T, T*`field` and P*`field`)

```
** Example 'WRITEX10': WRITE (with nT, T*field and P*field)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 JOB-TITLE
  2 NAME
  2 SALARY (1)
  2 BONUS  (1,1)
END-DEFINE
*
READ (3) EMPLOY-VIEW WITH JOB-TITLE FROM 'SALES PERSON'
  DISPLAY NOTITLE NAME 30T JOB-TITLE
          VERT AS 'SALARY/BONUS' SALARY(1) BONUS(1,1)
  AT BREAK OF JOB-TITLE
    WRITE 20T 'AVERAGE' T*JOB-TITLE OLD(JOB-TITLE) (AL=15)
              '(SAL)'   P*SALARY AVER(SALARY(1))  /
          46T '(BON)'   P*BONUS  AVER(BONUS(1,1)) /
  END-BREAK
  SKIP 1
END-READ
END
```

Output of Program WRITEX10:

```
        NAME                         CURRENT            SALARY
                                     POSITION           BONUS
--------------------         ------------------------- ----------

SAMUELSON                    SALES PERSON                  32000
                                                            6000

PAPAYANOPOULOS               SALES PERSON                  34000
                                                            7000

HELL                         SALES PERSON                  38000
                                                            9000

                   AVERAGE   SALES PERSON    (SAL)         34666
                                             (BON)          7333
```

AT BREAK Statement

The following example is referenced in the section *Control Breaks*.

ATBREX06 - AT BREAK OF statement (comparing NMIN, NAVER, NCOUNT with MIN, AVER, COUNT)

```
** Example 'ATBREX06': AT BREAK OF (comparing NMIN, NAVER, NCOUNT with
**                      MIN, AVER, COUNT)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 SALARY (1:2)
END-DEFINE
*
WRITE TITLE '-- SALARY STATISTICS BY CITY --' /
*
READ (2) EMPLOY-VIEW WITH CITY = 'NEW YORK'
  DISPLAY CITY  'SALARY (1)' SALARY(1) 15X 'SALARY (2)' SALARY(2)
  AT BREAK OF CITY
    WRITE /
      14T 'S A L A R Y   (1)'         39T 'S A L A R Y  (2)'          /
      13T '-   MIN:' MIN(SALARY(1))   38T '-   MIN:' MIN(SALARY(2))   /
      13T '-  AVER:' AVER(SALARY(1))  38T '-  AVER:' AVER(SALARY(2))  /
      16T COUNT(SALARY(1)) 'RECORDS'  41T COUNT(SALARY(2)) 'RECORDS'  //
      13T '-  NMIN:' NMIN(SALARY(1))  38T '-  NMIN:' NMIN(SALARY(2))  /
      13T '- NAVER:' NAVER(SALARY(1)) 38T '- NAVER:' NAVER(SALARY(2)) /
      16T NCOUNT(SALARY(1)) 'RECORDS' 41T NCOUNT(SALARY(2)) 'RECORDS'
  END-BREAK
END-READ
END
```

Output of Program ATBREX06:

```
                        -- SALARY STATISTICS BY CITY --

        CITY         SALARY (1)                SALARY (2)
-------------------- ----------                ----------

NEW YORK                  17000                     16100
NEW YORK                  38000                     34900

             S A L A R Y   (1)        S A L A R Y  (2)
            -   MIN:      17000      -   MIN:      16100
            -  AVER:      27500      -  AVER:      25500
                      2 RECORDS                2 RECORDS
```

```
            -  NMIN:      17000      -  NMIN:      16100
            - NAVER:      27500      - NAVER:      25500
                      2 RECORDS                2 RECORDS
```

COMPUTE, MOVE and COMPRESS Statements

The following examples are referenced in the section ***Data Computation***.

WRITEX11 - WRITE statement (with *n*X, *n*/*n* and COMPRESS)

```
** Example 'WRITEX11': WRITE (with nX, n/n and COMPRESS)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 SALARY       (1)
  2 FIRST-NAME
  2 NAME
  2 CITY
  2 ZIP
  2 CURR-CODE    (1)
  2 JOB-TITLE
  2 LEAVE-DUE
  2 ADDRESS-LINE (1)
*
1 #SALARY        (A8)
1 #FULL-NAME     (A25)
1 #FULL-CITY     (A25)
1 #FULL-SALARY   (A25)
1 #VACATION      (A16)
END-DEFINE
*
READ (3) EMPLOY-VIEW LOGICAL BY PERSONNEL-ID = '2001800'
  MOVE SALARY(1) TO  #SALARY
  COMPRESS FIRST-NAME NAME                    INTO #FULL-NAME
  COMPRESS ZIP        CITY                    INTO #FULL-CITY
  COMPRESS 'SALARY   :'  CURR-CODE(1)  #SALARY INTO #FULL-SALARY
  COMPRESS 'VACATION:'  LEAVE-DUE    'DAYS'   INTO #VACATION
  /*
  DISPLAY NOTITLE 'NAME AND ADDRESS' NAME
               5X 'PERS-NO.'         PERSONNEL-ID
               3X 'JOB TITLE'        JOB-TITLE (LC='JOB     :  ')
  WRITE   1/5 #FULL-NAME        1/37  #FULL-SALARY
          2/5 ADDRESS-LINE(1)   2/37  #VACATION
          3/5 #FULL-CITY
  SKIP 1
```

```
END-READ
END
```

Output of Program WRITEX11:

```
  NAME AND ADDRESS       PERS-NO.                JOB TITLE
--------------------     --------   -----------------------------------

FARRIS                   20018000   JOB     : PROGRAMMER
    JACKIE FARRIS                   SALARY  : USD 30500
    918 ELM STREET                  VACATION: 10 DAY
    32306 TALLAHASSEE

EVANS                    20018100   JOB     : PROGRAMMER
    JO EVANS                        SALARY  : USD 31000
    1058 REDSTONE LANE              VACATION: 11 DAY
    68508 LINCOLN

HERZOG                   20018200   JOB     : PROGRAMMER
    JOHN HERZOG                     SALARY  : USD 31500
    255 ZANG STREET #253            VACATION: 12 DAY
    27514 CHAPEL HILL
```

IFX03 - IF statement

```
** Example 'IFX03': IF
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 BONUS  (1,1)
  2 SALARY (1)
*
1 #INCOME  (N9)
1 #TEXT    (A26)
END-DEFINE
*
WRITE TITLE '-- DISTRIBUTION OF CATALOGS I AND II --' /
*
READ (3) EMPLOY-VIEW BY CITY = 'SAN FRANSISCO'
  COMPUTE #INCOME = BONUS(1,1) + SALARY(1)
  /*
  IF #INCOME > 40000
    MOVE 'CATALOGS I AND II' TO #TEXT
  ELSE
    MOVE 'CATALOG I'         TO #TEXT
  END-IF
  /*
```

```
  DISPLAY NAME 5X 'SALARY' SALARY(1) / BONUS(1,1)
  WRITE T*SALARY '-'(10) /
        16X 'INCOME:' T*SALARY #INCOME 3X #TEXT /
        16X '='(19)
  SKIP 1
END-READ
END
```

Output of Program IFX03:

```
                    -- DISTRIBUTION OF CATALOGS I AND II --

        NAME               SALARY
                           BONUS
--------------------     ----------

COLVILLE JR                   56000
                                  0
                         ----------
                INCOME:       56000   CATALOGS I AND II
                ===================

RICHMOND                       9150
                                  0
                         ----------
                INCOME:        9150   CATALOG I
                ===================

MONKTON                       13500
                                600
                         ----------
                INCOME:       14100   CATALOG I
                ===================
```

COMPRX03 - COMPRESS statement (using parameters LC and TC)

```
** Example 'COMPRX03': COMPRESS (using parameters LC and TC)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 SALARY     (1)
  2 CURR-CODE  (1)
  2 LEAVE-DUE
  2 NAME
  2 FIRST-NAME
  2 JOB-TITLE
*
1 #SALARY      (N9)
```

```
1 #FULL-SALARY (A25)
1 #VACATION    (A11)
END-DEFINE
*
READ (3) EMPLOY-VIEW WITH CITY = 'BOSTON'
  MOVE SALARY(1) TO #SALARY
  COMPRESS 'SALARY  :' CURR-CODE(1) #SALARY INTO #FULL-SALARY
  COMPRESS 'VACATION:' LEAVE-DUE            INTO #VACATION
  /*
  DISPLAY NOTITLE NAME FIRST-NAME
           'JOB DESCRIPTION' JOB-TITLE  (LC='JOB     : ')  /
           '/'               #FULL-SALARY                  /
           '/'               #VACATION  (TC='DAYS')
  SKIP 1
END-READ
END
```

Output of Program COMPRX03:

```
        NAME                FIRST-NAME                  JOB DESCRIPTION
-------------------- -------------------- -----------------------------------


SHAW                 LESLIE                JOB     : SECRETARY
                                           SALARY  : USD 18000
                                           VACATION: 2DAYS

STANWOOD             VERNON                JOB     : PROGRAMMER
                                           SALARY  : USD 31000
                                           VACATION: 1DAYS

CREMER               WALT                  JOB     : SECRETARY
                                           SALARY  : USD 20000
                                           VACATION: 3DAYS
```

System Variables

The following examples are referenced in the section *System Variables and System Functions*.

EDITMX05 - Edit mask (EM for date and time system variables)

```
** Example 'EDITMX05': Edit mask (EM for date and time system variables)
************************************************************************
WRITE NOTITLE //
  'DATE INTERNAL :' *DATX (DF=L) /
  '              :' *DATX (EM=N(9)' 'ZW.'WEEK 'YYYY)  /
  '              :' *DATX (EM=ZZJ'.DAY 'YYYY)        /
  '     ROMAN    :' *DATX (EM=R)  /
  '     AMERICAN :' *DATX (EM=MM/DD/YYYY)      12X 'OR  ' *DAT4U /
  '     JULIAN   :' *DATX (EM=YYYYJJJ)         15X 'OR  ' *DAT4J /
  '     GREGORIAN:' *DATX (EM=ZD.''L(10)''YYYY) 5X 'OR  ' *DATG ///
  'TIME INTERNAL :' *TIMX                      14X 'OR  ' *TIME /
  '              :' *TIMX (EM=HH.II.SS.T) /
  '              :' *TIMX (EM=HH.II.SS' 'AP) /
  '              :' *TIMX (EM=HH)
END
```

Output of Program EDITMX05:

```



DATE INTERNAL : 2004-12-13
              : Monday 51.WEEK 2004
              : 348.DAY 2004
     ROMAN    : MMIV
     AMERICAN : 12/13/2004            OR   12/13/2004
     JULIAN   : 2004348               OR   2004348
     GREGORIAN: 13.December2004       OR   13December 2004


TIME INTERNAL : 14:36:58              OR   14:36:58.8
              : 14.36.58.8
              : 02.36.58 PM
              : 14
```

READX04 - READ statement (in combination with FIND and the system variables *NUMBER and *COUNTER)

```
** Example 'READX04': READ (in combination with FIND and the system
**                    variables *NUMBER and *COUNTER)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
1 VEHIC-VIEW VIEW OF VEHICLES
```

```
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 10
RD. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  FD. FIND VEHIC-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      ENTER
    END-NOREC
    /*
    DISPLAY NOTITLE
            *COUNTER (RD.)(NL=8) NAME         (AL=15) FIRST-NAME (AL=10)
            *NUMBER  (FD.)(NL=8) *COUNTER (FD.)(NL=8) MAKE
  END-FIND
END-READ
END
```

Output of Program READX04:

```
  CNT          NAME       FIRST-NAME   NMBR      CNT             MAKE
--------- --------------- ---------- --------- --------- --------------------


        1 JONES           VIRGINIA           1         1 CHRYSLER
        2 JONES           MARSHA             2         1 CHRYSLER
        2 JONES           MARSHA             2         2 CHRYSLER
        3 JONES           ROBERT             1         1 GENERAL MOTORS
        4 JONES           LILLY              2         1 FORD
        4 JONES           LILLY              2         2 MG
        5 JONES           EDWARD             1         1 GENERAL MOTORS
        6 JONES           MARTHA             1         1 GENERAL MOTORS
        7 JONES           LAUREL             1         1 GENERAL MOTORS
        8 JONES           KEVIN              1         1 DATSUN
        9 JONES           GREGORY            1         1 FORD
       10 JOPER           MANFRED            0         0
```

WTITLX01 - WRITE TITLE statement (with *PAGE-NUMBER)

```
** Example 'WTITLX01': WRITE TITLE (with *PAGE-NUMBER)
************************************************************************
DEFINE DATA LOCAL
1 VEHIC-VIEW VIEW OF VEHICLES
  2 MAKE
  2 YEAR
  2 MAINT-COST (1)
END-DEFINE
*
LIMIT 5
*
```

```
READ VEHIC-VIEW
END-ALL
SORT BY YEAR USING MAKE MAINT-COST (1)
  DISPLAY NOTITLE YEAR MAKE MAINT-COST (1)
  AT BREAK OF YEAR
    MOVE 1 TO *PAGE-NUMBER
    NEWPAGE
  END-BREAK
  /*
  WRITE TITLE LEFT JUSTIFIED
        'YEAR:' YEAR 15X 'PAGE' *PAGE-NUMBER
END-SORT
END
```

Output of Program WTITLX01:

```
YEAR:  1980               PAGE      1
YEAR          MAKE         MAINT-COST
----- -------------------- ----------

 1980 RENAULT                 20000
 1980 RENAULT                 20000
 1980 PEUGEOT                 20000
```

System Functions

The following examples are referenced in the section *System Variables and System Functions*.

ATBREX06 - AT BREAK OF statement (comparing NMIN, NAVER, NCOUNT with MIN, AVER, COUNT)

```
** Example 'ATBREX06': AT BREAK OF (comparing NMIN, NAVER, NCOUNT with
**                       MIN, AVER, COUNT)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 SALARY (1:2)
END-DEFINE
*
WRITE TITLE '-- SALARY STATISTICS BY CITY --' /
*
READ (2) EMPLOY-VIEW WITH CITY = 'NEW YORK'
  DISPLAY CITY  'SALARY (1)' SALARY(1) 15X 'SALARY (2)' SALARY(2)
  AT BREAK OF CITY
    WRITE /
```

```
      14T 'S A L A R Y   (1)'         39T 'S A L A R Y  (2)'          /
      13T '-   MIN:' MIN(SALARY(1))   38T '-   MIN:' MIN(SALARY(2))   /
      13T '-  AVER:' AVER(SALARY(1))  38T '-  AVER:' AVER(SALARY(2))  /
      16T COUNT(SALARY(1)) 'RECORDS'  41T COUNT(SALARY(2)) 'RECORDS'  //
      13T '-  NMIN:' NMIN(SALARY(1))  38T '-  NMIN:' NMIN(SALARY(2))  /
      13T '- NAVER:' NAVER(SALARY(1)) 38T '- NAVER:' NAVER(SALARY(2)) /
      16T NCOUNT(SALARY(1)) 'RECORDS' 41T NCOUNT(SALARY(2)) 'RECORDS'
  END-BREAK
END-READ
END
```

Output of Program ATBREX06:

```
                        -- SALARY STATISTICS BY CITY --

        CITY         SALARY (1)               SALARY (2)
-------------------- ----------               ----------

NEW YORK                  17000                    16100
NEW YORK                  38000                    34900

             S A L A R Y   (1)        S A L A R Y  (2)
            -   MIN:      17000      -   MIN:      16100
            -  AVER:      27500      -  AVER:      25500
                      2 RECORDS                2 RECORDS

            -  NMIN:      17000      -  NMIN:      16100
            - NAVER:      27500      - NAVER:      25500
                      2 RECORDS                2 RECORDS
```

ATENPX01 - AT END OF PAGE statement (with system function available via GIVE SYSTEM FUNCTIONS in DISPLAY)

```
** Example 'ATENPX01': AT END OF PAGE (with system function available
**                     via GIVE SYSTEM FUNCTIONS in DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 SALARY (1)
END-DEFINE
*
READ (10) EMPLOY-VIEW BY PERSONNEL-ID = '20017000'
  DISPLAY NOTITLE GIVE SYSTEM FUNCTIONS
          NAME JOB-TITLE 'SALARY' SALARY(1)
  /*
  AT END OF PAGE
```

```
    WRITE / 24T 'AVERAGE SALARY: ...' AVER(SALARY(1))
  END-ENDPAGE
END-READ
END
```

Output of Program ATENPX01:

```
        NAME                     CURRENT            SALARY
                                POSITION
-------------------- ------------------------- ----------

CREMER               ANALYST                        34000
MARKUSH              TRAINEE                        22000
GEE                  MANAGER                        39500
KUNEY                DBA                            40200
NEEDHAM              PROGRAMMER                     32500
JACKSON              PROGRAMMER                     33000
PIETSCH              SECRETARY                      22000
PAUL                 SECRETARY                      23000
HERZOG               MANAGER                        48500
DEKKER               DBA                            48000

                       AVERAGE SALARY: ...      34270
```