Natural

Statements

Version 9.3.1

February 2025

ADABAS & NATURAL

This document applies to Natural Version 9.3.1 and all subsequent releases.

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

Copyright © 1992-2025 Software GmbH, Darmstadt, Germany and/or its subsidiaries and/or its affiliates and/or their licensors.

The name Software AG and all Software GmbH product names are either trademarks or registered trademarks of Software GmbH and/or its subsidiaries and/or its affiliates and/or their licensors. Other company and product names mentioned herein may be trademarks of their respective owners.

Detailed information on trademarks and patents owned by Software GmbH and/or its subsidiaries is located at https://softwareag.com/licenses.

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

This software may include portions of third-party products. For third-party copyright notices, license terms, additional rights or restrictions, please refer to "License Texts, Copyright Notices and Disclaimers of Third-Party Products". For certain specific third-party license restrictions, please refer to section E of the Legal Notices available under "License Terms and Conditions for Use of Software GmbH Products / Copyright and Trademark Notices of Software GmbH Products". These documents are part of the product documentation, located at https://softwareag.com/licenses and/or in the root installation directory of the licensed product(s).

Use, reproduction, transfer, publication or disclosure is prohibited except as specifically provided for in your License Agreement with Software GmbH.

Document ID: NATUX-NNATSTATEMENTS-931-20250213

Table of Contents

Preface

This document describes native Natural programming language (DML) statements and Natural SQL statements. It is organized under the following headings:

Statements Grouped by Function	Provides an overview of the Natural statements ordered by functional groups.
Syntax Symbols and Operand Definition Tables	Information on the symbols that are used within the diagrams that describe the syntax of Natural statements and on operand definition tables.
Using Natural SQL Statements	Describes rules specific to using Natural SQL statements.
Referenced Example Programs	Contains additional example programs that are referenced in the *Statements* and *System Variables* documentation.

Related Topics:

See also the *Programming Guide* for statement usage related topics such as: *User-Defined Variables* | *Dynamic and Large Variables* | *User-Defined Constants* | *Report Specification* | *Text Notation* | *User Comments* | *Rules for Arithmetic Assignment* | *Logical Condition Criteria* | *Function Call*

Statements in Alphabetical Order:

A - C	D - F	G - O	P - R	S - Z
ACCEPT/REJECT	DECIDE FOR	GET	PARSE XML	SELECT (SQL)
ADD	DECIDE ON	GET SAME	PASSW	SEND METHOD
ASSIGN	DEFINE CLASS	GET	PERFORM	SEPARATE
AT BREAK	DEFINE DATA	TRANSACTION	PERFORM BREAK	SET CONTROL
AT END OF DATA	DEFINE FUNCTION	DATA	PROCESSING	SET GLOBALS
AT END OF PAGE	DEFINE PRINTER	HISTOGRAM	PRINT	SET KEY
AT START OF DATA	DEFINE PROTOTYPE	IF	PROCESS	SET TIME
AT TOP OF PAGE	DEFINE	IF SELECTION	PROCESS COMMAND	SET WINDOW
BACKOUT	SUBROUTINE	IGNORE	PROCESS PAGE	SKIP
TRANSACTION	DEFINE WINDOW	INCLUDE	PROCESS	SORT
BEFORE BREAK	DEFINE WORK FILE	INPUT	SQL (SQL)	STACK
PROCESSING	DELETE	INSERT (SQL)	PROPERTY	STOP
CALL	DELETE (SQL)	INTERFACE	READ	STORE
CALL FILE	DISPLAY	LIMIT	READ RESULT SET	SUBTRACT
CALL LOOP	DIVIDE	LOOP	(SQL)	SUSPEND
CALLDBPROC (SQL)	DO/DOEND	METHOD	READ WORK FILE	IDENTICAL
CALLNAT	DOWNLOAD PC FILE	MOVE	READLOB	SUPPRESS
CLOSE	EJECT	MOVE INDEXED	REDEFINE	TERMINATE
CONVERSATION	END	MULTIPLY	REDUCE	UPDATE
CLOSE PC FILE	END TRANSACTION	NEWPAGE	REINPUT	UPDATE (SQL)
CLOSE PRINTER	ESCAPE	OBTAIN	REJECT	UPDATELOB

A - C	D - F	G - O	P - R	S - Z
CLOSE WORK FILE COMMIT (SQL) COMPRESS COMPUTE CREATE OBJECT	EXAMINE EXPAND FETCH FIND FOR FORMAT	ON ERROR OPEN CONVERSATION OPTIONS	RELEASE REPEAT REQUEST DOCUMENT RESET RESIZE RETRY ROLLBACK (SQL) RUN	UPLOAD PC FILE WRITE WRITE TITLE WRITE TRAILER WRITE WORK FILE

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://techcommunity.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://containers.softwareag.com/products** 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 GmbH 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

2 Statements Grouped by Function

Notes:

1. Certain statements can be used both in structured mode and in reporting mode, while others can be used in reporting mode only. See *Natural Programming Modes* in the *Programming Guide*.
2. The statements `DLOGOFF`, `DLOGON`, `SHOW`, `IMPORT` and `EXPORT` are only available when Entire DB is installed. For a description, see the *Entire DB* documentation.

Database Access and Update

The following types of statements are available:

- Natural DML Statements
- Natural SQL Statements

Natural DML Statements

The following Natural data manipulation language (DML) statements are used to access and manipulate information contained in a database.

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

AT END OF DATA	Specifies statements to be performed after the last of a set of records has been processed in a processing loop.
AT BREAK	Specifies statements to be performed when the value of a control field changes (break processing).
BEFORE BREAK PROCESSING	Specifies statements to be performed before performing break processing.
PERFORM BREAK PROCESSING	Immediately invokes break processing.

Natural SQL Statements

In addition to the Natural DML statements, Natural also provides SQL statements for use in Natural programs that manipulate data on an SQL database.

The following Natural SQL statements are available:

CALLDBPROC	Invokes a stored procedure of the SQL database system to which Natural is connected.
COMMIT	Indicates the end of a logical transaction and releases all data locked during the transaction. All data modifications are committed and made permanent.
DELETE	Deletes either rows in a table without using a cursor (**"searched"** DELETE) or rows in a table to which a cursor is positioned (**"positioned"** DELETE).
INSERT	Adds one or more new rows to a table.
PROCESS SQL	Issues SQL statements to the underlying database.
READ RESULT SET	Reads a result set which was created by a stored procedure that was invoked by a previous CALLDBPROC statement.
ROLLBACK	Undoes all database modifications made since the beginning of the last recovery unit.
SELECT	Supports both the **cursor-oriented selection** that is used to retrieve an arbitrary number of rows and the **non-cursor selection** (singleton SELECT) that retrieves at most one single row.
UPDATE	Performs an update operation on either rows in a table without using a cursor (**"searched"** UPDATE) or columns in a row to which a cursor is positioned (**"positioned"** UPDATE).

Arithmetic and Data Movement Operations

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

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

Loop Execution

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

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

Creation of Output Reports

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

FORMAT	Specifies output parameter settings.
DISPLAY	Specifies fields to be output in column form.
WRITE / PRINT	Specifies fields to be output in non-column form.
WRITE TITLE	Specifies text to be output at the top of each page of a report.
WRITE TRAILER	Specifies text to be output at the bottom of each page of a report.
AT TOP OF PAGE	Specifies processing to be performed when a new output page is started.
AT END OF PAGE	Specifies processing to be performed when the end of an output page is reached.

SKIP	Generates one or more blank lines in a report.
EJECT	Causes a page advance without titles or headings.
NEWPAGE	Causes a page advance with titles and headings.
SUSPEND IDENTICAL SUPPRESS	Suspends identical suppression for a single record.
DEFINE PRINTER	Allocates a report to a logical output destination.
CLOSE PRINTER	Closes a printer.

Screen Generation for Interactive Processing

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

INPUT	Creates a formatted screen (map) for data display/ entry.
REINPUT	Re-executes an INPUT statement (if invalid data were entered in response to the previous INPUT statement).
DEFINE WINDOW	Specifies the size, position and attributes of a window.
SET WINDOW	Activates and de-activates a window.
PROCESS PAGE	Creates a data mapping to a web rich GUI screen.
PROCESS PAGE USING	Performs rich GUI I/O processing using an adapter object generated from a page layout.
PROCESS PAGE UPDATE	Re-executes a PROCESS PAGE statement.
PROCESS PAGE MODAL	Initiates a processing block and controls the lifetime of a rich GUI window.

Processing of Logical Conditions

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

IF	Performs statements depending on a logical condition.
IF SELECTION	Verifies that in a sequence of alphanumeric fields one and only one contains a value.
DECIDE FOR	Performs statements depending on logical conditions.
DECIDE ON	Performs statements depending on the contents of a variable.

Invoking Programs and Routines

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

CALL	Invokes a non-Natural program from a Natural program.
CALLNAT	Invokes a Natural subprogram.
CALL FILE	Invokes a non-Natural program to read a record from a non-Adabas file.
CALL LOOP	Generates a processing loop containing a call to a non-Natural program.
DEFINE SUBROUTINE	Defines a Natural subroutine.
ESCAPE	Stops the execution of a routine.
FETCH	Invokes a Natural program.
PERFORM	Invokes a Natural subroutine.
PROCESS COMMAND	Invokes a command processor.
RUN	Compiles and executes a source program.

Functions

The following Natural statements are used to create functions:

DEFINE FUNCTION	Creates functions which can be called instead of operands in Natural statements. Functions are defined in Natural objects of type function.
DEFINE PROTOTYPE	Specifies the properties to be used for a function call.
Function Call	Used to call Natural objects of type function.

Program and Session Termination

The following Natural statements are used to terminate the execution of an application or to terminate the Natural session.

STOP	Terminates the execution of an application.
TERMINATE	Terminates the Natural session.

Control of Work Files / PC Files

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

WRITE WORK FILE	Writes data to a work file.
DOWNLOAD PC FILE	Enables transfer data from a mainframe or a Linux platform to the PC.
READ WORK FILE	Reads data from a work file.
UPLOAD PC FILE	Enables transfer data from a PC to a mainframe or a Linux platform.
CLOSE WORK FILE	Closes a work file.
CLOSE PC FILE	Closes a specific PC work file.
DEFINE WORK FILE	Assigns a file name to a work file.

Component Based Programming

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

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

Memory Management Control for Dynamic Variables or X-Arrays

EXPAND	Expands the allocated memory of dynamic variables to a given size or expands the number of occurrences of X-arrays.
REDUCE	Reduces the size of a dynamic variable or the number of occurrences of X-arrays.
RESIZE	Adjusts the size of a dynamic variableor the number of occurrences of X-arrays.

Natural Remote Procedure Call

OPEN CONVERSATION	Allows the RPC Client to open a conversation and specify the remote subprograms to be included in the conversation.
CLOSE CONVERSATION	Allows the client to close conversations. You can close the current conversation, another open conversation, or all open conversations.
DEFINE DATA CONTEXT	Defines variables known as context variables, which are meant to be available to multiple remote subprograms within one conversation, without having to explicitly pass the variables as parameters with the corresponding CALLNAT statements.

See also the section *Natural Statements Involved* in the *Natural RPC (Remote Procedure Call)* documentation.

Internet and XML

PARSE	Allows you to parse XML documents from a Natural program.
REQUEST DOCUMENT	Allows you to access an external system.

Miscellaneous

DEFINE DATA	Defines the data elements which are to be used in a Natural program or routine.
END	Indicates the end of the source code of a Natural program or routine.
INCLUDE	Incorporates Natural copycode at compilation.
ON ERROR	Intercepts runtime errors which would otherwise result in a Natural error message, followed by the termination of the Natural program.
RELEASE	Deletes the contents of the Natural stack; releases sets of ISN sets retained via a FIND statement; releases Natural global variables.
SET CONTROL	Performs a Natural terminal command from within a Natural program.
SET KEY	Assigns functions to terminal keys.
SET GLOBALS	Sets values for session parameters.
SET TIME	Establishes a point-in-time reference for a *TIMD system variable.
STACK	Places data and/or commands into the Natural stack.

Reporting Mode Statements

The following statements are for reporting mode only:

LOOP	Closes a processing loop.
DO/DOEND	Specify a group of statements to be executed based on a logical condition.
OBTAIN	Causes one or more fields to be read from a file.
REDEFINE	Redefines a field.

The following statements can be used both in structured mode and in reporting mode, however, the statement structure and, with some of them, the functionality is different:

AT START OF DATA	Specifies statements to be performed when the first of a set of records is processed in a processing loop.
AT END OF DATA	Specifies statements to be performed after the last of a set of records has been processed in a processing loop.
AT BREAK	Specifies statements to be performed when the value of a control field changes (break processing).
AT TOP OF PAGE	Specifies processing to be performed when a new output page is started.
AT END OF PAGE	Specifies processing to be performed when the end of an output page is reached.
BEFORE BREAK PROCESSING	Specifies statements to be performed before performing break processing.
CALL LOOP	Generates a processing loop containing a call to a non-Natural program.
CALL FILE	Invokes a non-Natural program to read a record from a non-Adabas file.
COMPUTE	Performs arithmetic operations or assigns values to fields.
DEFINE SUBROUTINE	Defines a Natural subroutine.
ESCAPE	Stops the execution of a processing loop.
FIND	Selects records from a database file based on user-specified criteria.
GET SAME	Re-reads the record currently being processed.
HISTOGRAM	Reads the values of a database field.
IF	Performs statements depending on a logical condition.
IF SELECTION	Verifies that in a sequence of alphanumeric fields one and only one contains a value.
ON ERROR	Intercepts runtime errors which would otherwise result in a Natural error message, followed by the termination of the Natural program.
READ	Reads a database file in physical or logical sequence of records.
READ WORK FILE	Reads data from a work file.
REPEAT	Initiates a processing loop (and terminates it based on a specified condition).

SORT	Sorts records.
STORE	Adds a new record to the database.
UPDATE	Updates a record in the database.
UPLOAD PC FILE	Enables transfer data from a PC to a mainframe or a Linux platform.

3 Syntax Symbols and Operand Definition Tables

Syntax Symbols

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

Syntax Symbol	Description
ABCDEF	Upper-case non-italic letters indicate that the term is either a Natural keyword or a Natural reserved word that must be entered exactly as specified.
ABCDEF	If an optional term in upper-case letters is completely underlined (not a hyperlink!), this indicates that the term is the default value. If you omit the term, the underlined value applies.
ABCDEF	If a term in upper-case letters is partially underlined (not a hyperlink!), this indicates that the underlined portion is an acceptable abbreviation of the term.
abcdef	Letters in italics are used to represent variable information. You must supply a valid value when specifying this term. **Note:** In place of *statement* or *statements*, you must supply one or several suitable statements, depending on the situation. If you do not want to supply a specific statement, you may insert the IGNORE statement.
[ ]	Elements contained within square brackets are optional. If the square brackets contain several lines stacked one above the other, each line is an optional alternative. You may choose at most one of the alternatives.
{ }	If the braces contain several lines stacked one above the other, each line is an alternative. You must choose exactly one of the alternatives.
|	The vertical bar separates alternatives.
...	A term preceding an ellipsis may optionally be repeated. A number after the ellipsis indicates how many times the term may be repeated. If the term preceding the ellipsis is an expression enclosed in square brackets or braces, the ellipsis applies to the entire bracketed expression.
,...	A term preceding a comma-ellipsis may optionally be repeated; if it is repeated, the repetitions must be separated by commas. A number after the comma-ellipsis indicates how many times the term may be repeated. If the term preceding the comma-ellipsis is an expression enclosed in square brackets or braces, the comma-ellipsis applies to the entire bracketed expression.
:...	A term preceding a colon-ellipsis may optionally be repeated; if it is repeated, the repetitions must be separated by colons. A number after the colon-ellipsis indicates how many times the term may be repeated. If the term preceding the colon-ellipsis is an expression enclosed in square brackets or braces, the colon-ellipsis applies to the entire bracketed expression.

Syntax Symbol	Description
Other symbols (except `[ ] { } | ... ,... :...`)	All other symbols except those defined in this table must be entered exactly as specified. *Exception*: The SQL scalar concatenation operator is represented by two vertical bars that must be entered literally as they appear in the syntax definition.

Example:

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

- `WRITE`, `USING`, `MAP` and `FORM` are Natural keywords which you must enter as specified.
- *`operand1`* and *`operand2`* are user-supplied variables for which you specify the names of the objects you wish to deal with.
- The braces indicate that you must choose whether to specify either `FORM` or `MAP`; however, you must specify one of the two.
- The square brackets indicate that `USING` and *`operand2`* are optional elements which you can, but need not, specify.
- The ellipsis indicates that you may specify *`operand2`* several times.

Operand Definition Table

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

Operand	Possible Structure						Possible Formats												Referencing Permitted	Dynamic Definition
operand1	C	S	A	G	N/M	E	A	U	N	P	I	F	B	D	T	L	C	O	yes/no	yes/no

This table provides the following information on each operand:

Possible Structure

Indicates the structure which the operand may take:

C	Constant.	
S	Single occurrence (scalar; that is, a field/variable which is neither an array nor a group).	
A	Array.	
G	Group.	
N/M	Natural system variable:	
	N	All system variables can be used.
	M	Only *modifiable* system variables can be used. For information on whether the content of a system variable is modifiable or not, see the Natural *System Variables* documentation.
E	Arithmetic expressions.	

Possible Formats

Indicates the format which the operand may take:

A	Alphanumeric (ASCII code page)
U	Alphanumeric (Unicode)
N	Numeric unpacked
P	Packed numeric
I	Integer
F	Floating point
B	Binary
D	Date
T	Time
L	Logical
C	Attribute control
O	`HANDLE OF OBJECT`

Referencing Permitted

Indicates whether the operand may be referenced or not, using a statement label or the source code line number.

Dynamic Definition

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

II Using Natural SQL Statements

In addition to the native Natural DML statements, Natural provides Natural SQL statements for use in Natural programs that maintain data contained in an SQL or SQL-compliant database.

This chapter describes the special syntax rules and conventions that apply when using Natural SQL statements.

Overview of Natural SQL Statements:

`CALLDBPROC` | `COMMIT` | `DELETE` | `INSERT` | `PROCESS SQL` | `READ RESULT SET` | `ROLLBACK` | `SELECT` | `UPDATE`

4 Common Set and Extended Set

The SQL statements available within the Natural programming language comprise two different syntax sets:

- **Common Set**
  The Common Set basically corresponds to the standard SQL syntax definitions and is provided for each SQL-compliant database system supported by Natural. The Common Set is valid against all SQL databases.
- **Extended Set**
  The Extended Set, in addition, provides special enhancements to the Common Set to support specific features of the supported database systems. Currently, the Extended Set is partly available and is valid against Db2 databases only.

The Natural SQL statements documentation mainly describes the Natural SQL Common Set. The statement syntax adheres as far as possible to the syntax described in the relevant literature on SQL; please, refer to this literature for further details.

5 Basic Syntactical Items

This chapter describes basic syntactical items, which are referenced within the individual SQL statement descriptions.

Constants

The constants used in the syntactical descriptions of the Natural SQL statements are:

constant	The item *constant* refers to either a Natural constant or an **SQL** *datetime constant*.
integer	The item *integer* always represents an integer constant.

Note: If the character for decimal point notation (session parameter DC) is set to a comma (,), any specified numeric constant must not be followed directly by a comma, but must be separated from it by a blank character; otherwise an error or wrong results occur.

Invalid Syntax:	Valid Syntax:
VALUES (1,'A') leads to a syntax error.	VALUES (1 ,'A')
VALUES (1,2,3) leads to wrong results.	VALUES (1 ,2 ,3)

SQL Datetime Constants

An SQL datetime constant is a character string constant of a particular format that specifies one of the following:

DATE *string-constant*	Specifies an SQL date constant, for example: DATE '2013-15-01'.
TIME *string-constant*	Specifies an SQL time constant, for example: TIME '10:30:15'.
TIMESTAMP *string-constant*	Specifies an SQL time stamp constant, for example: TIMESTAMP '2014-15-01 10:20:15.123456'.

For information on the valid *string-constant* formats, refer to IBM's *Db2 SQL reference information*.

Names

The names used in the syntactical descriptions of the Natural SQL statements are:

- authorization-identifier
- ddm-name
- view-name
- column-name
- location-name

- table-name
- correlation-name

authorization-identifier

The item *authorization-identifier*, which is also called creator name, is used to qualify database tables and views. See also *authorization-identifier* under *table-name* below.

ddm-name

The item *ddm-name* always refers to the name of a Natural data definition module (DDM) as created with the Natural DDM Services.

view-name

The item *view-name* always refers to the name of a Natural view as defined in the `DEFINE DATA` statement.

column-name

The item *column-name* always refers to the name of a physical database column.

location-name

The item *location-name* always denotes the location of the table. Specification of location-name is optional and belongs to the **SQL Extended Set**.

table-name

The item *table-name* in this section is used to reference both SQL base tables and SQL viewed tables.

Syntax of item *table-name*:

```
[[location-name.]authorization-identifier.]ddm-name
```

Syntax Element Description:

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td>ddm-name</td><td>A Natural data definition module (DDM) must have been created for a table to be used. The name of such a DDM must be the same as the corresponding database table name or view name.</td></tr>
<tr><td>location-name</td><td>This optional item specifies the location of the table to be accessed.</td></tr>
<tr><td>authorization-identifier</td><td>There are two ways of specifying the authorization-identifier of a database table or view.

One way corresponds to the standard SQL syntax, in which the authorization-identifier is separated from the table name by a period. Using this form, the name of the DDM must be the same as the name of the database table without the authorization-identifier.

Example:

<pre>DEFINE DATA LOCAL
01 PERS VIEW OF PERSONNEL
   02 NAME
   02 AGE
END-DEFINE
SELECT *
  INTO VIEW PERS
  FROM SQL.PERSONNEL
...</pre>Alternatively, you can define the authorization-identifier as part of the DDM name. The DDM name then consists of the authorization-identifier and the database table name separated by a hyphen (-). The hyphen between the authorization-identifier and the table name is converted internally into a period.

Note: This form of DDM name can also be used with a FIND or READ statement, because it conforms to the DDM naming conventions applicable to these statements.

Example:

<pre>DEFINE DATA LOCAL
01 PERS VIEW OF SQL-PERSONNEL
   02 NAME
   02 AGE
END-DEFINE
SELECT *
  INTO VIEW PERS
  FROM SQL-PERSONNEL
...</pre>If the authorization-identifier has been specified neither explicitly nor within the DDM name, it is determined by the SQL database system.</td></tr>
</table>

Syntax Element	Description

	In addition to being used in `SELECT` statements, table names can also be specified in `DELETE`, `INSERT` and `UPDATE` statements.
	
	Examples:
	
	`...`
	`DELETE FROM SQL.PERSONNEL`
	`  WHERE AGE IS NULL`
	` ...`
	
	`...`
	`INSERT INTO SQL.PERSONNEL (NAME,AGE)`
	`  VALUES ('ADKINSON',35)`
	`...`
	
	`...`
	`UPDATE SQL.PERSONNEL`
	`SET SALARY = SALARY * 1.1`
	`WHERE AGE > 30`
	`...`

correlation-name

The item *correlation-name* represents an alias name for a *table-name*. It can be used to qualify column names; it also serves to implicitly qualify fields in a Natural view when used with the `INTO` clause of the `SELECT` statement.

Example:

```
DEFINE DATA LOCAL
01 PERS-NAME    (A20)
01 EMPL-NAME    (A20)
01 AGE          (I2)
END-DEFINE
...
SELECT X.NAME , Y.NAME , X.AGE
  INTO PERS-NAME , EMPL-NAME , AGE
  FROM SQL-PERSONNEL X , SQL-EMPLOYEES Y
  WHERE X.AGE = Y.AGE
END-SELECT
...
```

Although in most cases the use of *correlation-names* is not necessary, they may help to make the statement clearer.

Parameters

Syntax of item *parameter*:

```
[:] host-variable [INDICATOR [:] host-variable] [LINDICATOR [:] host-variable]
```

Syntax Element Description:

Syntax Element	Description
host-variable	A *host-variable* is a Natural user-defined variable (no system variable) which is referenced in an SQL statement. It can be either an individual field or defined as part of a Natural view. When defined as a receiving field (for example, in the `INTO` clause), a *host-variable* identifies a variable to which a value is assigned by the database system. When defined as a sending field (for example, in the `WHERE` clause), a *host-variable* specifies a value to be passed from the program to the database system. See also ***Natural Formats and SQL Data Types***.

[:]	**Colon:**
	
	To comply with SQL standards, a *host-variable* can also be prefixed by a colon (:). When used with flexible SQL, *host-variables* must be qualified by colons.
	
	Example:
	
	`SELECT NAME INTO :#NAME FROM PERSONNEL`
	`WHERE AGE = :VALUE`
	
	The colon is always required if the variable name is identical to an SQL reserved word. In a context in which either a *host-variable* or a column can be referenced, the use of a name without a colon is interpreted as a reference to a column.

INDICATOR	**INDICATOR Clause:** The `INDICATOR` clause is an optional feature to distinguish between a “null” value (that is, no value at all) and the actual values `0` or “blank”. When specified with a receiving *host-variable* (target field), the `INDICATOR` *host-variable* (null indicator field) serves to find out whether a column to be retrieved is “null”. Example:

Syntax Element	Description

	`DEFINE DATA LOCAL`
	`1 NAME      (A20)`
	`1 NAMEIND   (I2)`
	`END-DEFINE`
	`SELECT *`
	`  INTO NAME INDICATOR NAMEIND`
	`...`
	
	In this example, `NAME` represents the receiving *host-variable* and `NAMEIND` the null indicator field.
	
	If a null indicator field has been specified and the column to be retrieved is null, the value of the null indicator field is negative and the target field is set to `0` or “blank” depending on its data type. Otherwise, the value of the null indicator field is greater than or equal to `0`.
	
	When specified with a sending *host-variable* (source field), the null indicator field is used to designate a null value for this field.
	
	Example:
	
	`DEFINE DATA LOCAL`
	`1 NAME      (A20)`
	`1 NAMEIND   (I2)`
	`UPDATE ...`
	`SET NAME = :NAME INDICATOR :NAMEIND`
	`WHERE ...`
	
	In this example, `:NAME` represents the sending *host-variable* and `:NAMEIND` the null indicator field. By entering a negative value as input for the null indicator field, a null value is assigned to a database column.
	
	An `INDICATOR` *host-variable* is of format/length I2.

`LINDICATOR`	**LINDICATOR Clause:** The `LINDICATOR` clause is an optional feature which is used to support columns of varying lengths, for example, `VARCHAR` or `LONG VARCHAR` type. When specified with a receiving *host-variable* (target field), the `LINDICATOR` *host-variable* (length indicator field) contains the number of characters actually returned by the database into the target field. The target field is always padded with blanks. If the `VARCHAR` or `LONG VARCHAR` column contains more characters than fit in the target field, the length indicator field is set to the length actually returned (that is, the length of the target field) and the null indicator field (if specified) is set to the total length of this column. Example

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td></td><td>

<pre>DEFINE DATA LOCAL
1 ADDRESSLIND (I2)
1 ADDRESS   (A50/1:6)
END-DEFINE
SELECT *
  INTO :ADDRESS(*) LINDICATOR :ADDRESSLIND
  ...</pre>

In this example, :ADDRESS(*) represents the target field which receives the first 300 bytes (if available) of the addressed VARCHAR or LONG VARCHAR column, and :ADDRESSLIND represents the length indicator field which contains the number of characters actually returned.

When specified with a sending *host-variable* (source field), the length indicator field specifies the number of characters of the source field which are to be passed to the database.

Example:

<pre>DEFINE DATA LOCAL
1 NAMELIND (I2)
1 NAME   (A20)
1 AGE   (I2)
END-DEFINE
MOVE 4   TO NAMELIND
MOVE 'ABC%' TO NAME
SELECT AGE
  INTO :AGE
WHERE NAME LIKE :NAME LINDICATOR :NAMELIND
  ...</pre>

A LINDICATOR *host-variable* is of format/length I2 or I4. For performance reasons, it should be specified immediately before the corresponding target or source field; otherwise, the field is copied to the temporary storage at runtime.

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

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

Let a fixed length field be defined with a LINDICATOR field specified as I2. If the VARCHAR column contains more characters than fit into this fixed length field, the length indicator field is set to the length actually returned and the null indicator field (if specified) is set to the total length of this column (retrieval). This is not possible for fixed length fields greater than or equal to 32 KB (length does not fit into null indicator field).

</td></tr>
</table>

Natural Formats and SQL Data Types

The Natural data format of a **host-variable** is converted to an SQL data type according to the following table:

Natural Format/Length	SQL Data Type
A*n*	CHAR (*n*)
B2	SMALLINT
B4	INT
B*n*; *n* not equal to 2 or 4	CHAR (*n*)
F4	REAL
F8	DOUBLE PRECISION
I2	SMALLINT
I4	INT
N*nn.m*	NUMERIC (*nn*+*m*,*m*)
P*nn.m*	NUMERIC (*nn*+*m*,*m*)
T	TIME
D	DATE
G*n*; for view fields only	GRAPHIC (*n*)

Natural does not check whether the converted SQL data type is compatible to the database column. Except for fields of format N, no data conversion is done.

In addition, the following extensions to standard Natural formats are available with Natural SQL:

- A one-dimensional array of format A can be used to support alphanumeric columns longer than 253 bytes. This array must be defined beginning with index 1 and can only be referenced by using an asterisk (*) as the index. The corresponding SQL data type is CHAR (*n*), where *n* is the total number of bytes in the array.
- A special *host-variable* indicated by the keyword LINDICATOR can be used to support variable-length columns. The corresponding SQL data type is VARCHAR (*n*); see also the LINDICATOR clause.
- The Natural formats date (D) and time (T) can be used with Entire Access and will be converted into the corresponding database-dependent formats (see the Entire Access documentation for details)

A sending field specified as one-dimensional array without a LINDICATOR field is converted into the SQL data type VARCHAR. The length is the total number of bytes in the array, not taking into account trailing blanks.

6 Natural View Concept

Some Natural SQL statements also support the use of Natural views.

A Natural view can be specified instead of a parameter list, where each field of the view - except group fields, redefining fields and fields prefixed with `L@` or `N@`- corresponds to one parameter (host variable).

Fields with names prefixed with `L@` or `N@` can only exist with corresponding master fields; that is, fields of the same name, where:

- `L@` fields are converted into `LINDICATOR` fields,
- `N@` fields are converted into `INDICATOR` fields.

`L@` fields should have been specified at view definition, immediately before the master fields to which they apply.

```
DEFINE DATA LOCAL
01 PERS VIEW OF SQL-PERSONNEL
  02 PERSID     (I4)
  02 NAME       (A20)
  02 N@NAME    (I2)                     /* null indicator of NAME
  02 L@ADDRESS (I2)                     /* length indicator of ADDRESS
  02 ADDRESS    (A50/1:6)
  02 N@ADDRESS (I2)                     /* null indicator of ADDRESS
01 #PERSID      (I4)
END-DEFINE
   ...
SELECT *
  INTO VIEW PERS
  FROM SQL-PERSONNEL
  WHERE PERSID = #PERSID
   ...
END-SELECT
```

The above example is equivalent to the following one:

```
...
SELECT *
  INTO PERSID,
       NAME INDICATOR N@NAME,
       ADDRESS(*)INDICATOR N@ADDRESS LINDICATOR L@ADDRESS
  FROM SQL-PERSONNEL
  WHERE PERSID = #PERSID
...
END-SELECT
```

Note: When accessing `VARCHAR` data types with Natural for Windows or Natural for Linux and Cloud, there must be a corresponding length indicator variable in the view.

7 Scalar Expressions

```
{ [ + ] { factor               }                                }
{ [ - ] { (scalar-expression)  }                                }
{                                                               }
{ scalar-expression scalar-operator scalar-expression           }
```

Scalar Expression

A `scalar-expression` consists of a **factor** or other scalar expressions including scalar operators.

Concerning reference priority, scalar expressions behave as follows:

- When a non-qualified variable name is specified in a scalar expression, the first approach is to resolve the variable name as column name of the referenced table.
- If no column with the specified name is available in the referenced table, Natural tries to resolve this variable as a Natural user-defined variable (host variable).

Scalar Operator

```
{ +      }
{ -      }
{ *      }
{ /      }
{ | |    }
{ CONCAT }
```

A `scalar-operator` can be any of the operators listed above. The minus (-) and slash (/) operators must be separated by at least one blank from preceding operators.

Factor

Common Set Syntax:

```
{ atom                }
{ column-reference    }
{ aggregate-function  }
{ special-register    }
```

Extended Set Syntax:

```
  atom
{ column-reference    }
{ aggregate-function  }
{ special-register    }
{ scalar-function     }
{ length-stringunit   }
  labeled-duration
```

A *factor* can consist of one of the items listed in the above diagram and described in the text below.

Atom

```
{ parameter }
{ constant  }
```

An *atom* can be either a *parameter* or a *constant*.

Column Reference

```
[ table-name.       ]
[ correlation-name. ] column-name
```

A *column-reference* is a column name optionally qualified by either a *table-name* or a *correlation-name* (see also the section ***Basic Syntactical Items***). Qualified names are often clearer than unqualified names and sometimes they are essential.

Note: A table name in this context must not be qualified explicitly with an authorization identifier. Use a correlation name instead if you need a qualified table name.

If a column is referenced by a *table-name* or *correlation-name*, it must be contained in the corresponding table. If neither a *table-name* nor a *correlation-name* is specified, the respective column must be in one of the tables specified in the FROM clause (see ***Table Expression***).

Aggregate Function

```
{ COUNT            { (*)                              }
                   { (DISTINCT column-reference)      }

  { AVG }
  { MAX }          { (DISTINCT column-reference)      }
  { MIN }          { ([ALL] scalar-expression)        }
  { SUM }                                             }
```

SQL provides a number of special functions to enhance its basic retrieval power. The so-called SQL aggregate functions currently available and supported by Natural are:

AVG	gives the average of the values in a column
COUNT	gives the number of values in a column
MAX	gives the highest value in a column
MIN	gives the lowest value in a column
SUM	gives the sum of the values in a column

Apart from COUNT(*), each of these functions operates on the collection of scalar values in an argument (that is, a single column or a *scalar-expression*) and produces a scalar value as its result.

Example:

```
DEFINE DATA LOCAL
1 AVGAGE   (I2)
END-DEFINE
...
SELECT AVG (AGE)
  INTO AVGAGE
  FROM SQL-PERSONNEL
  ...
```

DISTINCT

In general, the argument can optionally be preceded by the keyword DISTINCT to eliminate redundant duplicate values before the function is applied.

If DISTINCT is specified, the argument must be the name of a single column; if DISTINCT is omitted, the argument can consist of a general *scalar-expression*.

`DISTINCT` is not allowed with the special function `COUNT(*)`, which is provided to count all rows without eliminating any duplicates.

Special Register

special-register

```
USER
```

With the exception of `USER`, the following special registers do not conform to standard SQL. They are specific to Db2 and belong to the Natural SQL **Extended Set**:

```
CURRENT DATE
CURRENT_DATE
CURRENT TIME
CURRENT_TIME
CURRENT TIMESTAMP
CURRENT CLIENT_ACCTNG
CLIENT ACCTNG
CURRENT CLIENT_APPLNAME
CLIENT APPLNAME
CURRENT CLIENT_USERID
CLIENT USERID
CURRENT CLIENT_WRKSTNNAME
CLIENT WRKSTNNAME
CURRENT DEGREE
CURRENT TIMEZONE
CURRENT SERVER
CURRENT_TIMEZONE
CURRENT_SERVER
SESSION_USER
CURRENT_PATH
CURRENT SCHEMA
CURRENT DECFLOAT ROUNDING MODE
CURRENT LOCK TIMEOUT
CURRENT PACKAGE PATH
CURRENT REFRESH AGE
CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION ↩
```

A reference to a *special-register* returns a scalar value.

Scalar Function

scalar-function

A scalar function is a built-in function that can be used in the construction of scalar computational expressions.

Scalar functions are specific to Db2 and belong to the Natural SQL **Extended Set**.

The scalar functions Natural for Db2 supports are listed below:

```
COALESCE
DATE
TIME
TIMESTAMP
VALUE
```

Each scalar function is followed by one or more scalar expressions in parentheses. The number of scalar expressions depends upon the scalar function. Multiple scalar expressions must be separated from one another by commas.

Example:

```
SELECT NAME
  INTO NAME
  FROM SQL-PERSONNEL
  WHERE  VALUE(NAME, CITY)  = 'VIZAG'
       ...
```

Length of String Unit

length-stringunit

Specifies the unit used for the length of a string. Commonly used for SQL scalar string functions. The supported length of string units are listed below:

```
OCTETS
CODEUNITS16
CODEUNITS32
```

where `OCTETS` specifies that the length is expressed in bytes, `CODEUNITS16` specifies that the length is expressed in 16-bit UTF-16 code units, and `CODEUNITS32` specifies that the length is expressed in 32-bit UTF-32 code units.

Labeled Duration

labeled-duration

```
                      YEAR
                      YEARS
                      MONTH
                      MONTHS
                      DAY
                      DAYS
scalar-expression     HOUR
                      HOURS
                      MINUTE
                      MINUTES
                      SECOND
                      SECONDS
                      MICROSECOND
                      MICROSECONDS
```

A *labeled-duration* denotes a specific unit of time as expressed by a number which can be an expression followed by one of the duration keywords.

labeled-duration does not conform to standard SQL, and is therefore supported by the Natural **SQL Extended Set** only.

8 Search Conditions

```
[NOT] { predicate
        (search-condition) }

search-condition { AND
                   OR } search-condition
```

Search Condition

A *search-condition* can consist of a simple *predicate* or multiple *search-conditions*. Multiple *search-conditions* are combined with the Boolean operators AND, OR and NOT, and can contain parentheses if required to indicate a desired order of evaluation.

Example

```
DEFINE DATA LOCAL
01 NAME   (A20)
01 AGE    (I2)
END-DEFINE
...
SELECT *
  INTO NAME, AGE
  FROM SQL-PERSONNEL
  WHERE AGE = 32 AND NAME > 'K'
END-SELECT
...
```

Predicate

```
scalar-expression comparison { scalar-expression
                               subquery }

scalar-expression [NOT] BETWEEN scalar-expression AND scalar-expression

column-reference [NOT] LIKE atom

column-reference IS [NOT] NULL

scalar-expression [NOT] IN { subquery
                             (atom, ...) }

scalar-expression comparison { ALL
                               ANY
                               SOME } subquery

EXISTS subquery
```

`XMLEXISTS (`*`xquery-expression-constant`*`{BY REF|PASSING`*`xquery-argument`*`,...})`

A *`predicate`* specifies a condition that can be “true”, “false” or “unknown”.

In a *`search-condition`*, a *`predicate`* can consist of a simple or complex comparison operation or other kinds of conditions.

Example:

```
SELECT NAME, AGE
  INTO VIEW PERS
  FROM SQL-PERSONNEL
  WHERE AGE BETWEEN 20 AND 30
    OR AGE IN ( 32, 34, 36 )
    AND NAME LIKE '%er'
    ...
```

Note: The percent sign (%) may conflict with Natural terminal commands. If so, you must define a terminal command control character different from `%`; see *Changing the Terminal Command Control Character* in the *Terminal Commands* documentation.

The individual predicates are explained in the following topics (for further information on predicates, please refer to the relevant literature). According to the syntax above, they are called as follows:

- Comparison Predicate
- BETWEEN Predicate
- LIKE Predicate
- NULL Predicate
- IN Predicate
- Quantified Predicate
- EXISTS Predicate
- XMLEXISTS Predicate

Comparison Predicate

`{`*`scalar-expression comparison scalar-expression`*`}`

A comparison predicate compares two values or a set of values with another set of values.

In the syntax diagram above, *`comparison`* can be one of the following operators:

=	equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
<>	not equal to

See information on *scalar-expression*.

Subquery

(*select-expression*)

A *subquery* is a *select-expression* that is nested inside another such expression.

Example:

```
DEFINE DATA LOCAL
1 #NAME    (A20)
1 #PERSNR  (I4)
END-DEFINE
...
SELECT NAME, PERSNR
  INTO #NAME, #PERSNR
  FROM SQL-PERSONNEL
  WHERE PERSNR IN
    ( SELECT PERSNR
        FROM SQL-AUTOMOBILES
        WHERE COLOR = 'black' )
        ...
END-SELECT
```

For further information, see ***Select Expressions***.

BETWEEN Predicate

scalar-expression [NOT] BETWEEN *scalar-expression* AND *scalar-expression*

A BETWEEN predicate compares a value with a range of values.

See information on *scalar-expression*.

LIKE Predicate

```
column-reference [NOT] LIKE atom
```

A `LIKE` predicate searches for strings that have a certain pattern.

See information on *column-reference* and *atom*.

NULL Predicate

```
                   { IS [NOT] NULL }
column-reference   { ISNULL        }
                   { NOTNULL       }
```

A `NULL` predicate tests for null values.

See information on *column-reference*.

IN Predicate

```
scalar-expression [NOT] IN  { subquery ... }
                            { (atom)       }
```

An `IN` predicate compares a value or a set of values with a collection of values.

See information on *scalar-expression* and *atom*.

See information on *subquery*.

Quantified Predicate

```
                                  { ALL  }
scalar-expression  comparison     { ANY  }  subquery
                                  { SOME }
```

A quantified predicate compares a value or a set of values with a collection of values.

See information on *scalar-expression*, *comparison* and *subquery*.

EXISTS Predicate

```
EXISTS subquery
```

An `EXISTS` predicate tests for the existence of certain rows.

The `EXISTS` predicate evaluates to true only if the result of evaluating the *`subquery`* is not empty; that is, if there exists at least one record (row) in the `FROM` table of the *`subquery`* satisfying the search condition of the `WHERE` clause of this *`subquery`*.

Example of `EXISTS`:

```
DEFINE DATA LOCAL
1 #NAME    (A20)
END-DEFINE
...
SELECT NAME
  INTO #NAME
  FROM SQL-PERSONNEL
  WHERE EXISTS
    ( SELECT *
      FROM SQL-EMPLOYEES
      WHERE PERSNR > 1000
        AND NAME < 'L' )
        ...
END-SELECT
...
```

See information on *`subquery`*.

XMLEXISTS Predicate

```
XMLEXISTS (xquery-expression-constant [ BY REF                       ] )
                                      [ PASSING xquery-argument,...  ]
```

`xquery-argument`

```
{ xquery-context-item-expression               }
{ xquery-context-item-expression AS identifier }
```

The `XMLEXISTS` predicate belongs to the Natural SQL Extended Set.

The `XMLEXISTS` predicate tests whether an `XPATH` expression returns a sequence of one or more items. For further information, see the IBM *Db2 XML Guide*.

9 Select Expressions

```
SELECT selection table-expression
```

A *select-expression* specifies a result table. It is used in the following Natural SQL statements: INSERT | SELECT | UPDATE

Selection

```
[ ALL      ]  { scalar-expression [[AS] correlation-name], ... }
[ DISTINCT ]  { *                                              }
```

A *selection* specifies the columns of the result set tables to be selected.

Syntax Element Description:

Syntax Element	Description
ALL|DISTINCT	**Elimination of Duplicate Rows:** Duplicate rows are not automatically eliminated from the result of a *select-expression*. To request this, specify the keyword DISTINCT. The alternative to DISTINCT is ALL. ALL is assumed if neither is specified.

scalar-expression	**Scalar Expression:**
	
	Instead of, or as well as, simple column names, a selection can also include general scalar expressions containing scalar operators and scalar functions which provide computed values (see also the section ***Scalar Expressions***).
	
	Example:
	
	`SELECT NAME, 65 - AGE`
	`  FROM SQL-PERSONNEL`
	`  ...`

AS	The optional keyword AS introduces a *correlation-name* for a column.
correlation-name	**Correlation Name:** A *correlation-name* can be assigned to a *scalar-expression* as an alias name for a result column. The *correlation-name* need not be unique. If no *correlation-name* is specified for a result column, the corresponding *column-name* will be used (if the result column is derived from a column name; if not, the result table will have no name). The name of a result column may be used, for example, as column name in the ORDER BY clause of a SELECT statement.
*	**Asterisk Notation:**

Syntax Element	Description

	All columns of the result table are selected.
	
	Example:
	
	`SELECT *`
	`  FROM SQL-PERSONNEL, SQL-AUTOMOBILES`
	`  ...`

Table Expression

from-clause [*where-clause*]
[*group-by-clause*] [*having-clause*]
[*order-by-clause*] [*fetch-first-clause*]

The *table-expression* specifies from where and according to what criteria rows are to be selected.

The following topics are covered below:

- FROM Clause
- Table Reference
- WHERE Clause
- GROUP BY Clause
- HAVING Clause
- ORDER BY Clause
- FETCH FIRST Clause
- Examples of Table Expressions

FROM Clause

FROM *table-reference*,...

This clause specifies from which tables the result set is built.

Table Reference

```
{ table-name [[AS] correlation-name] }
{ subquery [AS] correlation-name     }
{ joined-table                       }
```

The tables specified in the `FROM` clause must contain the column fields used in the selection list.

You can either specify a single table or produce an intermediate table resulting from a subquery or a "join" operation (see below).

Since various tables (that is, DDMs) can be addressed in one `FROM` clause and since a *table-expression* can contain several `FROM` clauses if *subqueries* are specified, the database ID (DBID) of the first DDM specified in the first `FROM` clause of the whole expression is used to identify the underlying database involved.

Optionally, a *correlation-clause* can be assigned to a *table-name*. For a *subquery*, a *correlation-clause* must be assigned.

Joined Table

```
{ table-reference [ { INNER         } ] JOIN table-reference ON join-condition }
{                 [ { LEFT [OUTER]  } ]                                        }
{                 [ { RIGHT [OUTER] } ]                                        }
{                 [ { FULL [OUTER]  } ]                                        }
{                                                                              }
{ (joined-table)                                                               }
```

A *joined-table* specifies an intermediate table resulting from a "join" operation.

The "join" can be an `INNER`, `LEFT OUTER`, `RIGHT OUTER` or `FULL OUTER JOIN`. If you do not specify anything, `INNER` applies.

Multiple "join" operations can be nested; that is, the tables which create the intermediate result table can themselves be intermediate result tables of a "join" operation or a *subquery*; and the latter, in turn, can also have a *joined-table* or another *subquery* in its `FROM` clause.

Join Condition

For `INNER`, `LEFT OUTER`, and `RIGHT OUTER` joins:

```
search-condition
```

For `FULL OUTER` joins:

```
full-join-expression = full-join-expression [AND ...]
```

Full Join Expression

```
{ column-name                                  }
{ { VALUE    }  (column-name , ...)            }
{ { COALESCE }                                 }
```

Within a *join-expression* only *column-names* and the *scalar-function* `VALUE` (or its synonym `COALESCE`) are allowed.

See details on *column-name*.

WHERE Clause

```
[WHERE search-condition]
```

The `WHERE` clause is used to specify the selection criteria (*search-condition*) for the rows to be selected.

Example:

```
DEFINE DATA LOCAL
01 NAME   (A20)
01 AGE    (I2)
END-DEFINE
...
SELECT *
  INTO NAME, AGE
  FROM SQL-PERSONNEL
  WHERE AGE = 32
END-SELECT
...
```

For further information, see ***Search Conditions***.

GROUP BY Clause

[GROUP BY *column-reference*,...]

The GROUP BY clause rearranges the table represented by the FROM clause into groups in a way that all rows within each group have the same value for the GROUP BY columns.

Each *column-reference* in the selection list must be either a GROUP BY column or specified within an *aggregate-function*. Aggregate functions are applied to the individual groups (not to the entire table). The result table contains as many rows as groups.

For further information, see ***Column Reference*** and ***Aggregate Function***.

Example:

```
DEFINE DATA LOCAL
1 #AGE     (I2)
1 #NUMBER  (I2)
END-DEFINE
...
SELECT AGE , COUNT(*)
  INTO #AGE, #NUMBER
  FROM SQL-PERSONNEL
  GROUP BY AGE
  ...
```

If the GROUP BY clause is preceded by a WHERE clause, all rows that do not satisfy the WHERE clause are excluded before any grouping is done.

HAVING Clause

[HAVING *search-condition*]

If the HAVING clause is specified, the GROUP BY clause should also be specified.

Just as the WHERE clause is used to exclude rows from a result table, the HAVING clause is used to exclude groups and therefore also based on a *search-condition*. Scalar expressions in a HAVING clause must be single-valued per group.

For further information, see ***Scalar Expressions*** and ***Search Conditions***.

Example:

```
DEFINE DATA LOCAL
1 #NAME     (A20)
1 #AVGAGE   (I2)
1 #NUMBER   (I2)
END-DEFINE
...
SELECT NAME, AVG(AGE), COUNT(*)
  INTO #NAME, #AVGAGE, #NUMBER
  FROM SQL-PERSONNEL
  GROUP BY NAME
  HAVING COUNT(*) > 1
  ...
```

ORDER BY Clause

```
         { sort-key [ ASC  ]    ,... }
ORDER BY {          [ DESC ]         }
         { INPUT SEQUENCE            }
         { ORDER OF table-designator }
```

sort-key

```
{ column-name         }
{ integer             }
{ sort-key-expression }
```

FETCH FIRST Clause

```
[ FETCH FIRST { 1       } { ROWS } ONLY ]
[             { integer } { ROW  }      ]
```

Examples of Table Expressions

Example 1:

```
DEFINE DATA LOCAL
01 #NAME      (A20)
01 #FIRSTNAME (A15)
01 #AGE       (I2)
...
END-DEFINE
...
SELECT NAME, FIRSTNAME, AGE
  INTO #NAME, #FIRSTNAME, #AGE
```

```
  FROM SQL-PERSONNEL
    WHERE NAME IS NOT NULL
      AND AGE > 20
...
  DISPLAY #NAME #FIRSTNAME #AGE
END-SELECT
...
END
```

Example 2:

```
DEFINE DATA LOCAL
01 #COUNT    (I4)
...
END-DEFINE
...
SELECT SINGLE COUNT(*) INTO #COUNT FROM SQL-PERSONNEL
...
```

10 Flexible SQL

The so-called “Flexible SQL”, which is a further possibility of issuing SQL statements, enables you to use arbitrary SQL syntax.

Using Flexible SQL

In addition to the SQL syntax described in the previous sections, flexible SQL enables you to use arbitrary SQL syntax.

Characters << and >>

Flexible SQL is enclosed in << and >> characters. It can include arbitrary SQL text and host variables. Within flexible SQL, host variables *must* be prefixed by a colon (:).

The flexible SQL string can cover several statement lines. Comments are possible, too (see also the statement `PROCESS SQL`).

Flexible SQL can be used as a replacement for any of the following syntactical SQL items:

- *atom*
- *column-reference*
- *scalar-expression*
- *predicate*

Flexible SQL can also be used between the clauses of a select expression:

```
SELECT selection
  << ... >>
  INTO ...
  FROM ...
  << ... >>
  WHERE ...
  << ... >>
  GROUP BY ...
  << ... >>
  HAVING ...
  << ... >>
  ORDER BY ...
  << ... >>
```

Note: The SQL text used in flexible SQL is not recognized by the Natural compiler. The SQL text (with replaced host variables) is simply copied into the SQL string passed to the database system. Syntax errors in flexible SQL are detected at runtime when the database executes the corresponding statement.

Example 1

```
SELECT NAME
FROM SQL-EMPLOYEES
WHERE << MONTH (BIRTH) >>  =  << MONTH (CURRENT_DATE) >>
```

Example 2:

```
SELECT NAME
FROM SQL-EMPLOYEES
WHERE << MONTH (BIRTH) = MONTH (CURRENT_DATE) >>
```

Example 3:

```
SELECT NAME
FROM SQL-EMPLOYEES
WHERE SALARY > 50000
<< INTERSECT
   SELECT NAME
   FROM SQL-EMPLOYEES
   WHERE DEPT = 'DEPT10'
>>
```

Specifying Text Variables in Flexible SQL

Within flexible SQL, you can also specify so-called "text variables".

```
<<:T:host-variable [LINDICATOR:host-variable]>>
```

The syntax items are described below:

:T:	A text variable is a *host-variable* prefixed by :T:. It must be in alphanumeric format. At runtime, a text variable within an SQL statement will be replaced by its contents that is, the text string contained in the text variable will be inserted into the SQL string. After the replacement, trailing blanks will be removed from the inserted text string. You have to make sure yourself that the content of a text variable results in a syntactically correct SQL string. In particular, the content of a text variable must not contain *host-variables*. A statement containing a text variable will always be executed in dynamic SQL mode.
LINDICATOR	**LINDICATOR Option:**

	The text variable can be followed by the keyword `LINDICATOR` and a length indicator variable (that is, a *host-variable* prefixed by colon). The length indicator variable has to be of format/length I2. If no `LINDICATOR` variable is specified, the entire content of the text variable will be inserted into the SQL string. If you specify a `LINDICATOR` variable, only the first *n* characters (*n* being the value of the `LINDICATOR` variable) of the text variable content will be inserted into the SQL string. If the number in the `LINDICATOR` variable is greater than the length of the text variable content, the entire text variable content will be inserted. If the number in the `LINDICATOR` variable is negative or 0, nothing will be inserted. See general information on *host-variable*.

Example Using Text Variable

```
DEFINE DATA LOCAL
   01 TEXTVAR (A200)
   01 TABLES VIEW OF SYSIBM-SYSTABLES
      02 NAME
      02 CREATOR
   END-DEFINE
   *
   MOVE 'WHERE NAME > ''SYS'' AND CREATOR = ''SYSIBM''' TO TEXTVAR
   *
   SELECT * INTO VIEW TABLES
     FROM SYSIBM-SYSTABLES
     << :T:TEXTVAR >>
     DISPLAY TABLES
   END-SELECT
   *
   END
```

The generated SQL statement will look as follows:

```
SELECT NAME, CREATOR FROM SYSIBM.SYSTABLES:T: FOR FETCH ONLY
```

The executed SQL statement will look as follows:

```
SELECT TABNAME, CREATOR FROM SYSIBM.SYSTABLES
  WHERE TABNAME > 'SYS' AND CREATOR = 'SYSIBM'
```

ROW CHANGE Expression with Flexible SQL

```
<<ROW CHANGE TOKEN FOR table-designator>>
```

A ROW CHANGE expression returns a token that represents the last change to a row.

TOKEN	Specifies a token of type BIGINT that represents a relative point in the modification sequence of a row.
FOR *table-designator*	Identifies the table in which the expression is referenced. *table-designator* has to be a valid Natural SQL DDM.

Example Using Row Change Expression with Flexible SQL:

```
DEFINE DATA LOCAL
   01 TEXTVAR (A200)
   01 TABLES VIEW OF SYSIBM-SYSTABLES
      02 NAME
      02 CREATOR
   END-DEFINE
   *
   SELECT << ROW CHANGE TOKEN FOR SYSTABLES >>
    INTO TEXTVAR
     FROM SYSIBM-SYSTABLES
     DISPLAY TEXTVAR
   END-SELECT
   *
END
```

OLAP Specification

```
{ ordered-OLAP-specification }
{ numbering-specification    }
{ aggregation-specification  }
```

ordered-OLAP-specification

```
   CUME_DIST ( )
   PERCENT_RANK ( )
 { RANK ( )           }  OVER
 { DENSE_RANK ( )     }  ([window-partition-clause]
 { NTILE (num-tile)   }  window-order-clause)
   lag-function
   lead-function
```

lag-function

```
LAG ( expression [ , offset [ , default [ , { 'RESPECT NULLS' } ] ] ] )
                                            { 'IGNORE NULLS'  }
```

lead-function

```
LEAD ( expression [ , offset [ , default [ , { 'RESPECT NULLS' } ] ] ] )
                                             { 'IGNORE NULLS'  }
```

numbering-specification

```
ROW_NUMBER ( ) OVER ([window-partition-clause] [window-order-clause])
```

aggregation-specification

```
{ aggregate-function  }        OVER ( [window-partition-clause])
{ OLAP-column-function }
{ RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING                   }
{                                RANGE BETWEEN UNBOUNDED                      }
{ window-order-clause   {        PRECEDING AND UNBOUNDED        }             }
{                                FOLLOWING                                    }
{                                window-aggregation-group-clause              }
```

aggregate-function

```
⎧ AVG function         ⎫
⎪ CORRELATION function ⎪
⎪ COUNT function       ⎪
⎪ COUNT_BIG function   ⎪
⎨ COVARIANCE function  ⎬
⎪ MAX function         ⎪
⎪ MIN function         ⎪
⎪ STDDEV function      ⎪
⎪ SUM function         ⎪
⎩ VARIANCE function    ⎭
```

OLAP-column-function

```
⎧ first-value-function     ⎫
⎪ last-value-function      ⎪
⎨ nth-value-function       ⎬
⎩ ratio-to-report-function ⎭
```

first-value-function

```
FIRST_VALUE ( expression [ , { 'RESPECT NULLS' } ] )
                              { 'IGNORE NULLS'  }
```

last-value-function

```
LAST_VALUE ( expression [ , { 'RESPECT NULLS' } ] )
                             { 'IGNORE NULLS'  }
```

nth-value-function

```
NTH_VALUE ( expression , nth-row )
```

ratio-to-report-function

```
RATIO_TO_REPORT ( expression )
```

window-aggregation-group-clause

```
{ ROWS  } { group-start   }
{ RANGE } { group-between }
          { group-end     }
```

group-start

```
{ UNBOUNDED PRECEDING               }
{ unsigned-constant PRECEDING       }
{ CURRENT ROW                       }
```

group-between

```
BETWEEN group-bound-1 AND group-bound-2
```

group-bound-1

```
{ UNBOUNDED PRECEDING               }
{ unsigned-constant PRECEDING       }
{ unsigned-constant FOLLOWING       }
{ CURRENT ROW                       }
```

group-bound-2

```
{ UNBOUNDED FOLLOWING               }
{ unsigned-constant PRECEDING       }
{ unsigned-constant FOLLOWING       }
{ CURRENT ROW                       }
```

group-end

```
{ UNBOUNDED FOLLOWING               }
{ unsigned-constant FOLLOWING       }
```

window-partition-clause

```
PARTITION BY partitioning-expression,...
```

window-order-clause

```
                          [      ASC              ]
                          [   {  NULLS LAST      }]
ORDER BY                  [   {  ASC NULLS FIRST }]
{sort-key-expression      [   {  DESC            }]  },...
                          [   {  DESC NULLS FIRST}]
                          [      DESC NULLS LAST  ]
```

RANK	Specifies that the rank of a row is defined as 1 plus the number of rows that strictly precede the row.
DENSE_RANK	Specifies that the rank of a row is defined as 1 plus the number of preceding rows that are distinct with respect to the ordering.
ROW_NUMBER	Specifies that a sequential row number is computed for the row that is defined by the ordering, starting with 1 for the first row.
PARTITION BY	Defines the partition within which the OLAP operation is applied.
ORDER BY	Defines the ordering of rows within a partition that is used to determine the value of the OLAP specification.
ASC	Specifies that the values of *sort-key-expression* are used in ascending order.
DESC	Specifies that the values of *sort-key-expression* are used in descending order.
NULLS_FIRST	Specifies that the window ordering considers null values before all non-null values in the sort order.
NULLS LAST	Specifies that the window ordering considers null values after all non-null values in the sort order.

Example:

Display the ranking of employees that have a total salary of more than $30,000, in order by last name.

```
SELECT EMPNO, LASTNAME, FIRSTNME, SALARY+BONUS AS TOTAL_SALARY,
  <<RANK() OVER(ORDER BY SALARY+BONUS DESC) AS RANK_SALARY>>
  FROM DSN8910-EMP WHERE SALARY+BONUS > 30000
  ORDER BY LASTNAME;
```

Case Expression with Flexible SQL

case-expression

```
<< CASE { searched-when-clause        } [ ELSE { NULL              } ] END >>
        { ...                         }        { scalar-expression }
        { simple-when-clause          }
```

A *case-expression* does not conform to standard SQL and is therefore supported by the Natural **SQL Extended Set** only.

Searched WHEN Clause

```
WHEN search-condition THEN { NULL              }
                           { scalar-expression }
```

A Searched When Clause does not conform to standard SQL and is therefore supported by the Natural **SQL Extended Set** only.

See details on *search-condition*.

Simple WHEN Clause

```
scalar-expression { WHEN scalar-expression THEN { NULL              } } ...
                                                { scalar-expression }
```

A Simple WHEN Clause does not conform to standard SQL and is therefore supported by the Natural **SQL Extended Set** only.

Example:

```
DEFINE DATA LOCAL
1 VWA VIEW OF NAT-D0001
2 ID
2 NAME
2 CITY
01 #RES1 (A8)
01 #CASE ( I4) INIT<0>
END-DEFINE
SELECT  CITY,
<<
CASE SUBSTR(CITY,1,1)
WHEN 'V' THEN 'Administration'
WHEN 'D' THEN 'Accounting'
```

```
WHEN 'K' THEN 'Operations'
END
>>
INTO VWA.CITY , #RES1
FROM NAT-D0001
WRITE VWA.CITY #RES1
END-SELECT
END
```

Cast Expression with Flexible SQL

cast-expression

<<CAST (*scalar-expression* AS *data-type*) >>

A CAST expression does not conform to standard SQL and is therefore supported by the Natural **SQL Extended Set** only.

Example:

```
DEFINE DATA LOCAL
1 VWA VIEW OF NAT-D001
2 ID
2 NAME
2 CITY
01 #RES1 (I4)
END-DEFINE
SELECT
<< CAST (ID AS INTEGER)
>>
INTO #RES1
FROM NAT-D001 WHERE ID = 1
WRITE #RES1
END-SELECT
END
```

XML Functions with Flexible SQL

XML-Functions

Any available XML functions must be treated with flexible SQL if those functions have their own specific keyword or syntax, if you are using the AS keyword and order by statement or any specific statement recognized by SQL. You must place the symbol of the flexible SQL within that stated portion. Additionally, between the left parathesis and the left arrow symbol of flexible SQL, you must leave a space or you receive a compiler error.

Example:

```
DEFINE DATA LOCAL
1 D033412A VIEW OF NATQA-D033412A
2 NAME
2 YEARS_OF_SERVICE
2 ANNUAL_LEAVE
2 TIME_IN
2 BACKGROUND
END-DEFINE
SELECT XMLSERIALIZE( <<CONTENT XMLELEMENT>>( <<NAME "Annual Leave">>,XMLATTRIBUTES( ↩
<<ANNUAL_LEAVE AS "al">>),XMLAGG(XMLELEMENT( <<NAME "name">>,NAME)<<ORDER BY NAME>>) ↩
)<<AS CLOB(110)>>) INTO #XMLSERIALIZE
FROM NATQA-D033412A
GROUP BY ANNUAL_LEAVE
END-SELECT
END
```

Scalar-Function and Column-Function (Aggregating) with Flexible SQL

`Scalar-functions` and *`column-functions`* are only supported with their proper syntax, as stated in the section ***Scalar Expression***. After the function name, within the left and right parentheses between the scalar expressions, there must be a comma. Therefore, not putting a comma between one scalar expression and another is restricted.

Any additional usage of keywords or any SQL statements within the parentheses, which is not recognized as a scalar expression with or without a comma, must be included with the flexible SQL to make it work.

Additionally, between the left parathesis and the left arrow symbol of flexible SQL you must leave a space or you receive a compiler error.

Example:

```
DEFINE DATA LOCAL
01 V1 VIEW OF DSN8910-EMP
02 EMPNO
02 FIRSTNME
02 LASTNAME
02 SALARY
02 BONUS
01 M1  (I4)
END-DEFINE
M1 := 10000
```

```
SELECT * INTO VIEW  V1
FROM DSN8910-EMP
WHERE SALARY > GREATEST(CAST( <<:M1 AS INTEGER>> ))
DISPLAY V1
END-SELECT
ENDEND
```

III Referenced Example Programs

11 Referenced Example Programs

This chapter contains additional example programs that are referenced in the Natural statements and system variables reference documentation. All these examples are contained in the library SYSEXSYN.

Note: Generally, the example programs shown in the statement descriptions are written in structured mode. For statements where the reporting-mode syntax differs considerably from the structured-mode syntax, references to equivalent reporting-mode examples are also provided. The example programs are available in source-code form in the Natural library SYSEXSYN. Further example programs of using Natural statements are documented in the section *Referenced Example Programs* in the *Programming Guide*. These example programs are provided in the Natural library SYSEXPG. Ask your Natural administrator about the availability of these libraries at your site. The example programs use data from the files EMPLOYEES and VEHICLES, which are supplied by Software AG for demonstration purposes.

ASSIGN

The following example is referenced in the ASSIGN/COMPUTE statement description:

ASGEX1R - ASSIGN (reporting mode)

```
** Example 'ASGEX1R': ASSIGN (reporting mode)
************************************************************************
RESET #A (N3)
      #B (A6)
      #C (N0.3)
      #D (N0.5)
      #E (N1.3)
      #F (N5)
      #G (A25)
      #H (A3/1:3)
*
#A = 5                                 WRITE NOTITLE '=' #A
#B = 'ABC'                             WRITE '=' #B
#C = .45                               WRITE '=' #C
#D = #E = -0.12345                     WRITE '=' #D / '=' #E
ASSIGN ROUNDED #F = 199.999            WRITE '=' #F
#G = 'HELLO'                           WRITE '=' #G
*
#H (1) = 'UVW'
#H (3) = 'XYZ'                         WRITE '=' #H (1:3)
*
END
```

Output of Program AEDEX1R:

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

AT BREAK

The following examples are referenced in the `AT BREAK` statement description:

ATBEX1R - AT BREAK (reporting mode)

```
** Example 'ATBEX1R': AT BREAK (reporting mode)
************************************************************************
*
LIMIT 10
READ EMPLOYEES BY CITY
  AT BREAK OF CITY DO
    SKIP 1
  DOEND
  /*
  DISPLAY NOTITLE CITY (IS=ON) COUNTRY (IS=ON) NAME
LOOP
END
```

Output of Program `ATBEX1R`:

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


AIKEN                 USA     SENKO

AIX EN OTHE           F       GODEFROY

AJACCIO                       CANALE

ALBERTSLUND           DK      PLOUG

ALBUQUERQUE           USA     HAMMOND
                              ROLLING
                              FREEMAN
                              LINCOLN

ALFRETON              UK      GOLDBERG
```

```
ALICANTE            E       GOMEZ
```

ATBEX5R - AT BREAK statement with multiple break levels (reporting mode)

```
** Example 'ATBEX5R': AT BREAK (multiple break levels) (reporting mode)
************************************************************************
RESET LEAVE-DUE-L (N4)
*
LIMIT 5
FIND EMPLOYEES WITH CITY = 'PHILADELPHIA' OR = 'PITTSBURGH'
               SORTED BY CITY DEPT
  MOVE LEAVE-DUE TO LEAVE-DUE-L
  DISPLAY CITY (IS=ON) DEPT (IS=ON) NAME LEAVE-DUE-L
  AT BREAK OF DEPT
    WRITE NOTITLE /
          T*DEPT OLD(DEPT) T*LEAVE-DUE-L SUM(LEAVE-DUE-L) /
  AT BREAK OF CITY
    WRITE NOTITLE
          T*CITY OLD(CITY) T*LEAVE-DUE-L SUM(LEAVE-DUE-L) //
LOOP
*
END
```

Output of Program `ATBEX5R`:

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

PHILADELPHIA         MGMT30     WOLF-TERROINE           11
                                MACKARNESS              27

                     MGMT30                             38

                     TECH10     BUSH                    39
                                NETTLEFOLDS             24

                     TECH10                             63

PHILADELPHIA                                           101


PITTSBURGH           MGMT10     FLETCHER                34

                     MGMT10                             34

PITTSBURGH                                              34
```

AT END OF DATA

The following example is referenced in the AT END OF DATA statement description:

AEDEX1R - AT END OF DATA (reporting mode)

```
** Example 'AEDEX1R': AT END OF DATA (reporting mode)
************************************************************************
LIMIT 5
EMP. FIND EMPLOYEES WITH CITY = 'STUTTGART'
  IF NO RECORDS FOUND
    ENTER
  DISPLAY PERSONNEL-ID NAME FIRST-NAME
          SALARY (1) CURR-CODE (1)
  /*
  AT END OF DATA DO
    IF *COUNTER (EMP.) = 0 DO
      WRITE 'NO RECORDS FOUND'
      ESCAPE BOTTOM
    DOEND
    WRITE NOTITLE / 'SALARY STATISTICS:'
                  / 7X 'MAXIMUM:' MAX(SALARY(1))  CURR-CODE (1)
                  / 7X 'MINIMUM:' MIN(SALARY(1))  CURR-CODE (1)
                  / 7X 'AVERAGE:' AVER(SALARY(1)) CURR-CODE (1)
  DOEND
LOOP
END
```

Output of Program AEDEX1R:

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

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

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

AT END OF PAGE

The following example is referenced in the `AT END OF PAGE` statement description:

AEPEX1R - AT END OF PAGE (reporting mode)

```
** Example 'AEPEX1R': AT END OF PAGE (reporting mode)
************************************************************************
FORMAT PS=10
LIMIT 10
READ EMPLOYEES BY PERSONNEL-ID FROM '20017000'
  DISPLAY NOTITLE GIVE SYSTEM FUNCTIONS
          NAME JOB-TITLE 'SALARY' SALARY(1) CURR-CODE (1)
  /*
  AT END OF PAGE DO
     WRITE / 28T 'AVERAGE SALARY: ...' AVER(SALARY(1)) CURR-CODE (1)
  DOEND
  /*
LOOP
END
```

Output of Program `AEPEX1R`:

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


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

                           AVERAGE SALARY: ...      33533 USD
```

AT START OF DATA

The following example is referenced in the `AT START OF DATA` statement description:

ASDEX1R - AT START OF DATA (reporting mode)

```
** Example 'ASDEX1R': AT START OF DATA (reporting mode)
************************************************************************
RESET #CITY (A20) #CNTL (A1)
*
REPEAT
  INPUT 'ENTER VALUE FOR CITY' #CITY
  /*
  IF #CITY = ' ' OR= 'END' DO
    STOP
  DOEND
  FIND EMPLOYEES WITH CITY = #CITY
    IF NO RECORDS FOUND DO
      WRITE NOTITLE NOHDR 'NO RECORDS FOUND'
      ESCAPE
    DOEND
    /*
    AT START OF DATA DO
      INPUT (AD=O) 'RECORDS FOUND' *NUMBER //
                   'ENTER ''D'' TO DISPLAY RECORDS' #CNTL (AD=A)
      IF #CNTL NE 'D' DO
        ESCAPE BOTTOM
      DOEND
    DOEND
    /*
    DISPLAY NAME FIRST-NAME
  LOOP
LOOP
END
```

Output of Program `ASDEX1R`:

```
ENTER VALUE FOR CITY PARIS
```

After entering and confirming city name:

```

RECORDS FOUND        26

ENTER 'D' TO DISPLAY RECORDS D
```

After entering and confirming `D`:

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

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

AT TOP OF PAGE

The following example is referenced in the `AT TOP OF PAGE` statement description:

ATPEX1R - AT TOP OF PAGE (reporting mode)

```
** Example 'ATPEX1R': AT TOP OF PAGE (reporting mode)
************************************************************************
*
FORMAT PS=15
LIMIT 15
*
READ EMPLOYEES BY NAME STARTING FROM 'L'
  DISPLAY 2X NAME 4X FIRST-NAME CITY DEPT
  WRITE TITLE UNDERLINED 'EMPLOYEE REPORT'
  WRITE TRAILER '-' (78)
  /*
  AT TOP OF PAGE DO
    WRITE 'BEGINNING NAME:' NAME
  DOEND
  /*
  AT END OF PAGE DO
    SKIP 1
    WRITE 'ENDING NAME:   ' NAME
```

```
  DOEND
LOOP
END
```

DEFINE SUBROUTINE

The following example is referenced in the `DEFINE SUBROUTINE` statement description:

DSREX1R - DEFINE SUBROUTINE (reporting mode)

```
** Example 'DSREX1R': DEFINE SUBROUTINE (reporting mode)
************************************************************************
RESET #ARRAY-ALL (A300)
      #X (N2) #Y (N2)
REDEFINE #ARRAY-ALL (#ARRAY (A75/1:4))
         #ARRAY-ALL (#ALINE (A25/1:4,1:3))
*
FORMAT PS=20
LIMIT 5
*
MOVE 1 TO #X #Y
*
FIND EMPLOYEES WITH NAME = 'SMITH'
  OBTAIN ADDRESS-LINE (1:2)
  /*
  MOVE NAME            TO #ALINE (#X,#Y)
  MOVE ADDRESS-LINE(1) TO #ALINE (#X+1,#Y)
  MOVE ADDRESS-LINE(2) TO #ALINE (#X+2,#Y)
  MOVE PHONE           TO #ALINE (#X+3,#Y)
  IF #Y = 3 DO
    MOVE 1 TO #Y
    PERFORM PRINT
  DOEND
  ELSE DO
    ADD 1 TO #Y
  DOEND
  AT END OF DATA DO
    PERFORM PRINT
  DOEND
LOOP
*
DEFINE SUBROUTINE PRINT
  WRITE NOTITLE (AD=OI) #ARRAY(*)
  RESET #ARRAY(*)
  SKIP 1
RETURN
*
END
```

Output of Program AEDEX1R:

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

SMITH                    SMITH
5 HAWTHORN               13002 NEW ARDEN COUR
OAK BROOK                SILVER SPRING
150-9351                 639-8963
```

FIND

The following examples are referenced in the FIND statement description:

FNDFIR - FIND statement with FIRST option (reporting mode)

```
** Example 'FNDFIR': FIND FIRST
************************************************************************
*
FIND FIRST EMPLOYEES WITH CITY = 'DERBY'
*
WRITE NOTITLE 'TOTAL RECORDS SELECTED:' *NUMBER
SKIP 2
WRITE '***FIRST PERSON SELECTED***' //
      'NAME:       ' NAME /
      'DEPARTMENT:' DEPT /
      'JOB TITLE: ' JOB-TITLE
*
END
```

Output of Program FNDFIR:

```
TOTAL RECORDS SELECTED:         141


***FIRST PERSON SELECTED***

NAME:       DEAKIN
DEPARTMENT: SALE01
JOB TITLE:  SALES ACCOUNTANT
```

FNDNUM - FIND statement with NUMBER option (reporting mode)

```
** Example 'FNDNUM': FIND NUMBER
************************************************************************
RESET #BIRTH (D)
*
MOVE EDITED '19500101' TO #BIRTH (EM=YYYYMMDD)
*
FIND NUMBER EMPLOYEES WITH CITY = 'MADRID'
                      WHERE BIRTH LT #BIRTH
*
WRITE NOTITLE 'TOTAL RECORDS SELECTED:       ' *NUMBER
            / 'TOTAL BORN BEFORE 1 JAN 1950: ' *COUNTER
*
END
```

Output of Program FNDNUM:

```
TOTAL RECORDS SELECTED:                 41
TOTAL BORN BEFORE 1 JAN 1950:           16
```

FNDUNQ - FIND statement with UNIQUE option (reporting mode)

```
** Example 'FNDUNQ': FIND UNIQUE
************************************************************************
RESET #NAME (A20)
*
*
INPUT 'ENTER EMPLOYEE NAME: ' #NAME
IF #NAME = ' '
  STOP
*
FIND UNIQUE EMPLOYEES WITH NAME = #NAME
*
DISPLAY NOTITLE NAME FIRST-NAME JOB-TITLE
*
ON ERROR DO
  WRITE 'NAME EITHER NOT UNIQUE OR DOES NOT EXIST'
  FETCH 'FNDUNQ'
DOEND
*
END
```

Output of Program FNDUNQ:

```
ENTER EMPLOYEE NAME: HEURTEBISE
```

After entering and confirming name HEURTEBISE:

```
        NAME              FIRST-NAME              CURRENT
                                                  POSITION
-------------------- -------------------- -------------------------

HEURTEBISE           MICHEL               CONTROLEUR DE GESTION
```

FOR

The following example is referenced in the FOR statement description:

FOREX1R - FOR (reporting mode)

```
** Example 'FOREX1R': FOR (reporting mode)
************************************************************************
RESET #INDEX (I1)
      #ROOT  (N2.7)
*
FOR #INDEX 1 TO 5
  COMPUTE #ROOT = SQRT (#INDEX)
  WRITE NOTITLE '=' #INDEX 3X '=' #ROOT
LOOP
*
SKIP 1
FOR #INDEX 1 TO 5 STEP 2
  COMPUTE #ROOT = SQRT (#INDEX)
  WRITE '=' #INDEX 3X '=' #ROOT
LOOP
*
END
```

Output of Program FOREX1R:

```
#INDEX:    1   #ROOT:   1.0000000
#INDEX:    2   #ROOT:   1.4142135
#INDEX:    3   #ROOT:   1.7320508
#INDEX:    4   #ROOT:   2.0000000
#INDEX:    5   #ROOT:   2.2360679

#INDEX:    1   #ROOT:   1.0000000
#INDEX:    3   #ROOT:   1.7320508
#INDEX:    5   #ROOT:   2.2360679
```

HISTOGRAM

The following example is referenced in the HISTOGRAM statement description:

HSTEX1R - HISTOGRAM (reporting mode)

```
** Example 'HSTEX1R': HISTOGRAM (reporting mode)
************************************************************************
*
LIMIT 8
HISTOGRAM EMPLOYEES CITY STARTING FROM 'M'
 DISPLAY NOTITLE CITY
         'NUMBER OF/PERSONS' *NUMBER *COUNTER
LOOP
*
END
```

Output of Program HSTEX1R:

```
        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
```

IF

The following example is referenced in the IF statement description:

IFEX1R - IF (reporting mode)

```
** Example 'IFEX1R': IF (reporting mode)
************************************************************************
RESET #BIRTH (D)
*
MOVE EDITED '19450101' TO #BIRTH (EM=YYYYMMDD)
SUSPEND IDENTICAL SUPPRESS
LIMIT 20
*
FND. FIND EMPLOYEES WITH CITY = 'FRANKFURT'
                    SORTED BY NAME BIRTH
  IF SALARY (1) LT 40000
    WRITE NOTITLE '*****' NAME 30X 'SALARY LT 40000'
  ELSE DO
    IF BIRTH GT #BIRTH DO
      FIND VEHICLES WITH PERSONNEL-ID = PERSONNEL-ID (FND.)
        DISPLAY (IS=ON) NAME BIRTH (EM=YYYY-MM-DD)
                        SALARY (1) MAKE (AL=8)
      LOOP
    DOEND
  DOEND
LOOP
END
```

Output of Program `IFEX1R`:

```
        NAME             DATE      ANNUAL     MAKE
                          OF       SALARY
                         BIRTH
-------------------- ---------- ---------- --------

BAECKER              1956-01-05      74400 BMW
***** BECKER                                            SALARY LT 40000
BLOEMER              1979-11-07      45200 FIAT
FALTER               1954-05-23      70800 FORD
***** FALTER                                            SALARY LT 40000
***** GROTHE                                            SALARY LT 40000
***** HEILBROCK                                         SALARY LT 40000
***** HESCHMANN                                         SALARY LT 40000
HUCH                 1952-09-12      67200 MERCEDES
***** KICKSTEIN                                         SALARY LT 40000
***** KLEENE                                            SALARY LT 40000
***** KRAMER                                            SALARY LT 40000
```

PERFORM BREAK PROCESSING

The following example is referenced in the `PERFORM BREAK PROCESSING` statement description:

PBPEX1R - PERFORM BREAK PROCESSING (reporting mode)

```
** Example 'PBPEX1R': PERFORM BREAK PROCESSING (reporting mode)
************************************************************************
RESET #LINE (N2) #INDEX (N2)
*
MOVE 1 TO #LINE
FOR #INDEX 1 TO 18
  PERFORM BREAK PROCESSING
  /*
  AT BREAK OF #INDEX /1/ DO
    WRITE NOTITLE / 'PLEASE COMPLETE LINES 1-9 ABOVE' /
    MOVE 1 TO #LINE
  DOEND
  /*
  WRITE NOTITLE '_' (64) '=' #LINE
  ADD 1 TO #LINE
LOOP
END
```

Output of Program `PBPEX1R`:

```
________________________________________________________________ #LINE:   1
________________________________________________________________ #LINE:   2
________________________________________________________________ #LINE:   3
________________________________________________________________ #LINE:   4
________________________________________________________________ #LINE:   5
________________________________________________________________ #LINE:   6
________________________________________________________________ #LINE:   7
________________________________________________________________ #LINE:   8
________________________________________________________________ #LINE:   9

PLEASE COMPLETE LINES 1-9 ABOVE

________________________________________________________________ #LINE:   1
________________________________________________________________ #LINE:   2
________________________________________________________________ #LINE:   3
________________________________________________________________ #LINE:   4
________________________________________________________________ #LINE:   5
________________________________________________________________ #LINE:   6
________________________________________________________________ #LINE:   7
________________________________________________________________ #LINE:   8
________________________________________________________________ #LINE:   9

PLEASE COMPLETE LINES 1-9 ABOVE
```

READ

The following example is referenced in the `READ` statement description:

REAEX1R - READ (reporting mode)

```
** Example 'REAEX1R': READ (reporting mode)
************************************************************************
LIMIT 3
*
WRITE 'READ IN PHYSICAL SEQUENCE'
READ EMPLOYEES IN PHYSICAL SEQUENCE
  DISPLAY NOTITLE PERSONNEL-ID NAME *ISN *COUNTER
LOOP
*
WRITE / 'READ IN ISN SEQUENCE'
READ EMPLOYEES BY ISN STARTING FROM 1 ENDING AT 3
  DISPLAY PERSONNEL-ID NAME *ISN *COUNTER
LOOP
*
WRITE / 'READ IN NAME SEQUENCE'
READ EMPLOYEES BY NAME
  DISPLAY PERSONNEL-ID NAME *ISN *COUNTER
LOOP
*
WRITE / 'READ IN NAME SEQUENCE STARTING FROM ''M'''
READ EMPLOYEES BY NAME STARTING FROM 'M'
  DISPLAY PERSONNEL-ID NAME *ISN *COUNTER
LOOP
*
END
```

Output of Program `REAEX1R`:

```
PERSONNEL         NAME            ISN         CNT
   ID
--------- -------------------- ----------- -----------

READ IN PHYSICAL SEQUENCE
50005800  ADAM                           1           1
50005600  MORENO                         2           2
50005500  BLOND                          3           3

READ IN ISN SEQUENCE
50005800  ADAM                           1           1
50005600  MORENO                         2           2
50005500  BLOND                          3           3

READ IN NAME SEQUENCE
```

```
60008339  ABELLAN                         478            1
30000231  ACHIESON                        878            2
50005800  ADAM                              1            3

READ IN NAME SEQUENCE STARTING FROM 'M'
30008125  MACDONALD                       923            1
20028700  MACKARNESS                      765            2
40000045  MADSEN                          508            3
```

REPEAT

The following examples are referenced in the REPEAT statement description:

RPTEX1R - REPEAT (reporting mode)

```
** Example 'RPTEX1R': REPEAT (reporting mode)
************************************************************************
RESET #PERS-NR (A8)
*
REPEAT
  INPUT 'ENTER A PERSONNEL NUMBER:' #PERS-NR
  IF #PERS-NR = ' '
    ESCAPE BOTTOM
  FIND EMPLOYEES WITH PERSONNEL-ID = #PERS-NR
    IF NO RECORD FOUND
      REINPUT 'NO RECORD FOUND'
    DISPLAY NOTITLE NAME
  LOOP
LOOP
*
END
```

Output of Program RPTEX1R:

```
ENTER A PERSONNEL NUMBER:
```

RPTEX2R - REPEAT with WHILE and UNTIL option (reporting mode)

```
** Example 'RPTEX2R': REPEAT (with WHILE and UNTIL option)
************************************************************************
RESET #X (I1) #Y (I1)
*
*
REPEAT WHILE #X <= 5
  ADD 1 TO #X
  WRITE NOTITLE '=' #X
LOOP
*
```

```
SKIP 3
REPEAT
  ADD 1 TO #Y
  WRITE '=' #Y
  UNTIL #Y = 6
LOOP
*
END
```

Output of Program RPTEX2R:

```
#X:     1
#X:     2
#X:     3
#X:     4
#X:     5
#X:     6



#Y:     1
#Y:     2
#Y:     3
#Y:     4
#Y:     5
#Y:     6
```

SORT

The following example is referenced in the SORT statement description:

SRTEX1R - SORT (reporting mode)

```
** Example 'SRTEX1R': SORT (reporting mode)
************************************************************************
RESET #AVG (P11) #TOTAL-TOTAL (P11) #TOTAL-SALARY (P11)
      #AVER-PERCENT (N3.2)
*
LIMIT 3
FIND EMPLOYEES WITH CITY = 'BOSTON'
  OBTAIN SALARY(1:2)
  COMPUTE #TOTAL-SALARY = SALARY (1) + SALARY (2)
  ACCEPT IF #TOTAL-SALARY GT 0
  /*
  SORT BY PERSONNEL-ID USING #TOTAL-SALARY SALARY(*) CURR-CODE
       GIVE AVER(#TOTAL-SALARY)
  /*
  AT START OF DATA DO
```

```
    WRITE NOTITLE '*' (40)
          'AVG CUMULATIVE SALARY:' *AVER (#TOTAL-SALARY) /
    MOVE *AVER (#TOTAL-SALARY) TO #AVG
  DOEND
  COMPUTE ROUNDED #AVER-PERCENT = #TOTAL-SALARY / #AVG * 100
  ADD #TOTAL-SALARY TO #TOTAL-TOTAL
  /*
  DISPLAY NOTITLE PERSONNEL-ID SALARY (1) SALARY (2)
          #TOTAL-SALARY CURR-CODE (1)
          'PERCENT/OF/AVER' #AVER-PERCENT
  AT END OF DATA
    WRITE / '*' (40) 'TOTAL SALARIES PAID: ' #TOTAL-TOTAL
LOOP
*
END
```

Output of Program SRTEX1R:

```
PERSONNEL   ANNUAL     ANNUAL   #TOTAL-SALARY CURRENCY PERCENT
   ID       SALARY     SALARY                   CODE     OF
                                                        AVER
--------- ---------- ---------- ------------- -------- -------

**************************************** AVG CUMULATIVE SALARY:        44633

20000100       31000      29400        60400  USD       135.30
20019200       18000      17100        35100  USD        78.60
20020400       20000      18400        38400  USD        86.00

**************************************** TOTAL SALARIES PAID:        133900
```

STORE

The following example is referenced in the STORE statement description:

STOEX1R - STORE (reporting mode)

```
** Example 'STOEX1R': STORE  (reporting mode)
**
** CAUTION: Executing this example will modify the database records!
************************************************************************
RESET #PERSONNEL-ID (A8)
      #NAME         (A20)
      #FIRST-NAME   (A15)
      #BIRTH-D      (D)
      #MAR-STAT     (A1)
      #BIRTH        (A8)
      #CITY         (A20)
```

```
      #COUNTRY      (A3)
      #CONF         (A1)
*
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
  IF #NAME = ' '
    REINPUT WITH TEXT 'ENTER A LAST-NAME' MARK 2 AND SOUND ALARM
  IF #FIRST-NAME = ' '
    REINPUT WITH TEXT 'ENTER A FIRST-NAME' MARK 3 AND SOUND ALARM
  /*
  /*  ENSURE PERSON IS NOT ALREADY ON FILE
  /*
  FIND NUMBER EMPLOYEES WITH PERSONNEL-ID =  #PERSONNEL-ID
  IF *NUMBER > 0
    REINPUT 'PERSON WITH SAME PERSONNEL-ID ALREADY EXISTS'
            MARK 1 AND SOUND ALARM
  MOVE 'N' TO #CONF
  /*
  /*  GET FURTHER INFORMATION
  /*
  INPUT
    'ADDITIONAL PERSONNEL DATA'                        ////
    'PERSONNEL-ID             :' #PERSONNEL-ID (AD=IO) /
    'NAME                     :' #NAME         (AD=IO) /
    'FIRST-NAME               :' #FIRST-NAME   (AD=IO) ///
    'MARITAL STATUS           :' #MAR-STAT             /
    'DATE OF BIRTH (YYYYMMDD) :' #BIRTH                /
    'CITY                     :' #CITY                 /
    'COUNTRY (3 CHARACTERS)   :' #COUNTRY              //
    'ADD THIS RECORD (Y/N)    :' #CONF          (AD=M)
  /*
  /*   ENSURE REQUIRED FIELDS CONTAIN VALID DATA
  /*
  IF NOT (#MAR-STAT = 'S'  OR = 'M' OR = 'D' OR = 'W')
    REINPUT TEXT 'ENTER VALID MARITAL STATUS  S=SINGLE ' -
                 'M=MARRIED D=DIVORCED W=WIDOWED' MARK 1
  IF NOT (#BIRTH = MASK(YYYYMMDD) AND #BIRTH = MASK(1582-2699))
    REINPUT TEXT 'ENTER CORRECT DATE' MARK 2
  IF #CITY  = ' '
    REINPUT TEXT 'ENTER A CITY NAME' MARK 3
  IF #COUNTRY = ' '
    REINPUT TEXT 'ENTER A COUNTRY CODE' MARK 4
  IF NOT (#CONF = 'N' OR= 'Y')
    REINPUT TEXT 'ENTER Y (YES) OR N (NO)' MARK 5
```

```
  IF #CONF = 'N'
    ESCAPE TOP
  /*
  /*  ADD THE RECORD
  /*
  MOVE EDITED #BIRTH TO #BIRTH-D (EM=YYYYMMDD)
  /*
  STORE RECORD IN EMPLOYEES
     WITH PERSONNEL-ID = #PERSONNEL-ID
          NAME         = #NAME
          FIRST-NAME   = #FIRST-NAME
          MAR-STAT     = #MAR-STAT
          BIRTH        = #BIRTH-D
          CITY         = #CITY
          COUNTRY      = #COUNTRY
  END OF TRANSACTION
  /*
  WRITE NOTITLE 'RECORD HAS BEEN ADDED'
  /*
LOOP
END
```

UPDATE

The following example is referenced in the UPDATE statement description:

UPDEX1R - UPDATE (reporting mode)

```
** Example 'UPDEX1R': UPDATE (reporting mode)
**
** CAUTION: Executing this example will modify the database records!
************************************************************************
RESET #NAME (A20)
*
INPUT 'ENTER A NAME:' #NAME (AD=M)
IF #NAME = ' '
  STOP
*
FIND EMPLOYEES WITH NAME = #NAME
  IF NO RECORDS FOUND
    REINPUT WITH 'NO RECORDS FOUND'  MARK 1
  /*
  INPUT 'NAME:       ' NAME (AD=O) /
        'FIRST NAME:' FIRST-NAME (AD=M) /
        'CITY:       ' CITY (AD=M)
  /*
  UPDATE USING SAME RECORD
  /*
  END TRANSACTION
```

```
  /*
LOOP
*
END
```

Output of Program `UPDEX1R`:

```
ENTER A NAME:
```

Example Programs for System Variables

The following examples are referenced in the `*OCCURRENCE` system variable description:

OCC1P - System Variable *OCCURRENCE

```
** Example 'OCC1P': *OCCURRENCE
************************************************************************
DEFINE DATA LOCAL
1 #N1 (N7/1:10)
1 #N2 (N7/1:10,1:10)
1 #N3 (N7/1:10,1:10,1:10)
END-DEFINE
*
CALLNAT 'OCC1N' #N1(*) #N2(1:2,1:4) #N3(1:6,1:7,1:8)
*
END
```

Subprogram `OCC1N` Called by Program `OCC1P`:

```
** Example 'OCC1N': *OCCURRENCE (called by OCC1P)
**********************************************************************
DEFINE DATA
PARAMETER
1 PARM1 (N7/1:V)
1 PARM2 (N7/1:V,1:V)
1 PARM3 (N7/1:V,1:V,1:V)
LOCAL
1 #OCC2 (I4/1:2)
1 #OCC3 (I4/1:3)
1 #OCC1 (I4)
END-DEFINE
*
MOVE *OCC(PARM1)   TO #OCC1
MOVE *OCC(PARM2,*) TO #OCC2(*)
MOVE *OCC(PARM3,*) TO #OCC3(*)
*
DISPLAY #OCC1 #OCC2(*) #OCC3(*)
DISPLAY *OCC(PARM1,*) *OCC(PARM2,*) *OCC(PARM3,*)
```

```
*
NEWPAGE
*
WRITE NOHDR
      'Occurrences of 1. parameter:' *OCC(PARM1)
    / 'Occurrences of 1. parameter:' *OCC(PARM1,1)
    / 'Occurrences of 1. parameter:' *OCC(PARM1,*)
    / 'Occurrences of 2. parameter:' *OCC(PARM2,1) *OCC(PARM2,2)
    / 'Occurrences of 2. parameter:' *OCC(PARM2,*)
    / 'Occurrences of 3. parameter:' *OCC(PARM3,1) *OCC(PARM3,2)
                                     *OCC(PARM3,3)
    / 'Occurrences of 3. parameter:' *OCC(PARM3,*)
*
END
```

Output of Program `OCC1P` - Page 1:

```
Page      1                                                   05-01-18  10:21:30

   #OCC1       #OCC2       #OCC3
----------- ----------- -----------

         10           2           6
                      4           7
                                  8
         10           2           6
                      4           7
                                  8
```

Output of Program `OCC1P` - Page 2:

```
Page      2                                                   05-01-18  10:21:30

Occurrences of 1. parameter:          10
Occurrences of 1. parameter:          10
Occurrences of 1. parameter:          10
Occurrences of 2. parameter:           2           4
Occurrences of 2. parameter:           2           4
Occurrences of 3. parameter:           6           7           8
Occurrences of 3. parameter:           6           7           8
```

OCC2P - System Variable *OCCURRENCE

```
** Example 'OCC2P': *OCCURRENCE
************************************************************************
DEFINE DATA LOCAL
1 #N (N7/1:10)
1 #I (I4)
END-DEFINE
*
FOR #I=1 TO 10
  MOVE #I TO #N(#I)
END-FOR
*
WRITE 'Passing occurrences 1:5'
CALLNAT 'OCC2N' #N(1:5)
*
WRITE 'Passing occurrences 5:10'
CALLNAT 'OCC2N' #N(5:10)
*
END
```

Subprogram `OCC2N` Called by Program `OCC2P`:

```
** Example 'OCC2N': *OCCURRENCE (called by OCC2P)
**********************************************************************
DEFINE DATA
PARAMETER
1 #ARR (N7/1:V)
LOCAL
1 I    (N7)
END-DEFINE
*
FOR I=1 TO *OCC(#ARR)
  DISPLAY #ARR(I)
END-FOR
*
END
```

Output of Program `OCC2P`:

```
Page      1                                                 05-01-18  10:33:03

Passing occurrences 1:5
       1
       2
       3
       4
       5
Passing occurrences 5:10
       5
       6
```

IV

12 ACCEPT/REJECT

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

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

Related Statements: AT BREAK | AT START OF DATA | AT END OF DATA | BACKOUT TRANSACTION | BEFORE BREAK PROCESSING | DELETE | END TRANSACTION | FIND | HISTOGRAM | GET | GET SAME | GET TRANSACTION DATA | LIMIT | PASSW | PERFORM BREAK PROCESSING | READ | RETRY | STORE | UPDATE

Belongs to Function Group: ***Database Access and Update***

Function

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

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

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

Syntax Description

Syntax Element	Description
IF	**IF Clause:** 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.
logical-condition	**Logical Condition Criterion:** 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.

Syntax Element	Description
	The fields used to specify the logical criterion may be database fields or user-defined variables. For additional information on logical conditions, see *Logical Condition Criteria* in the *Programming Guide*. **Note:** When ACCEPT/REJECT is used with a HISTOGRAM statement, only the database field specified in the HISTOGRAM statement may be used as a logical criterion.

Processing of Multiple ACCEPT/REJECT Statements

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

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

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

Limit Notation

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

Examples

Example 1 - ACCEPT

```
** Example 'ACREX1': ACCEPT
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 SEX
  2 MAR-STAT
END-DEFINE
*
LIMIT 50
READ EMPLOY-VIEW
  ACCEPT IF SEX='M' AND MAR-STAT = 'S'
  WRITE  NOTITLE '=' NAME '=' SEX 5X '=' MAR-STAT
END-READ
END
```

Output of Program ACREX1:

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

Example 2 - ACCEPT / REJECT

```
** Example 'ACREX2': ACCEPT/REJECT
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY    (1)
*
1 #PROC-COUNT (N8) INIT <0>
END-DEFINE
*
EMP. FIND EMPLOY-VIEW WITH NAME = 'JACKSON'
  WRITE NOTITLE *COUNTER NAME FIRST-NAME 'SALARY:' SALARY(1)
  /*
  ACCEPT IF SALARY (1) LT 50000
  WRITE *COUNTER 'ACCEPTED FOR FURTHER PROCESSING'
  /*
  REJECT IF SALARY (1) GT 30000
  WRITE *COUNTER 'NOT REJECTED'
  /*
  ADD 1 TO #PROC-COUNT
END-FIND
*
SKIP 2
WRITE NOTITLE 'TOTAL PERSONS FOUND ' *NUMBER (EMP.) /
              'TOTAL PERSONS SELECTED' #PROC-COUNT
END
```

Output of Program ACREX2:

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


TOTAL PERSONS FOUND            3
TOTAL PERSONS SELECTED         1
```

13 ADD

Related Statements: COMPRESS | COMPUTE | DIVIDE | EXAMINE | MOVE | MOVE ALL | MULTIPLY | RESET | SEPARATE | SUBTRACT

Belongs to Function Group: ***Arithmetic and Data Movement Operations***

Function

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

This statements has two different syntax structures.

Notes:

1. At the time the `ADD` statement is executed, each operand used in the arithmetic operation must contain a valid value.
2. For additions involving arrays, see also the section *Arithmetic Operations with Arrays*.
3. As for the formats of the operands, see also the section *Performance Considerations for Mixed Formats*.

Syntax 1 - ADD Statement without GIVING Clause

```
ADD [ROUNDED] { (arithmetic-expression) } ... TO operand2
              { operand1                }
```

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

Operand Definition Table (Syntax 1):

Operand	Possible Structure						Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A		N				N	P	I	F		D	T					yes	no
operand2		S	A		M				N	P	I	F		D	T					yes	yes

Syntax Element Description:

Syntax Element	Description:
arithmetic-expression	See ***Arithmetic Expression*** in the `COMPUTE` statement.
operand1 TO *operand2*	**Operands:** *operand1* and *operand2* are summands. The result is stored in *operand2* (result field). Hence, the statement is equivalent to: `operand2 := operand2 + operand1 + ...`
ROUNDED	**ROUNDED Option:** If the keyword `ROUNDED` is used, the result will be rounded. For information on rounding, see *Rules for Arithmetic Assignment, Field Truncation and Field Rounding* in the *Programming Guide*.

Example:

The statement

```
ADD #A(*) TO #B(*)  is equivalent to  COMPUTE #B(*) := #A(*) + #B(*)
ADD #S    TO #R     is equivalent to  COMPUTE #R    := #S + #R
ADD #S #T TO #R     is equivalent to  COMPUTE #R    := #S + #T + #R
ADD #A(*) TO #R     is equivalent to  COMPUTE #R    := #A(*) + #R
```

Syntax 2 - ADD Statement with GIVING Clause

```
ADD [ROUNDED] { (arithmetic-expression) } ... GIVING operand2
              { operand1                }
```

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

Operand Definition Table (Syntax 2):

Operand	Possible Structure						Possible Formats												Referencing Permitted	Dynamic Definition
operand1	C	S	A		N				N	P	I	F		D	T				yes	no
operand2		S	A		M		A	U	N	P	I	F	B*	D	T				yes	yes

* Format B of *operand2* may be used only with a length of less than or equal to 4.

Syntax Element Description:

Syntax Element	Description:
arithmetic-expression	See ***Arithmetic Expression*** in the COMPUTE statement.
operand1 GIVING *operand2*	**Operands:** *operand1* is a summand. *operand2* is only used to receive the result of the operation; it is *not included* in the addition. Hence, the statement is equivalent to: `operand2 := operand1 + ...`
ROUNDED	**ROUNDED Option:** If the keyword ROUNDED is used, the result will be rounded. For information on rounding, see *Rules for Arithmetic Assignment, Field Truncation and Field Rounding* in the *Programming Guide*.

Note: If Syntax 2 is used, the following applies: Only the (*operand1*) field(s) left of the keyword GIVING are the terms of the addition, the field right of the keyword GIVING (*operand2*) is just used to receive the result value. If just a single (*operand1*) field is supplied, the ADD operation turns into an assignment.

Example:

The statement

```
ADD #S        GIVING #R  is equivalent to  COMPUTE #R := #S

ADD #S #T     GIVING #R  is equivalent to  COMPUTE #R := #S + #T

ADD #A(*) 0   GIVING #R  is equivalent to  COMPUTE #R := #A(*) + 0
                         which is a legal operation, due to the rules defined
                         in Arithmetic Operations with Arrays

ADD #A(*)     GIVING #R  is equivalent to  COMPUTE #R := #A(*)
                         which is an illegal operation, due to the rules
                         defined in Assignment Operations with Arrays
```

Example

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

Output of Program ADDEX1:

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

ADD .231 3.6 GIVING #B              #B:  3.8

ADD ROUNDED 2.9 3.8 GIVING #C       #C:  7

CURRENT DATE: 2005-01-10            CURRENT DATE + 7: 2005-01-17

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

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

14 ASSIGN

See the statement `COMPUTE`.

15 AT BREAK

Structured Mode Syntax

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

Reporting Mode Syntax

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

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

Related Statements: ACCEPT/REJECT | AT START OF DATA | AT END OF DATA | BACKOUT TRANSACTION | BEFORE BREAK PROCESSING | DELETE | END TRANSACTION | FIND | GET | GET SAME | GET TRANSACTION DATA | HISTOGRAM | LIMIT | PASSW | PERFORM BREAK PROCESSING | READ | RETRY | STORE | UPDATE

Belongs to Function Group: ***Database Access and Update***

Function

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

The automatic break processing works as follows: Immediately after a record was read by the processing loop, the control field is checked. If a value change is detected in comparison to the previous record, the statements included in the AT BREAK statement block are executed. This does not apply to the very first record in the processing loop. In addition, when the processing loop is terminated (as reading of records is complete or due to an ESCAPE BOTTOM statement), a final execution of the statements in the AT BREAK statement block is triggered.

For further information, see *Automatic Break Processing* in the *Programming Guide*.

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

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

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

Natural system functions may be used in conjunction with an `AT BREAK` statement, see *Natural System Functions for Use in Processing Loops* in the *System Functions* documentation and *Example of System Functions with AT BREAK Statement* in the *Programming Guide*.

For further information, see also the section *AT BREAK Statement* in the *Programming Guide*. It covers topics such as:

- *Control Break Based on a Database Field*
- *Control Break Based on a User-Defined Variable*

Syntax Description

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:

Syntax Element	Description

`(r)`	**Reference Notation:**
	By default, the final `AT BREAK` condition (for loop termination) is always related to the outermost active processing loop initiated with a `FIND`, `READ`, `READ WORK FILE`, `HISTOGRAM` or `SORT` statement.
	
	With the notation `(r)` you can relate the final break condition of an `AT BREAK` statement to another specific currently open processing loop (that is, the loop in which the `AT BREAK` statement is located or any outer loop).
	
	Example:
	
	`...`
	`READ ...`
	`  FIND ...`
	`    FIND ...`
	`      AT BREAK ...`
	`        FIND ...`
	`        END-FIND`
	`      END-BREAK`
	`    END-FIND`
	`  END-FIND`
	`END-READ`
	`...`

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td></td><td>In this example, the final <code>AT BREAK</code> condition is related to the <code>READ</code> loop initiated in line 0120. It would be possible to have it related to one of the <code>FIND</code> loops initiated in line 0130 and 0140, but not to the one initiated in line 0160.

If <code>(r)</code> is specified for a break hierarchy, it must be specified with the first <code>AT BREAK</code> statement and applies also to all <code>AT BREAK</code> statements which follow.</td></tr>
<tr><td>operand1</td><td>Control Field:
The field used as the break control field is usually a database field. If a user-defined variable is used, it must be initialized prior to the evaluation of automatic break processing (see <code>BEFORE BREAK PROCESSING</code> statement). A specific occurrence of an array can also be used as a control field.</td></tr>
<tr><td>/n/</td><td>Notation /n/:
The notation /n/ may be used to indicate that only the first n positions (counting from left to right) of the control field are to be checked for a change in value. This notation can only be used with operands of format A, B, N or P.

A control break occurs when the value of the control field changes, or when all records in the processing loop for which the <code>AT BREAK</code> statement applies have been processed.</td></tr>
<tr><td>statement ...</td><td>Statement(s) to be Executed at Break Condition:

In structured mode, you must supply one or several suitable statements, depending on the situation. For an example of a statement, see Examples below.</td></tr>
<tr><td><code>END-BREAK</code></td><td rowspan="2">End of AT BREAK Statement:

In structured mode, the Natural reserved word <code>END-BREAK</code> must be used to end the <code>AT BREAK</code> statement.

In reporting mode, use the <code>DO ... DOEND</code> statements to supply one or several suitable statements, depending on the situation, and to end the <code>AT BREAK</code> statement. If you specify only a single statement, you can omit the <code>DO ... DOEND</code> statements. With respect to good coding practice, this is not recommended.</td></tr>
<tr><td>statement
DO statement
... DOEND</td></tr>
</table>

Multiple Break Levels

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

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

Example:

<table>
<tr><th>Structured Mode:</th><th>Reporting Mode:</th></tr>
<tr><td>

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

</td><td>

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

</td></tr>
</table>

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

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

See also ***Example 3*** below and the section *Multiple Control Break Levels* in the *Programming Guide*.

Examples

This section covers the following topics:

- Example 1 - AT BREAK
- Example 2 - AT BREAK Using /n/ Notation
- Example 3 - AT BREAK with Multiple Break Levels

For further examples of `AT BREAK`, see *Natural System Functions for Use in Processing Loops*, Examples ATBEX3 and ATBEX4.

Example 1 - AT BREAK

```
** Example 'ATBEX1S': AT BREAK (structured mode)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 COUNTRY
  2 NAME
END-DEFINE
*
LIMIT 10
READ EMPLOY-VIEW BY CITY
AT BREAK OF CITY
    SKIP 1
  END-BREAK
  DISPLAY NOTITLE CITY (IS=ON) COUNTRY (IS=ON) NAME
END-READ
*
END
```

Output of Program ATBEX1S:

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


AIKEN                USA     SENKO

AIX EN OTHE          F       GODEFROY

AJACCIO                      CANALE

ALBERTSLUND          DK      PLOUG

ALBUQUERQUE          USA     HAMMOND
                             ROLLING
                             FREEMAN
                             LINCOLN

ALFRETON             UK      GOLDBERG

ALICANTE             E       GOMEZ
```

Equivalent reporting-mode example: **ATBEX1R**.

Example 2 - AT BREAK Using /n/ Notation

```
** Example 'ATBEX2': AT BREAK (with /n/ notation)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 DEPT
  2 NAME
END-DEFINE
*
LIMIT 10
READ EMPLOY-VIEW BY DEPT STARTING FROM 'A'
AT BREAK OF DEPT /4/
    SKIP 1
  END-BREAK
  DISPLAY NOTITLE DEPT NAME
END-READ
*
END
```

Output of Program ATBEX2:

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

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

COMP01     HEURTEBISE
COMP01     TANCHOU
```

Example 3 - AT BREAK with Multiple Break Levels

```
** Example 'ATBEX5S': AT BREAK (multiple break levels) (structured mode)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 DEPT
  2 NAME
  2 LEAVE-DUE
1 #LEAVE-DUE-L (N4)
END-DEFINE
```

```
*
LIMIT 5
FIND EMPLOY-VIEW WITH CITY = 'PHILADELPHIA' OR = 'PITTSBURGH'
                 SORTED BY CITY DEPT
  MOVE LEAVE-DUE TO #LEAVE-DUE-L
  DISPLAY CITY (IS=ON) DEPT (IS=ON) NAME #LEAVE-DUE-L
  /*
 AT BREAK OF DEPT
    WRITE NOTITLE /
          T*DEPT OLD(DEPT) T*#LEAVE-DUE-L SUM(#LEAVE-DUE-L) /
  END-BREAK
  AT BREAK OF CITY
    WRITE NOTITLE
          T*CITY OLD(CITY) T*#LEAVE-DUE-L SUM(#LEAVE-DUE-L) //
  END-BREAK
END-FIND
*
END
```

Output of Program ATBEX5:

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


PHILADELPHIA         MGMT30     WOLF-TERROINE            11
                                MACKARNESS               27

                     MGMT30                              38

                     TECH10     BUSH                     39
                                NETTLEFOLDS              24

                     TECH10                              63

PHILADELPHIA                                            101


PITTSBURGH           MGMT10     FLETCHER                 34

                     MGMT10                              34

PITTSBURGH                                               34
```

Equivalent reporting-mode example: **ATBEX5R**.

16 AT END OF DATA

Structured Mode Syntax

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

Reporting Mode Syntax

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

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

Related Statements: ACCEPT/REJECT | AT BREAK | AT START OF DATA | BACKOUT TRANSACTION | BEFORE BREAK PROCESSING | DELETE | END TRANSACTION | FIND | GET | GET SAME | GET TRANSACTION DATA | HISTOGRAM | LIMIT | PASSW | PERFORM BREAK PROCESSING | READ | RETRY | STORE | UPDATE

Belongs to Function Group: ***Database Access and Update***

Function

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

This section covers the following topics:

- Processing
- Values of Database Fields
- Positioning
- System Functions

See also *AT START/END OF DATA Statements* in the *Programming Guide*.

Processing

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

Values of Database Fields

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

Positioning

This statement must be specified within the same program module which contains the loop creating statement.

System Functions

Natural system functions may be used in conjunction with an `AT END OF DATA` statement as described in *Using System Functions in Processing Loops* in the *System Functions* documentation.

Restrictions

- This statement can only be used in a processing loop that has been initiated with one of the following statements: `FIND`, `READ`, `READ WORK FILE`, `HISTOGRAM` or `SORT`.
- It may be used only once per processing loop.
- It is *not* evaluated if the processing loop referenced for `END OF DATA` processing is not entered.

Syntax Description

Syntax Element	Description
`(r)`	**Reference to a Specific Processing Loop:** An `AT END OF DATA` statement may be related to a specific active processing loop by using the notation `(r)`. If this notation is not used, the `AT END OF DATA` statement will be related to the outermost active database processing loop.
`statement ...`	**Statement(s) to be Executed at End of Data Condition:** In structured mode, you must supply one or several suitable statements, depending on the situation. For an example of a statement, see ***Example*** below.

Syntax Element	Description
`END-ENDDATA`	**End of AT END OF DATA Statement:**
statement ... `DO` *statement* `... DOEND`	In structured mode, the Natural reserved word `END-ENDDATA` must be used to end the `AT END OF DATA` statement. In reporting mode, use the `DO ... DOEND` statements to supply one or several suitable statements, depending on the situation, and to end the `AT END OF DATA` statement. If you specify only a single statement, you can omit the `DO ... DOEND` statements. With respect to good coding practice, this is not recommended.

Example

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

See also *Natural System Functions for Use in Processing Loops* in the *System Functions* documentation.

Output of Program AEDEX1S:

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


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

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

Equivalent reporting-mode example: **AEDEX1R**.

17 AT END OF PAGE

Structured Mode Syntax

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

Reporting Mode Syntax

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

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

Related Statements: AT TOP OF PAGE | CLOSE PRINTER | DEFINE PRINTER | DISPLAY | EJECT | FORMAT | NEWPAGE | PRINT | SKIP | SUSPEND IDENTICAL SUPPRESS | WRITE | WRITE TITLE | WRITE TRAILER

Belongs to Function Group: ***Creation of Output Reports***

Function

The AT END OF PAGE statement is used to specify processing that is to be performed when an end-of-page condition is detected (see session parameter PS in the *Parameter Reference*). An end-of-page condition may also occur as a result of a SKIP or NEWPAGE statement, but not as a result of an EJECT or INPUT statement.

See also the following sections in the *Programming Guide*:

- *Report Format and Control*
- *Report Specification - (rep) Notation*
- *Layout of an Output Page*
- *AT END OF PAGE Statement*

Processing

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

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

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

Logical Page Size

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

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

Last-Page Handling

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

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

System Functions

Natural system functions may be used in conjunction with an `AT END OF PAGE` statement as described in the section *Using System Functions in Processing Loops* in the *System Functions* documentation.

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

INPUT Statement with AT END OF PAGE

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

See also:

- *Split Screen Feature* of `INPUT` Statement
- *Example 2 - AT END OF PAGE with INPUT Statement*

Syntax Description

Syntax Element	Description
`(rep)`	**Report Specification:** The notation `(rep)` may be used to specify the identification of the report for which the `AT END OF PAGE` statement is applicable. A value in the range `0 - 31` or a logical name which has been assigned using the `DEFINE PRINTER` statement may be specified. If `(rep)` is not specified, the `AT END OF PAGE` statement will apply to the first report (Report 0). For information on how to control the format of an output report created with Natural, see *Report Format and Control* in the *Programming Guide*.
`statement`	**Statement(s) to be Executed at End of Page Condition:** In structured mode, you must supply one or several suitable statements, depending on the situation. For an example of a statement, see *Example* below.
`END-ENDPAGE`	**End of AT END OF PAGE Statement:** In structured mode, the Natural reserved word `END-ENDPAGE` must be used to end the `AT END OF PAGE` statement. In reporting mode, use the `DO ... DOEND` statements to supply one or several suitable statements, depending on the situation, and to end the `AT END OF PAGE` statement. If you specify only a single statement, you can omit the `DO ... DOEND` statements. With respect to good coding practice, this is not recommended.

`statement`
`DO statement ...`
`DOEND`	

Example

- Example 1 - AT END OF PAGE
- Example 2 - AT END OF PAGE with INPUT Statement

Example 1 - AT END OF PAGE

```
** Example 'AEPEX1S': AT END OF PAGE (structured mode)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 SALARY    (1)
  2 CURR-CODE (1)
END-DEFINE
*
FORMAT PS=10
LIMIT 10
READ EMPLOY-VIEW BY PERSONNEL-ID FROM '20017000'
  DISPLAY NOTITLE GIVE SYSTEM FUNCTIONS
          NAME JOB-TITLE 'SALARY' SALARY(1) CURR-CODE (1)
  /*

 AT END OF PAGE
   WRITE / 28T 'AVERAGE SALARY: ...' AVER(SALARY(1)) CURR-CODE (1)
  END-ENDPAGE


END-READ
*
END
```

See also *Natural System Functions for Use in Processing Loops*.

Output of Program AEPEX1S:

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

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

```
                         AVERAGE SALARY: ...      33533 USD
```

Equivalent reporting-mode example: **AEPEX1R**.

Example 2 - AT END OF PAGE with INPUT Statement

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

Output of Program AEPEX2S:

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


ABELLAN              KEPA                 28014      MADRID
ACHIESON             ROBERT               DE3 4TR    DERBY
ADAM                 SIMONE               89300      JOIGNY
ADKINSON             JEFF                 11201      BROOKLYN
ADKINSON             PHYLLIS              90211      BEVERLEY HILLS
```

```
ADKINSON             HAZEL                20760      GAITHERSBURG
ADKINSON             DAVID                27514      CHAPEL HILL
ADKINSON             CHARLIE              21730      LEXINGTON
ADKINSON             MARTHA               17010      FRAMINGHAM
ADKINSON             TIMMIE               17300      BEDFORD
ADKINSON             BOB                  66044      LAWRENCE
AECKERLE             SUSANNE              7000       STUTTGART
AFANASSIEV           PHILIP               39401      HATTIESBURG
AFANASSIEV           ROSE                 60201      EVANSTON
AHL                  FLEMMING             2300       SUNDBY

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

18 AT START OF DATA

Structured Mode Syntax

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

Reporting Mode Syntax

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

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

Related Statements: ACCEPT/REJECT | AT BREAK | AT END OF DATA | BACKOUT TRANSACTION | BEFORE BREAK PROCESSING | DELETE | END TRANSACTION | FIND | GET | GET SAME | GET TRANSACTION DATA | HISTOGRAM | LIMIT | PASSW | PERFORM BREAK PROCESSING | READ | RETRY | STORE | UPDATE

Belongs to Function Group: ***Database Access and Update***

Function

The statement AT START OF DATA is used to perform processing immediately after the first of a set of records is read for a processing loop that has been initiated by one of the following statements: READ, FIND, HISTOGRAM, SORT or READ WORK FILE.

See also *AT START/END OF DATA Statements* in the *Programming Guide*.

Processing

If the loop-initiating statement contains a WHERE clause, the at-start-of-data condition will be true when the first record is read which meets both the basic search and the WHERE criteria.

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

Value of Database Fields

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

Positioning

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

Syntax Description

Syntax Element	Description
`(r)`	**Reference to a Specific Processing Loop:** An `AT START OF DATA` statement may be related to a specific outer active processing loop by using the notation `(r)`. If this notation is not used, the statement is related to the outermost active processing loop.
statement ...	**Statement(s) to be Executed at Start of Data Condition:** In structured mode, you must supply one or several suitable statements, depending on the situation. For an example of a statement, see ***Example*** below.
`END-START`	**End of AT START OF DATA Statement:** In structured mode, the Natural reserved word `END-START` must be used to end the `AT START OF DATA` statement. In reporting mode, use the `DO ... DOEND` statements to supply one or several suitable statements, depending on the situation, and to end the `AT START OF DATA` statement. If you specify only a single statement, you can omit the `DO ... DOEND` statements. With respect to good coding practice, this is not recommended.
statement ... `DO` *statement* `... DOEND`	

Example

```
** Example 'ASDEX1S': AT START OF DATA (structured mode)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
*
1 #CNTL (A1)  INIT <' '>
```

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

Output of Program ASDEX1S:

```
ENTER VALUE FOR CITY PARIS
```

After entering and confirming name of city:

```
RECORDS FOUND        26

ENTER 'D' TO DISPLAY RECORDS D
```

Records displayed:

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


MAIZIERE             ELISABETH
MARX                 JEAN-MARIE
REIGNARD             JACQUELINE
RENAUD               MICHEL
REMOUE               GERMAINE
LAVENDA              SALOMON
BROUSSE              GUY
GIORDA               LOUIS
SIECA                FRANCOIS
```

```
CENSIER                 BERNARD
DUC                     JEAN-PAUL
CAHN                    RAYMOND
MAZUY                   ROBERT
FAURIE                  HENRI
VALLY                   ALAIN
BRETON                  JEAN-MARIE
GIGLEUX                 JACQUES
KORAB-BRZOZOWSKI        BOGDAN
XOLIN                   CHRISTIAN
LEGRIS                  ROGER
VVVV
```

Equivalent reporting-mode example: **ASDEX1R**.

19 AT TOP OF PAGE

Structured Mode Syntax

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

Reporting Mode Syntax

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

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

Related Statements: AT END OF PAGE | CLOSE PRINTER | DEFINE PRINTER | DISPLAY | EJECT | FORMAT | NEWPAGE | PRINT | SKIP | SUSPEND IDENTICAL SUPPRESS | WRITE | WRITE TITLE | WRITE TRAILER

Belongs to Function Group: ***Creation of Output Reports***

Function

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

See also the following sections in the *Programming Guide*:

- *Report Format and Control*
- *Report Specification - (rep) Notation*
- *Layout of an Output Page*
- *AT TOP OF PAGE Statement*

Processing

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

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

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

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

Restriction

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

Syntax Description

Syntax Element	Description
`(`*rep*`)`	**Report Specification:** The notation `(`*rep*`)` may be used to specify the identification of the report for which the `AT TOP OF PAGE` statement is applicable. A value in the range `0 - 31` or a logical name which has been assigned using the `DEFINE PRINTER` statement may be specified. If `(`*rep*`)` is not specified, the `AT TOP OF PAGE` statement applies to the first report (Report 0). For information on how to control the format of an output report created with Natural, see *Report Format and Control* in the *Programming Guide*.
statement ...	**Statement(s) to be Executed at Start of Data Condition:** In structured mode, you must supply one or several suitable statements, depending on the situation. For an example of a statement, see ***Example*** below.
`END-TOPPAGE`	**End of AT TOP OF PAGE Statement:** In structured mode, the Natural reserved word `END-TOPPAGE` must be used to end the `AT TOP OF PAGE` statement.
statement ... `DO` *statement* ... `DOEND`	

Syntax Element	Description
	In reporting mode, use the DO ... DOEND statements to supply one or several suitable statements, depending on the situation, and to end the AT TOP OF PAGE statement. If you specify only a single statement, you can omit the DO ... DOEND statements. With respect to good coding practice, this is not recommended.

Example

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

Output of Program ATPEX1S:

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


  LAFON                   CHRISTIANE           PARIS                VENT18
  LANDMANN                HARRY                ESCHBORN             MARK29
  LANE                    JACQUELINE           DERBY                MGMT02
```

```
  LANKATILLEKE            LALITH              FRANKFURT           PROD22
  LANNON                  BOB                 LINCOLN             SALE20
  LANNON                  LESLIE              SEATTLE             SALE30
  LARSEN                  CARL                FARUM               SYSA01
  LARSEN                  MOGENS              VEMMELEV            SYSA02
----------------------------------------------------------------------------


ENDING NAME:    LARSEN
```

Equivalent reporting-mode example: **ATPEX1R**.

20 BACKOUT TRANSACTION

```
BACKOUT [TRANSACTION]
```

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

Related Statements: ACCEPT/REJECT | AT BREAK | AT START OF DATA | AT END OF DATA | BEFORE BREAK PROCESSING | DELETE | END TRANSACTION | FIND | GET | GET SAME | GET TRANSACTION DATA | HISTOGRAM | LIMIT | PASSW | PERFORM BREAK PROCESSING | READ | RETRY | STORE | UPDATE

Belongs to Function Group: ***Database Access and Update***

Function

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

The statement is executed only if a database transaction under control of Natural has taken place. For which databases the statement is executed depends on the setting of the profile parameter ET (execution of END/BACKOUT TRANSACTION statements):

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

Backout Transaction Issued by Natural

If the user interrupts the current Natural operation with a terminal command (command %% or CLEAR key), Natural issues a BACKOUT TRANSACTION statement.

See also the terminal command %% in the *Terminal Commands* documentation.

Additional Information

For additional information on the use of the transaction backout feature, see the sections *Database Update - Transaction Processing* and *Backing Out a Transaction* in the *Programming Guide*.

Restriction

This statement is not available with Entire System Server.

Database-Specific Considerations

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

Example

```
** Example 'BOTEX1': BACKOUT TRANSACTION
**
** CAUTION: Executing this example will modify the database records!
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 DEPT
  2 LEAVE-DUE
  2 LEAVE-TAKEN
*
1 #DEPT (A6)
1 #RESP (A3)
END-DEFINE
*
LIMIT 3
INPUT 'DEPARTMENT TO BE UPDATED:' #DEPT
IF #DEPT = ' '
  STOP
END-IF
*
FIND EMPLOY-VIEW WITH DEPT = #DEPT
  IF NO RECORDS FOUND
    REINPUT 'NO RECORDS FOUND'
  END-NOREC
  INPUT 'NAME:       ' NAME (AD=O) /
        'LEAVE DUE:  ' LEAVE-DUE (AD=M) /
        'LEAVE TAKEN:' LEAVE-TAKEN (AD=M)
```

```
  UPDATE
END-FIND
*
INPUT 'UPDATE TO BE PERFORMED? YES/NO:' #RESP
DECIDE ON FIRST #RESP
  VALUE 'YES'
    END TRANSACTION
  VALUE 'NO'
    BACKOUT TRANSACTION
  NONE
    REINPUT 'PLEASE ENTER YES OR NO'
END-DECIDE
*
END
```

Output of Program BOTEX1:

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

Result for department `MGMT30`:

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

Confirmation query:

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

21 BEFORE BREAK PROCESSING

Structured Mode Syntax

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

Reporting Mode Syntax

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

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

Related Statements: ACCEPT/REJECT | AT BREAK | AT START OF DATA | AT END OF DATA | BACKOUT TRANSACTION | DELETE | END TRANSACTION | FIND | GET | GET SAME | GET TRANSACTION | HISTOGRAM | LIMIT | PASSW | PERFORM BREAK PROCESSING | READ | RETRY | STORE | UPDATE

Belongs to Function Group: ***Database Access and Update***

Function

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

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

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

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

See also the following sections in the *Programming Guide*:

- *Control Breaks*
- *BEFORE BREAK PROCESSING Statement*
- *Example of BEFORE BREAK PROCESSING Statement*

Restrictions

- The BEFORE BREAK PROCESSING statement may only be used with a processing loop that has been initiated with one of the following statements:
  - FIND
  - READ
  - HISTOGRAM
  - SORT
  - READ WORK FILE

  It may be placed anywhere within the processing loop and is always related to the processing loop in which it is contained. Only one BEFORE BREAK PROCESSING statement may be specified per processing loop.
- The BEFORE BREAK PROCESSING statement must not be used in conjunction with the statement PERFORM BREAK PROCESSING.

Syntax Description

Syntax Element	Description
statement...	**Statement(s) for Break Processing:** In place of *statement*, you must supply one or several suitable statements, depending on the situation. For an example of a statement, see ***Example*** below. If no break processing is to be performed (that is, no AT BREAK statement is specified for the processing loop), any statements specified with a BEFORE BREAK PROCESSING statement will *not* be executed.
END-BEFORE	**End of BEFORE BREAK PROCESSING Statement:** In structured mode, the Natural reserved word END-BEFORE must be used to end the BEFORE BREAK PROCESSING statement. In reporting mode, use the DO ... DOEND statements to supply one or several suitable statements, depending on the situation, and to end the BEFORE BREAK PROCESSING statement. If you specify only a single statement, you can omit the DO ... DOEND statements. With respect to good coding practice, this is not recommended.
statement ... DO *statement* ... DOEND	

Example

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

Output of Program BBPEX1:

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

LA BASSEE            HULOT                    165000      70000
AVERAGE INCOME FOR LA BASSEE                                     235000

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

22 CALL

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

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

Related Statements: `CALL FILE` | `CALL LOOP` | `CALLNAT` | `DEFINE SUBROUTINE` | `ESCAPE` | `FETCH` | `PERFORM`

Belongs to Function Group: ***Invoking Programs and Routines***

Function

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

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

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S				A													yes	no
operand2	C	S	A	G		A	U	N	P	I	F	B	D	T	L	C	G		yes	yes

Syntax Element Description:

Syntax Element	Description
`INTERFACE4`	**Interface Usage:** The optional keyword `INTERFACE4` specifies the type of the interface that is used for the call of the external program. See the section ***INTERFACE4*** below.
operand1	**Name of Called Function:** The name of the function to be called (*operand1*) can be specified as a constant or - if different functions are to be called dependent on program logic - as an alphanumeric variable of length 1 to 32. A function name must be placed left-justified in the variable.
`[USING]` *operand2*	**Parameters to be Passed:**

Syntax Element	Description
	The `CALL` statement may contain up to 128 parameters (*operand2*). One address is passed in the parameter list for each parameter field specified. If a group name is used, the group is converted to individual fields; that is, if a user wishes to specify the beginning address of a group, the first field of the group must be specified. **Note:** If an application-independent variable (AIV) or context variable is passed as a parameter to a user exit, the following restriction applies: if the user exit invokes a Natural subprogram which creates a new AIV or context variable, the parameter is invalid after the return from the subprogram. This is true regardless of whether the new AIV/context variable is created by the subprogram itself or by another object invoked directly or indirectly by the subprogram.

Return Code

The condition code of any called function may be obtained by using the Natural system function `RET` (Return Code Function).

Example:

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

User Exits

User exits are needed to make external functions available and to access operating-system interfaces that are not available to Natural.

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

Notes:

1. If you want to use user exits in a `CALL` statement, **User-defined libraries** must be set in the **Installation Assignments** of the **Local Configuration File**. Refer to *Installation Assignments* in the

section *Local Configuration File* of the *Overview of Configuration File Parameters* in the *Configuration Utility* documentation.

2. If you want to specify several libraries, you have to separate the names with a colon, for example `userlib1:userlib2:userlib3`.

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

A user exit is added to Natural in three steps:

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

The following topics are covered below:

Step 1 - Defining the Jump Table

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

`<install-dir>`/natural/samples/sysexuex

Step 2 - Writing the External Functions

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

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


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

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

The data type FINFO is defined as follows:

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

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

`<install-dir>`*/natural/samples/sysexuex*

Step 3 - Compiling and Linking

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

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

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

`<install-dir>`*/natural/samples/sysexuex*

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

How to Build a Shared Library

1. From the example directory, which is contained in `<install-dir>`*/natural/samples/sysexuex*, copy the following files into your work directory:
   - *Makedyn*
   - *jumptab.c*
   - *ncuxinit.c*
2. Copy the C source files which contain your user exits into the same work directory.
3. Edit the file *jumptab.c* to include the names and function pointers for your user exits. To do so, you add in Section 2 the external declarations of your user exits, and in Section 3 you add the name/function-pointer pairs for your user exits.
4. Edit the makefile as follows:
   - Specify the names of the object files containing the user exits in the following line:

     ```
     USEROBJS =
     ```

   - Specify the name of the resulting shared library in the following line:

     ```
     USERLIB =
     ```

   - If you need to include private header files, specify the directories containing them in the following line:

     ```
     INCDIR =
     ```

5. To remove all unneeded files, issue the command:

   ```
   make -f Makedyn clean
   ```

6. To compile and link your shared library, issue the command:

```
make -f Makedyn lib
```

How to Generate a Static Nucleus

1. From the example directory, which is contained in `<install-dir>`*/natural/samples/sysexuex*, copy the following files into your work directory:
   - *Makefile*
   - *jumptab.c*
2. Copy the C source files which contain your user exits into the same work directory.
3. Edit the file *jumptab.c* to include the names and function pointers for your user exits. To do so, you add in Section 2 the external declarations of your user exits, and in Section 3 you add the name/function-pointer pairs for your user exits.
4. Edit the makefile as follows:
   - Specify the names of the object files containing the user exits in the following line:

     ```
     USEROBJS = ↩
     ```

   - If you need to include private header files, specify the directories containing them in the following line:

     ```
     INCDIR =
     ```

5. Issue the command `make` to get information about further processing options.

Example:

See the sample user exit function in `<install-dir>`*/natural/samples/sysexuex*.

Example Programs

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

INTERFACE4

The keyword INTERFACE4 specifies the type of the interface that is used for the call of the external program. This keyword is optional. If this keyword is specified, the interface, which is defined as INTERFACE4, is used for the call of the external program.

The following table lists the differences between the CALL statement used with INTERFACE4 and the one used without INTERFACE4:

	CALL statement without keyword INTERFACE4	CALL statement with keyword INTERFACE4
Number of parameters possible	128	32767
Maximum data size of one parameter	65535	1 GB
Retrieve array information	no	yes
Support of large and dynamic operands	no	yes
Parameter access via API	no	yes

The following topics are covered below:

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

INTERFACE4 - External 3GL Program Interface

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

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

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

Operand Structure for INTERFACE4

The operand structure of `INTERFACE4` is named `parameter_description` and is defined as follows. The structure is delivered with the header file *natuser.h*.

struct parameter_description			
void *	address	Address of the parameter data, not aligned, `realloc()` and `free()` are not allowed.	
int	format	Field data format: `NCXR_TYPE_ALPHA`, etc. (*natuser.h*).	
int	length	Length (before decimal point, if applicable).	
int	precision	Length after decimal point (if applicable).	
int	byte_length	Length of field in bytes int dimension number of dimensions (`0` to `IF4_MAX_DIM`).	
int	dimensions	Number of dimensions (`0` to `IF4_MAX_DIM`).	
int	length_all	Total data length of array in bytes.	
int	flags	Several flag bits combined by bitwise `OR` operation, meaning:	
		`IF4_FLG_PROTECTED`:	The parameter is write-protected.
		`IF4_FLG_DYNAMIC`:	The parameter is a dynamic variable.
		`IF4_FLG_NOT_CONTIGUOUS`:	The array elements are not contiguous (have spaces between them).
		`IF4_FLG_AIV`:	The parameter is an application-independent variable.
		`IF4_FLG_DYNVAR`:	The parameter is a dynamic variable.
		`IF4_FLG_XARRAY`:	The parameter is an X-array.
		`IF4_FLG_LBVAR_0`:	The lower bound of dimension 0 is variable.
		`IF4_FLG_UBVAR_0`:	The upper bound of dimension 0 is variable.
		`IF4_FLG_LBVAR_1`:	The lower bound of dimension 1 is variable.
		`IF4_FLG_UBVAR_1`:	The upper bound of dimension 1 is variable.
		`IF4_FLG_LBVAR_2`:	The lower bound of dimension 2 is variable.
		`IF4_FLG_UBVAR_2`:	The upper bound of dimension 2 is variable.
int	occurrences[IF4_MAX_DIM]	Array occurrences in each dimension.	
int	indexfactors[IF4_MAX_DIM]	Array index factors for each dimension.	

`void *`	`dynp`	Reserved for internal use.
`void *`	`pops`	Reserved for internal use.

The address element is null for arrays of dynamic variables and for X-arrays. In these cases, the array data cannot be accessed as a whole, but must be accessed through the parameter access functions described below.

For arrays with fixed bounds of variables with fixed length, the array contents can be accessed directly using the address element. In these cases the address of an array element (i,j,k) is computed as follows (especially if the array elements are not contiguous):

```
elementaddress = address + i * indexfactors[0] + j * indexfactors[1] + k * ↩
indexfactors[2]
```

If the array has less than 3 dimensions, leave out the last terms.

INTERFACE4 - Parameter Access

A set of functions is available to be used for the access of the parameters. The process flow is as follows:

- The 3GL program is called via the `CALL` statement with the `INTERFACE4` option, and the parameters are passed to the 3GL program as described above.
- The 3GL program can now use the exported functions of Natural, to retrieve either the parameter data itself, or information about the parameter, such as format, length, array information, etc.
- The **exported functions** can also be used to pass back parameter data.

There are also functions to create and initialize a new parameter set in order to call arbitrary subprograms from a 3GL program. With this technique a parameter access is guaranteed to avoid memory overwrites done by the 3GL program. (Natural's data is safe: memory overwrites within the 3GL program's data are still possible).

Exported Functions

The following topics are covered below:

- Resize an X-array Parameter

Get Parameter Information

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

Prototype:

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

Parameter Description:

parmnum	Ordinal number of the parameter. This identifies the parameter of the passed parameter list. Range: 0 ... numparm-1.	
parmhandle	Pointer to the internal parameter structure	
descr	Address of a struct parameter_description	
return	Return Value:	Information:
	0	OK
	-1	Illegal parameter number.
	-2	Internal error.
	-7	Interface version conflict.

Get Parameter Data

This function is used by the 3GL program to get the data from any parameter.

Natural identifies the parameter by the given parameter number and writes the parameter data to the given buffer address with the given buffer size.

If the parameter data is longer than the given buffer size, Natural will truncate the data to the given length. The external 3GL program can make use of the function ncxr_get_parm_info, to request the length of the parameter data.

There are two functions to get parameter data: ncxr_get_parm gets the whole parameter (even if the parameter is an array), whereas ncxr_get_parm_array gets the specified array element.

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

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

Prototypes:

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

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

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

Parameter Description:

<table>
<tr><td><code>parmnum</code></td><td colspan="2">Ordinal number of the parameter. This identifies the parameter of the passed parameter list. Range: <code>0 ... numparm-1</code>.</td></tr>
<tr><td><code>parmhandle</code></td><td colspan="2">Pointer to the internal parameter structure</td></tr>
<tr><td><code>buffer_length</code></td><td colspan="2">Length of the buffer, where the requested data has to be written to</td></tr>
<tr><td><code>buffer</code></td><td colspan="2">Address of buffer, where the requested data has to be written to. This buffer should be aligned to allow easy access to I2/I4/F4/F8 variables.</td></tr>
<tr><td><code>indexes</code></td><td colspan="2">Array with index information</td></tr>
<tr><td rowspan="12"><code>return</code></td><td>Return Value:</td><td>Information:</td></tr>
<tr><td>< <code>0</code></td><td>Error during retrieval of the information:</td></tr>
<tr><td><code>-1</code></td><td>Illegal parameter number.</td></tr>
<tr><td><code>-2</code></td><td>Internal error.</td></tr>
<tr><td><code>-3</code></td><td>Data has been truncated.</td></tr>
<tr><td><code>-4</code></td><td>Data is not an array.</td></tr>
<tr><td><code>-7</code></td><td>Interface version conflict.</td></tr>
<tr><td><code>-100</code></td><td>Index for dimension 0 is out of range.</td></tr>
<tr><td><code>-101</code></td><td>Index for dimension 1 is out of range.</td></tr>
<tr><td><code>-102</code></td><td>Index for dimension 2 is out of range.</td></tr>
<tr><td><code>0</code></td><td>Successful operation.</td></tr>
<tr><td>> <code>0</code></td><td>Successful operation, but the data was only this number of bytes long (buffer was longer than the data).</td></tr>
</table>

Write Back Operand Data

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

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

Prototypes:

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

Parameter Description:

parmnum	Ordinal number of the parameter. This identifies the parameter of the passed parameter list. Range: 0 ... numparm-1.	
parmhandle	Pointer to the internal parameter structure.	
buffer_length	Length of the data to be copied back to the address of buffer, where the data comes from.	
indexes	Index information	
return	Return Value:	Information:
	< 0	Error during copying of the information:
	-1	Illegal parameter number.
	-2	Internal error.
	-3	Too much data has been given. The copy back was done with parameter length.
	-4	Parameter is not an array.
	-5	Parameter is protected (constant or AD=O).
	-6	Dynamic variable could not be resized due to an "out of memory" condition.
	-7	Interface version conflict.
	-13	The given buffer includes an incomplete Unicode character.
	-100	Index for dimension 0 is out of range.

	-101	Index for dimension 1 is out of range.
	-102	Index for dimension 2 is out of range.
	0	Successful operation.
	> 0	Successful operation, but the parameter was this number of bytes long (length of parameter greater than given length).

Create, Initialize and Delete a Parameter Set

If a 3GL program wants to call a Natural subprogram, it needs to build a parameter set that corresponds to the parameters the subprogram expects. The function `ncxr_create_parm` is used to create a set of parameters to be passed with a call to `ncxr_if_callnat`. The set of parameters created is represented by an opaque parameter handle, like the parameter set that is passed to the 3GL program with the `CALL INTERFACE4` statement. Thus, the newly created parameter set can be manipulated with functions `ncxr_put_parm*` and `ncxr_get_parm*` as described above.

The newly created parameter set is not yet initialized after having called the function `ncxr_create_parm`. An individual parameter is initialized to a specific data type by a set of `ncxr_parm_init*` functions described below. The functions `ncxr_put_parm*` and `ncxr_get_parm*` are then used to access the contents of each individual parameter. After the caller has finished with the parameter set, they must delete the parameter handle. Thus, a typical sequence in creating and using a set of parameters for a subprogram to be called through `ncxr_if4_callnat` will be:

```
ncxr_create_parm
ncxr_init_ parm*
ncxr_init_ parm*
...
ncxr_put_ parm*
ncxr_put_ parm*
...
ncxr_get_parm_info*
ncxr_get_parm_info*
...
ncxr_if4_callnat
...
ncxr_get_parm_info*
ncxr_get_parm_info*
...
ncxr_get_ parm*
ncxr_get_ parm*
...
ncxr_delete_parm
```

Create Parameter Set

The function `ncxr_create_parm` is used to create a set of parameters to be passed with a call to `ncxr_if_callnat`.

Prototype:

```
int ncxr_create_parm( int parmnum, void** pparmhandle )
```

Parameter Description:

`parmnum`	Number of parameters to be created.	
`pparmhandle`	Pointer to the created parameter handle.	
`return`	Return Value:	Information:
	< 0	Error:
	-1	Illegal parameter count.
	-2	Internal error.
	-6	Out of memory condition.
	0	Successful operation.

Delete Parameter Set

The function `ncxr_delete_parm` is used to delete a set of parameters that was created with `ncxr_create_parm`.

Prototype:

```
int ncxr_delete_parm( void* parmhandle )
```

Parameter Description:

`parmhandle`	Pointer to the parameter handle to be deleted.	
`return`	Return Value:	Information:
	< 0	Error:
	-2	Internal error.
	0	Successful operation.

Initialize a Scalar of a Static Data Type

Prototype:

```
int ncxr_init_parm_s( int parmnum, void *parmhandle,
    char format, int length, int precision, int flags );
```

Parameter Description:

parmnum	Ordinal number of the parameter. This identifies the parameter in the passed parameter list. Range: 0 ... numparm-1.	
parmhandle	Pointer to the parameter handle.	
format	Format of the parameter.	
length	Length of the parameter.	
precision	Precision of the parameter.	
flags	IF4_FLG_PROTECTED	
return	Return Value:	Information:
	< 0	Error:
	-1	Invalid parameter number.
	-2	Internal error.
	-6	Out of memory condition.
	-8	Invalid format.
	-9	Invalid length or precision.
	0	Successful operation.

Initialize an Array of a Static Data Type

Prototype:

```
int ncxr_init_parm_sa( int parmnum, void *parmhandle,
    char format, int length, int precision,
    int dim, int *occ, int flags );
```

Parameter Description:

parmnum	Ordinal number of the parameter. This identifies the parameter in the passed parameter list. Range: 0 ... numparm-1.
parmhandle	Pointer to the parameter handle.
format	Format of the parameter.
length	Length of the parameter.
precision	Precision of the parameter.

`dim`	Dimension of the array.	
`occ`	Number of occurrences per dimension.	

`flags`	A combination of the flags
	
	`IF4_FLG_PROTECTED`
	`IF4_FLG_LBVAR_0`
	`IF4_FLG_UBVAR_0`
	`IF4_FLG_LBVAR_1`
	`IF4_FLG_UBVAR_1`
	`IF4_FLG_LBVAR_2`
	`IF4_FLG_UBVAR_2`	

`return`	Return Value:	Information:
	< `0`	Error:
	`-1`	Invalid parameter number.
	`-2`	Internal error.
	`-6`	Out of memory condition.
	`-8`	Invalid format.
	`-9`	Invalid length or precision.
	`-10`	Invalid dimension count.
	`-11`	Invalid combination of variable bounds.
	`0`	Successful operation.

Initialize a Scalar of a Dynamic Data Type

Prototype:

```
int ncxr_init_parm_d( int parmnum, void *parmhandle,
    char format, int flags );
```

Parameter Description:

`parmnum`	Ordinal number of the parameter. This identifies the parameter in the passed parameter list. Range: `0 ... numparm-1`.	
`parmhandle`	Pointer to the parameter handle.	
`format`	Format of the parameter.	
`flags`	`IF4_FLG_PROTECTED`	
`return`	Return Value:	Information:
	< `0`	Error:
	`-1`	Invalid parameter number.
	`-2`	Internal error.
	`-6`	Out of memory condition.

	-8	Invalid format.
	0	Successful operation.

Initialize an Array of a Dynamic Data Type

Prototype:

```
int ncxr_init_parm_da( int parmnum, void *parmhandle,
    char format, int dim, int *occ, int flags );
```

Parameter Description:

parmnum	Ordinal number of the parameter. This identifies the parameter in the passed parameter list. Range: 0 ... numparm-1.	
parmhandle	Pointer to the parameter handle.	
format	Format of the parameter.	
dim	Dimension of the array.	
occ	Number of occurrences per dimension.	
flags	A combination of the flags IF4_FLG_PROTECTED IF4_FLG_LBVAR_0 IF4_FLG_UBVAR_0 IF4_FLG_LBVAR_1 IF4_FLG_UBVAR_1 IF4_FLG_LBVAR_2 IF4_FLG_UBVAR_2	
return	Return Value:	Information:
	< 0	Error:
	-1	Invalid parameter number.
	-2	Internal error.
	-6	Out of memory condition.
	-8	Invalid format.
	-10	Invalid dimension count.
	-11	Invalid combination of variable bounds.
	0	Successful operation.

Resize an X-array Parameter

Prototype:

```
int ncxr_resize_parm_array( int parmnum, void *parmhandle, int *occ );
```

Parameter Description:

parmnum	Ordinal number of the parameter. This identifies the parameter in the passed parameter list. Range: `0 ... numparm-1`.	
parmhandle	Pointer to the parameter handle.	
occ	New number of occurrences per dimension.	
return	Return Value:	Information:
	< 0	Error:
	-1	Invalid parameter number.
	-2	Internal error.
	-6	Out of memory condition.
	-12	Operand is not resizable (in one of the specified dimensions).
	0	Successful operation.

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

23 CALL FILE

Structured Mode Syntax

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

Reporting Mode Syntax

```
CALL FILE 'program-name' operand1 operand2
  statement ...
LOOP
```

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

Related Statements: `CALL` | `CALL LOOP` | `CALLNAT` | `DEFINE SUBROUTINE` | `ESCAPE` | `FETCH` | `PERFORM`

Belongs to Function Group: ***Invoking Programs and Routines***

Function

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

Restriction

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

Syntax Description

Operand Definition Table:

Operand	Possible Structure				Possible Formats	Referencing Permitted	Dynamic Definition
operand1		S	A		A U N P I F B D T L C	yes	yes
operand2		S	A	G	A U N P I F B D T L C	yes	yes

Syntax Element Description:

Syntax Element	Description
'*program-name*'	**Program to be Called:** The name of the non-Natural program to be called.
operand1	**Control Field:** *operand1* is used to provide control information.
operand2	**Record Area:** *operand2* defines the record area. The format of the record to be read can be described using field definitions (or `FILLER` *n*`X`) entries following the name of the first field in the record. The fields used to define the record format must not have been previously defined in the Natural program. This ensures that fields are allocated in the contiguous storage by Natural.
statement ...	**Processing Loop:** The `CALL FILE` statement initiates a processing loop which must be terminated with an `ESCAPE` or `STOP` statement. More than one `ESCAPE` statement may be specified to escape from a `CALL FILE` loop based on different conditions.
`END-FILE`	**End of CALL FILE Statement:** In structured mode, the Natural reserved keyword `END-FILE` must be used to end the `CALL FILE` statement. In reporting mode, the Natural statement `LOOP` is used to end the `CALL FILE` statement.
`LOOP`	

Example

Calling Program:

```
** Example 'CFIEX1': CALL FILE
************************************************************************
DEFINE DATA LOCAL
1 #CONTROL (A3)
1 #RECORD
  2 #A     (A10)
  2 #B     (N3.2)
  2 #FILL1 (A3)
  2 #C     (P3.1)
END-DEFINE
*
```

```
CALL FILE 'USER1' #CONTROL #RECORD
  IF #CONTROL = 'END'
    ESCAPE BOTTOM
  END-IF
END-FILE
/*****************************
/* ... PROCESS RECORD ...
/*****************************
END
```

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

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

xxx xxxxxxxxxx xxxxx   xxx   xxx
```

Called COBOL Program:

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

24 CALL LOOP

Structured Mode Syntax

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

Reporting Mode Syntax

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

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

Related Statements: `CALL` | `CALL FILE` | `CALLNAT` | `DEFINE SUBROUTINE` | `ESCAPE` | `FETCH` | `PERFORM`

Belongs to Function Group: ***Invoking Programs and Routines***

Function

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

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

Restriction

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

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S				A													yes	no
operand2	C	S	A	G		A	U	N	P	I	F	B	D	T	L	C			yes	yes

Syntax Element Description:

Syntax Element	Description
operand1	**Program to be Called:** The name of the non-Natural program to be called can be specified as a constant or - if different programs are to be called dependent on program logic - as an alphanumeric variable of length 1 to 32. A program name must be placed left-justified in the variable.
operand2	**Parameters:** The `CALL LOOP` statement can have a maximum of 40 parameters. The parameter list is constructed as described for the `CALL` statement. Fields used in the parameter list may be initially defined in the `CALL LOOP` statement itself or may have been previously defined.
statement ...	**Processing Loop:** The `CALL LOOP` statement initiates a processing loop which must be terminated with an `ESCAPE` statement.
`END-LOOP`	**End of CALL LOOP Statement:** In structured mode, the Natural reserved word `END-LOOP` must be used to end the `CALL LOOP` statement. In reporting mode, the Natural statement `LOOP` is used to end the `CALL LOOP` statement.
`LOOP`	

Example

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

25 CALLDBPROC (SQL)

```
CALLDBPROC dbproc ddm-name
                                       M
[ [USING] { parameter [    AD= {     O    } ] ] } ... ]
                                       A
  [RESULT SETS result-set...]
  [GIVING sqlcode]
```

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

Belongs to Function Group: ***Database Access and Update***

Function

The `CALLDBPROC` statement is used to invoke a stored procedure of the SQL database system to which Natural is connected.

The stored procedure can be either a Natural subprogram (only available when executed from Db2 for z/OS) or a program written in another programming language.

In addition to the passing of parameters between the invoking object and the stored procedure, `CALLDBPROC` supports "result sets"; these make it possible to return a larger amount of data from the stored procedure to the invoking object than would be possible via parameters.

The result sets are "temporary result tables" which are created by the stored procedure and which can be read and processed by the invoking object via a `READ RESULT SET` statement.

Note: In general, the invoking of a stored procedure could be compared with the invoking of a Natural subprogram: when the `CALLDBPROC` statement is executed, control is passed to the stored procedure; after processing of the stored procedure, control is returned to the invoking object and processing continues with the statement following the `CALLDBPROC` statement.

Syntax Description

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td>dbproc</td><td>Stored Procedure to be Invoked:

As dbproc you specify the name of the stored procedure to be invoked. The name can be specified either as an alphanumeric variable or as a constant (enclosed in apostrophes).

The name must adhere to the rules for stored procedure names of the target database system.

If the stored procedure is a Natural subprogram, the actual procedure name must not be longer than 8 characters.</td></tr>
<tr><td>ddm-name</td><td>Name of a Natural Data Definition Module:

The name of a DDM must be specified to provide the “address” of the database which executes the stored procedure. For further information, see ddm-name.</td></tr>
<tr><td>[USING] parameter</td><td>Parameter(s) to be Passed:

As parameter, you can specify parameters which are passed from the invoking object to the stored procedure. A parameter can be

■ a host-variable (optionally with INDICATOR and LINDICATOR clauses),
■ a constant, or
■ the keyword NULL.

See further details on host-variable.</td></tr>
<tr><td>AD=</td><td>Attribute Definition:

If parameter is a host-variable, you can mark it as follows:
<table>
<tr><td>AD=O</td><td>Non-modifiable, see session parameter AD=O.

(Corresponding procedure notation in Db2 for z/OS: IN.)</td></tr>
<tr><td>AD=M</td><td>Modifiable, see session parameter AD=M.

(Corresponding procedure notation in Db2 for z/OS: INOUT.)</td></tr>
<tr><td>AD=A</td><td>For input only, see session parameter AD=A.

(Corresponding procedure notation in Db2 for z/OS: OUT.)</td></tr>
</table>
If parameter is a constant, AD cannot be explicitly specified. For constants, AD=O always applies.</td></tr>
</table>

Syntax Element	Description
RESULT SETS *result-set*	**Field for Result-Set Locator Variable:** As *result-set* you specify a field in which a result-set locator is to be returned. A result set has to be a variable of format/length I4. The value of a result set variable is merely a number which identifies the result set and which can be referenced in a subsequent READ RESULT SET statement. The sequence of the *result-set* values correspond to the sequence of the result sets returned by the stored procedure. The contents of the result sets can be processed by a subsequent READ RESULT SET statement. If no result set is returned, the corresponding result-set variable will contain 0. Only one result set can be specified.
GIVING *sqlcode*	**GIVING *sqlcode* Option:** This option may be used to obtain the SQLCODE of the SQL CALL statement invoking the stored procedure. If this option is specified and the SQLCODE of the stored procedure is not 0, no Natural error message will be issued. In this case, the action to be taken in reaction to the SQLCODE value has to be coded in the invoking Natural object. The *sqlcode* field has to be a variable of format/length I4. If the GIVING *sqlcode* option is omitted, a Natural error message will be issued if the SQLCODE of the stored procedure is not 0.

Example

The following example shows a Natural program that calls the stored procedure DEMO_PROC to retrieve all names of table PERSON that belong to a given range.

Three parameter fields are passed to DEMO_PROC: the first and second parameters pass starting and ending values of the range of names to the stored procedure, and the third parameter receives a name that meets the criterion.

In this example, the names are returned in a result set that is processed using the READ RESULT SET statement.

```
DEFINE DATA LOCAL
1 PERSON VIEW OF DEMO-PERSON
  2 PERSON_ID
  2 LAST_NAME
1 #BEGIN    (A2) INIT <'AB'>
1 #END      (A2) INIT <'DE'>
1 #RESPONSE (I4)
1 #RESULT   (I4)
1 #NAME (A20)
END-DEFINE

...

CALLDBPROC 'DEMO_PROC' DEMO-PERSON #BEGIN (AD=O) #END (AD=O) #NAME (AD=A)
    RESULT SETS #RESULT
    GIVING #RESPONSE

READ RESULT SET #RESULT INTO #NAME FROM DEMO-PERSON
    GIVING #RESPONSE
  DISPLAY #NAME
END-RESULT

...

END
```

26 CALLNAT

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

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

Related Statements: `CALL` | `CALL FILE` | `CALL LOOP` | `DEFINE SUBROUTINE` | `ESCAPE` | `FETCH` | `PERFORM`

Belongs to Function Group: ***Invoking Programs and Routines***

Function

The `CALLNAT` statement is used to invoke a Natural subprogram for execution. (A Natural subprogram can only be invoked via a `CALLNAT` statement; it cannot be executed by itself.)

When the `CALLNAT` statement is executed, the execution of the invoking object (that is, the object containing the `CALLNAT` statement) will be suspended and the invoked subprogram will be executed. The execution of the subprogram continues until either its `END` statement is reached or processing of the subprogram is stopped by an `ESCAPE ROUTINE` statement being executed. In either case, processing of the invoking object will then continue with the statement following the `CALLNAT` statement.

Notes:

1. A subprogram can in turn invoke other subprograms.
2. A subprogram has no access to the global data area used by the invoking object. If a subprogram in turn invokes a subroutine or helproutine, it can establish its own global data area to be shared with the subroutine/helproutine.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S				A													yes	no
operand2	C	S	A	G		A	U	N	P	I	F	B	D	T	L	C	G	O	yes	yes

Syntax Element Description:

Syntax Element	Description
operand1	**Subprogram to be Invoked:** As *operand1*, you specify the name of the subprogram to be invoked. The name may be specified either as a constant of 1 to 32 characters, or - if different subprograms are to be called dependent on program logic - as an alphanumeric variable of length 1 to 8. The subprogram name may contain an ampersand (&); at execution time, this character will be replaced by the one-character code corresponding to the current value of the system variable `*LANGUAGE`. This makes it possible, for example, to invoke different subprograms for the processing of input, depending on the language in which input is provided.
operand2	**Parameters:** If parameters are passed to the subprogram, the structure of the parameter list must be defined in a `DEFINE DATA PARAMETER` statement. The parameters specified with the `CALLNAT` statement are the only data available to the subprogram from the invoking object. By default, the parameters are passed *by reference*, that is, the data are transferred via address parameters, the parameter values themselves are not moved. However, it is also possible to pass parameters *by value*, that is, pass the actual parameter values. To do so, you define these fields in the `DEFINE DATA PARAMETER` statement of the subprogram with the option `BY VALUE` or `BY VALUE RESULT` (see *`parameter-data-definition`* in the description of the `DEFINE DATA` statement). ■ If parameters are passed *by reference*, the following applies: The sequence, format and length of the parameters in the invoking object must match exactly the sequence, format and length of the `DEFINE DATA PARAMETER` structure in the invoked subprogram. The names of the variables in the invoking object and the invoked subprogram may be different. ■ If parameters are passed *by value*, the following applies: The sequence of the parameters in the invoking object must match exactly the sequence in the `DEFINE DATA PARAMETER` structure of the invoked subprogram. Formats and lengths of the variables in the invoking object and the subprogram may be different; however, they have to be data transfer compatible; see the corresponding table in the section *Rules for Arithmetic Assignments*, *Data Transfer* in

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td></td><td>
the Programming Guide. The names of the variables in the invoking object and the subprogram may be different. If parameter values that have been modified in the subprogram are to be passed back to the invoking object, you have to define these fields with <code>BY VALUE RESULT</code>. When <code>BY VALUE</code> is specified without <code>RESULT</code>, it is not possible to pass modified parameter values back to the invoking object (regardless of the <code>AD</code> specification; see also below).

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

For both ways of passing parameters, the following applies:

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

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

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

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

When the option <code>PCHECK</code> of the <code>COMPOPT</code> command is set to <code>ON</code>, the compiler will check the number, format, length and array index bounds of the parameters that are specified in a <code>CALLNAT</code> statement. Also, the <code>OPTIONAL</code> feature of the <code>DEFINE DATA PARAMETER</code> statement is considered in the parameter check.

Note: Numeric constant parameters are internally represented in packed form (format P). For further information see the Programming Guide > Numeric Constants.
</td></tr>
<tr><td><code>AD=</code></td><td>
Attribute Definition:

If <code>operand2</code> is a variable, you can mark it in one of the following ways:
<table>
<tr><td><code>AD=O</code></td><td>Non-modifiable, see session parameter <code>AD=O</code>.

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

This is the default setting.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Syntax Element</th><th colspan="2">Description</th></tr>
<tr><td></td><td>AD=A</td><td>Input only, see session parameter AD=A.</td></tr>
<tr><td></td><td colspan="2">If operand2 is a constant, AD cannot be explicitly specified. For constants AD=O always applies.</td></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 subprogram. The possible range of values for n is 1 - 4096.

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

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. In addition, a call by value result causes the movement to change to the opposite direction. When using a call-by-reference, both definitions must be DYNAMIC. If only one of them is DYNAMIC, a runtime error is raised. In case of a call by value (result) all combinations are possible.

The following table illustrates the valid combinations of statically and dynamically defined variables of the caller, and statically and dynamically defined parameters concerning the parameter transfer.

Call By Reference

operand2 of caller	Parameter definition	
	Static	Dynamic
Static	yes	no
Dynamic	no	yes

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

Call by Value (Result)

operand2 of caller	Parameter definition	
	Static	Dynamic
Static	yes	yes
Dynamic	yes	yes

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

Examples

- Example 1
- Example 2

Example 1

Calling Program:

```
** Example 'CNTEX1': CALLNAT
************************************************************************
DEFINE DATA LOCAL
1 #FIELD1 (N6)
1 #FIELD2 (A20)
1 #FIELD3 (A10)
END-DEFINE
*
CALLNAT 'CNTEX1N' #FIELD1 (AD=M) #FIELD2 (AD=O) #FIELD3 'P4 TEXT'
*
WRITE '=' #FIELD1 '=' #FIELD2 '=' #FIELD3
*
END
```

Called Subprogram CNTEX1N:

```
** Example 'CNTEX1N': CALLNAT  (called by CNTEX1)
************************************************************************
DEFINE DATA PARAMETER
1 #FIELDA (N6)
1 #FIELDB (A20)
1 #FIELDC (A10)
1 #FIELDD (A7)
END-DEFINE
*
*
```

```
#FIELDA := 4711
*
#FIELDB := 'HALLO'
*
#FIELDC := 'ABC'
*
WRITE '=' #FIELDA '=' #FIELDB '=' #FIELDC '=' #FIELDD
*
END
```

Example 2

Calling Program:

```
** Example 'CNTEX2': CALLNAT
************************************************************************
DEFINE DATA LOCAL
1 #ARRAY1  (N4/1:10,1:10)
1 #NUM     (N2)
END-DEFINE
*
*
CALLNAT 'CNTEX2N' #ARRAY1 (2:5,*)
*
FOR #NUM 1 TO 10
  WRITE #NUM #ARRAY1(#NUM,1:10)
END-FOR
*
END
```

Called Subprogram CNTEX2N:

```
** Example 'CNTEX2N': CALLNAT  (called by CNTEX2)
************************************************************************
DEFINE DATA
PARAMETER
1 #ARRAY (N4/1:4,1:10)
LOCAL
1 I      (I2)
END-DEFINE
*
*
FOR I 1 10
  #ARRAY(1,I) := I
  #ARRAY(2,I) := 100 + I
  #ARRAY(3,I) := 200 + I
  #ARRAY(4,I) := 300 + I
END-FOR
*
END
```

27 CLOSE CONVERSATION

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

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

Related Statements: DEFINE DATA CONTEXT | OPEN CONVERSATION

Belongs to Function Group: ***Natural Remote Procedure Call***

Function

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

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

Syntax Description

Operand Definition Table:

Operand	Possible Structure			Possible Formats	Referencing Permitted	Dynamic Definition
operand1		S	A	I	yes	no

Syntax Element Description:

Syntax Element	Description
operand1	**Identifier of Conversation to be Closed:** To close a specific open conversation, specify its ID as *operand1*. *operand1* must be a variable of format/length I4.
*CONVID	**Closing the Current Conversation:** To close the current conversation, specify *CONVID. The ID of the current conversation is determined by the value of the system variable *CONVID.
ALL	**Closing All Open Conversations:** To close all open conversations, specify ALL.

Further Information and Examples

See the following sections in the *Natural RPC (Remote Procedure Call)* documentation:

- *Natural RPC Operation in Conversational Mode*
- *Using a Conversational RPC*

V

28 CLOSE PC FILE

```
CLOSE { PC   } [FILE] work-file-number
      { WORK }
```

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

Related Statements: `DOWNLOAD PC FILE` | `UPLOAD PC FILE`

Belongs to Function Group: ***Control of Work Files / PC Files***

Function

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

A work file is also closed automatically when command mode is reached.

See also the Natural Connection and Entire Connection documentation.

Syntax Description

Syntax Element	Description
work-file-number	The *work-file-number* is the number of the PC work file to be closed. This number must correspond to one of the work file numbers for the PC as defined to Natural.

Example

The following program demonstrates the use of the `CLOSE PC FILE` statement.

```
** Example 'PCCLEX1': CLOSE PC FILE
**
** NOTE: Example requires that Natural Connection is installed.
************************************************************************
DEFINE DATA LOCAL
01 W-DAT   (A40)
01 REC-NUM (N3)
01 I       (P3)
END-DEFINE
*
REPEAT
```

```
UPLOAD PC FILE 7 ONCE W-DAT                /* Data upload
  AT END OF FILE
    ESCAPE BOTTOM
  END-ENDFILE
  INPUT 'Processing file' W-DAT (AD=O)
    /   'Enter record number to display' REC-NUM
  IF REC-NUM = 0
    STOP
  END-IF
  FOR I = 1 TO REC-NUM
    UPLOAD PC FILE 7 ONCE W-DAT
    AT END OF FILE
      WRITE 'Max. record number reached, last record is'
      ESCAPE BOTTOM
    END-ENDFILE
  END-FOR
  I := I - 1
  WRITE 'Record' I ':' W-DAT
CLOSE PC FILE 7                            /* Close PC file 7
END-REPEAT
END
```

Output of Program PCCLEX1:

When you run the program, a window appears in which you specify the name of the PC file from which the data is to be uploaded. The data is then uploaded from the PC. At the end of each loop, the PC file is closed.

29 CLOSE PRINTER

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

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

Related Statements: AT END OF PAGE | AT TOP OF PAGE | DEFINE PRINTER | DISPLAY | EJECT | FORMAT | NEWPAGE | PRINT | SKIP | SUSPEND IDENTICAL SUPPRESS | WRITE | WRITE TITLE | WRITE TRAILER

Belongs to Function Group: ***Creation of Output Reports***

Function

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

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

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

Syntax Description

Syntax Element	Description
logical-printer-name	**Logical Printer Name:** With the *logical-printer-name* you specify which printer is to be closed. The name is the same as in the corresponding DEFINE PRINTER statement in which you defined the printer. Naming conventions for the *logical-printer-name* are the same as for user-defined variables, see *Naming Conventions for User-Defined Variables* in *Using Natural*.
printer-number	**Printer Number:** Alternatively to the *logical-printer-name*, you may define the *printer-number* to specify which printer is to be closed. The *printer-number* may be a number in the range from 0 - 31. This is the number also to be used in a DISPLAY / WRITE or DEFINE PRINTER statement. Printer number 0 indicates the hardcopy printer.

Example

```
** Example 'CLPEX1': CLOSE PRINTER
************************************************************************
DEFINE DATA LOCAL
1 EMP-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
  2 BIRTH
*
1 #I-NAME (A20)
END-DEFINE
*
DEFINE PRINTER (PRT01=1)
*
REPEAT
  INPUT 'SELECT PERSON' #I-NAME
  IF #I-NAME = ' '
    STOP
  END-IF
  FIND EMP-VIEW WITH NAME = #I-NAME
    WRITE (PRT01) 'NAME         :' NAME ',' FIRST-NAME
           /      'PERSONNEL-ID :' PERSONNEL-ID
           /      'BIRTH        :' BIRTH (EM=YYYY-MM-DD)
  END-FIND
  /*
  CLOSE PRINTER (PRT01)
  /*
END-REPEAT
END
```

30 CLOSE WORK FILE

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

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

Related Statements: `DEFINE WORK FILE` | `READ WORK FILE` | `WRITE WORK FILE`

Belongs to Function Group: ***Control of Work Files / PC Files***

Function

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

A work file is closed automatically:

- When command mode is reached.
- When an end-of-file condition occurs during the execution of a `READ WORK FILE` statement.
- Before a `DEFINE WORK FILE` statement is executed which assigns another file to the work file number concerned.

Syntax Description

Syntax Element	Description
work-file-number	**Work File Number:** The work file number (as defined to Natural) to be used. The work file number is either ■ a numeric constant in the value range 1:32 or ■ a numeric variable of type (B/N/P/I) defined with a `CONST` clause which assigning a value in range (1:32). Variable is a scalar (non-array) without precision digits for type (N/P), length in between 1-4 for type (B), and no redefinition field.

Example

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

31 COMMIT (SQL)

`COMMIT`

Belongs to Function Group: ***Database Access and Update***

Function

The SQL `COMMIT` statement corresponds to the `END TRANSACTION` statement. It indicates the end of a logical transaction and releases all data locked during the transaction. All data modifications are committed and made permanent.

Important: As all cursors are closed when a logical unit of work ends, a `COMMIT` statement must not be placed within a database modification loop; instead, it has to be placed outside such a loop or after the outermost loop of nested loops.

Example

```
...
DELETE FROM SQL-PERSONNEL WHERE NAME = 'SMITH'
COMMIT
...
```

32 COMPRESS

```
COMPRESS [NUMERIC] [FULL]

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

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

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

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

Related Statements: `ASSIGN` | `COMPUTE` | `EXAMINE` | `MOVE` | `MOVE ALL` | `SEPARATE`

Belongs to Function Group: ***Arithmetic and Data Movement Operations***

Function

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

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A	G	N	A	U	N	P	I	F	B	D	T	L		G	O	yes	no
operand2		S				A	U					B							yes	yes
operand3	C	S						N	P	I		B*							yes	no
operand4	C	S						N	P	I		B*							yes	no
operand5	C	S						N	P	I		B*							yes	no
operand6	C	S						N	P	I		B*							yes	no
operand7	C	S				A	U					B							yes	no

* Format B of `operand3`, `operand4`, `operand5` and `operand6` may be used only with a length of less than or equal to 4.

Syntax Element Description:

Syntax Element	Description

`NUMERIC`	**Handling of Sign Characters:**
	
	This option determines how sign characters and decimal characters are to be handled:
	
	Without `NUMERIC`, decimal points and signs in numeric source values are suppressed before the values are transferred. For example:
	
	`COMPRESS -123  1.23 INTO #TARGET WITH DELIMITER '*'`
	`Content of #TARGET is: 123*123`
	
	With `NUMERIC`, decimal points and signs in numeric source values are also transferred to the target field.
	
	For floating point source values, decimal points and signs are transferred, regardless of whether `NUMERIC` has been specified or not.
	
	Example 1:
	
	`COMPRESS NUMERIC -123  1.23 INTO #TARGET WITH DELIMITER '*'`
	`Content of #TARGET is: -123*1.23`
	
	Example 2:
	
	`COMPRESS NUMERIC 'ABC' -0056.00 -0056.10 -0056.01`
	`INTO #TARGET WITH DELIMITER '*'`
	`Content of #TARGET is: ABC*-56*-56.1*-56.01`
	
	Example 3:
	
	`COMPRESS NUMERIC FULL 'ABC' -0056.00 -0056.10 -0056.01`
	`INTO #TARGET WITH DELIMITER '*'`
	`Content of #TARGET is: ABC*-0056.00*-0056.10*-0056.01`

`FULL`	**Handling of Source Field Values:** Without `FULL`, the following are removed from the source fields before the values are transferred: ■ leading zeros before the decimal point for fields of format N, P or I ■ trailing zeros after the decimal point for fields of format N or P ■ trailing blanks for fields of format A ■ and leading binary zeros for fields of format B For a numeric source field containing all zeros, one zero will be transferred. For example:

Syntax Element	Description

	`COMPRESS 'ABC '  001 INTO #TARGET WITH DELIMITER '*'`
	`Content of #TARGET is: ABC*1`
	
	With `FULL`, the values of the source fields in their actual lengths will be transferred to the target field. In other words:
	
	■ leading zeros before the decimal point for fields of format N, P or I
	■ trailing zeros after the decimal point for fields of format N or P
	■ and trailing blanks for fields of format A
	■ leading binary zeros for fields of format B
	
	are displayed as entered. For example:
	
	`COMPRESS FULL 'ABC '  001 INTO #TARGET WITH DELIMITER '*'`
	`Content of #TARGET is: ABC *001`

operand1	**Source Fields:** As *operand1*, you specify the fields whose contents are to be transferred. **Note:** If *operand1* is not of format A or B, its content is converted into alphanumeric representation before it is transferred. If necessary, the alphanumeric representation is truncated. Using *operand1* without an explicit Edit Mask, a ... - Time variable (format T) is transferred only with the time component, not the date component. - Logical variable (format L) with value <false> is represented by a blank and value <true> is represented by char "X".
operand2	**Target Field:** As *operand2*, you specify the field which is to receive the values of the source fields. If the target field is of format U (Unicode) and if a source field of format B is involved, the length of the sending binary field must be even.
LEAVING SPACE	**Values in Target Field Separated by a Blank:** If you use the `COMPRESS` statement without any further options, or if you specify `LEAVING SPACE` (which also applies by default), the values in the target field will be separated from one another by a blank.
LEAVING NO SPACE	**Values in Target Field Not Separated:** If you specify `LEAVING NO SPACE`, the values in the target field will not be separated from one another by a blank or any other character.
parameter	**Print Mode/Date Format/Edit Mask Parameters:** As *parameter*, you can specify the session parameters `PM`, `DF`, `EM`, or `EMU`:

Syntax Element	Description	

	`PM=I`	In order to support languages whose writing direction is from right to left, you can specify `PM=I` so as to transfer the value of *`operand1`* in inverse (right-to-left) direction to *`operand2`*. For example, as a result of the following statements, the content of `#B` would be `ZYXABC`:
		
		`MOVE 'XYZ' TO #A`
		`COMPRESS #A (PM=I) 'ABC'`
		`INTO #B LEAVING NO SPACE`
		
		Any trailing blanks in *`operand1`* will be removed (except if `FULL` is specified), then the value is reversed character by character and transferred to *`operand2`*.

	`DF`	If *`operand1`* is a date variable, you can specify the session parameter `DF` as *`parameter`* for this variable.
	`EM=`	**Edit Mask:** For details on edit masks, see the session parameter `EM` in the Parameter Reference. The `EM` parameter cannot be applied for group operands or when the `SUBSTRING` option is used.
	`EMU=`	**Unicode Edit Mask:** For details on Unicode edit masks, see the session parameter `EMU` in the Parameter Reference. The `EMU` parameter cannot be applied for group operands or when the `SUBSTRING` option is used.
`SUBSTRING` (*`operand1`*, *`operand3`*, *`operand4`*)	**SUBSTRING Option:** If *`operand1`* is of alphanumeric (A), Unicode (U) or binary (B) format, you can use the `SUBSTRING` option to transfer only a certain part of a source field. After the field name (*`operand1`*) you specify first the starting position (*`operand3`*) and then the length (*`operand4`*) of the field portion to be transferred.	
`INTO SUBSTRING` (*`operand2`*, *`operand5`*, *`operand6`*)	**INTO Clause:** Also, you can use the `SUBSTRING` option in the `INTO` clause to transfer source values into a certain part of the target field. In both cases, the use of the `SUBSTRING` option in a `COMPRESS` statement corresponds to that in a `MOVE` statement. See the `MOVE` statement for details on the `SUBSTRING` option.	
`WITH DELIMITERS`	**Input Delimiter Character:** If you wish the values in the target field to be separated from one another by a specific character, you use the `DELIMITERS` option.	

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td></td><td>If you specify <code>WITH DELIMITERS</code> without <code>operand7</code>, the values will be separated by the input delimiter character as defined with the session parameter <code>ID</code>.</td></tr>
<tr><td><code>WITH DELIMITERS operand7</code></td><td>Specific Delimiter Character:
If you specify <code>WITH DELIMITERS operand7</code>, the values will be separated by the character specified with <code>operand7</code>. <code>operand7</code> must be a single character. If <code>operand7</code> is a variable, it must be of format/length (A1) or (B1).

If the target field is of format A or B, the format/length of the delimiter has to be (A1), (B1) or (U1).

If the target field is of format U (Unicode), the format/length of the delimiter has to be (A1), (B2) or (U1).</td></tr>
<tr><td><code>WITH ALL</code></td><td>Handling of Delimiters:

Without <code>ALL</code>, a delimiter is placed in the target field only between values actually transferred. For example:

<code>COMPRESS 'A' ' ' 'C' ' ' INTO #TARGET WITH DELIMITERS '*'</code>
<code>Content of #TARGET is: A*C</code>

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

<code>COMPRESS 'A' ' ' 'C' ' ' INTO #TARGET WITH ALL DELIMITERS '*'</code>
<code>Content of #TARGET is: A**C*</code></td></tr>
</table>

Processing

A destination field of format B is handled like a destination field of format A.

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

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

If *operand2* is a dynamic variable, the `COMPRESS` operation terminates when all source operands have been processed. No truncation will be performed. The length of *operand2* after the `COMPRESS` operation will correspond to the combined length of the source operands. The current length of a dynamic variable can be ascertained by using the system variable `*LENGTH`.

Examples

This section covers the following topics:

Example 1 - Compress

```
** Example 'CMPEX1': COMPRESS
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 MIDDLE-I
*
1 #COMPRESSED-NAME (A20)
END-DEFINE
*
LIMIT 4
READ EMPLOY-VIEW BY NAME
  COMPRESS FIRST-NAME MIDDLE-I NAME INTO #COMPRESSED-NAME
  DISPLAY NOTITLE
          FIRST-NAME MIDDLE-I NAME 5X #COMPRESSED-NAME
END-READ
*
END
```

Output of Program CMPEX1:

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

KEPA                          ABELLAN                  KEPA ABELLAN
ROBERT               W        ACHIESON                 ROBERT W ACHIESON
SIMONE                        ADAM                     SIMONE ADAM
JEFF                 H        ADKINSON                 JEFF H ADKINSON
```

Example 2 - Compress Leaving No Space

```
** Example 'CMPEX2': COMPRESS (with LEAVING NO SPACE)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CURR-CODE (1)
  2 SALARY    (1)
*
1 #CCSALARY   (A20)
END-DEFINE
*
LIMIT 4
READ EMPL-VIEW BY NAME

  COMPRESS CURR-CODE (1) SALARY (1) INTO #CCSALARY
           LEAVING NO SPACE
  DISPLAY NOTITLE
          NAME CURR-CODE (1) SALARY (1) 5X #CCSALARY
END-READ
*
END
```

Output of Program CMPEX2:

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


ABELLAN              PTA         1450000     PTA1450000
ACHIESON             UKL           11300     UKL11300
ADAM                 FRA          159980     FRA159980
ADKINSON             USD           34500     USD34500
```

Example 3 - Compress with Delimiter

```
** Example 'CMPEX3': COMPRESS (with delimiter)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CURR-CODE (1)
  2 SALARY    (1)
*
1 #CCSALARY   (A20)
END-DEFINE
*
LIMIT 4
READ EMPL-VIEW BY NAME
```

```
  COMPRESS CURR-CODE (1) SALARY (1) INTO #CCSALARY
           WITH DELIMITER '*'
  DISPLAY NOTITLE NAME CURR-CODE (1) SALARY (1) 5X #CCSALARY
END-READ
*
END
```

Output of Program CMPEX3:

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

ABELLAN              PTA         1450000     PTA*1450000
ACHIESON             UKL           11300     UKL*11300
ADAM                 FRA          159980     FRA*159980
ADKINSON             USD           34500     USD*34500
```

Example 4 - Compress with Edit Mask EM

```
** Example 'CMPEX4': COMPRESS (with edit mask EM)
************************************************************************
DEFINE DATA LOCAL
1 #A10     (A10)   INIT <'ABCDEF'>
1 #I4      (I4)    INIT <-123>
1 #T       (T)     INIT <E'2021-11-22 10:24:36'>
1 #L       (L)     INIT <TRUE>
1 #RESULT  (A70)
END-DEFINE
*
COMPRESS '#A:'  #A10  (EM=X_X_X)
         '#I4:' #I4   (EM=-999Z)
         '#T:'  #T    (EM=YYYY-MM-DD_HH:II)
         '#L:'  #L    (EM=FALSE/TRUE)       INTO #RESULT
PRINT #RESULT
END
```

Output of Program CMPEX4:

```
#A: A_B_C #I4: -0123 #T: 2021-11-22_10:24 #L: TRUE
```

33 COMPUTE

Structured Mode Syntax

```
{ { COMPUTE } [ROUNDED] {operand1 [:]= } ...   { arithmetic-expression             } }
{ { ASSIGN  }                                  { operand2                          } }
{                                              { SUBSTRING                         } }
{                                              { (operand2,operand3,operand4)      } }
{ {operand1 := } ...                           { arithmetic-expression             } }
{                                              { operand2                          } }
{                                              { SUBSTRING                         } }
{                                              { (operand2,operand3,operand4)      } }
```

Reporting Mode Syntax

```
[ { COMPUTE } [ROUNDED] ] {operand1 [:]= } ... { arithmetic-expression        }
[ { ASSIGN  }           ]                      { operand2                     }
                                               { SUBSTRING                    }
                                               { (operand2,operand3,operand4) }
```

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

Related Statements: ADD | COMPRESS | DIVIDE | EXAMINE | MOVE | MOVE ALL | MULTIPLY | RESET | SEPARATE | SUBTRACT

Belongs to Function Group: ***Arithmetic and Data Movement Operations***

Function

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

A COMPUTE statement with multiple target operands (*operand1*) is identical to the corresponding individual COMPUTE statements if the source operand (*operand2*) is not an arithmetic expression.

```
#TARGET1 := #TARGET2 := #SOURCE
```

is identical to

```
#TARGET1 := #SOURCE
#TARGET2 := #SOURCE
```

Example:

```
DEFINE DATA LOCAL
1 #ARRAY(I4/1:3) INIT <3,0,9>
1 #INDEX(I4)
1 #RESULT(I4)
END-DEFINE
*
#INDEX := 1
*
#INDEX :=       /* #INDEX is 3
#RESULT :=      /* #RESULT is 9
#ARRAY(#INDEX)
*
#INDEX := 2
*
#INDEX  :=      /* #INDEX is 0
#ARRAY(3) :=    /* returns runtime error NAT1316
#ARRAY(#INDEX)
END
```

If the source operand is an arithmetic expression, the expression is evaluated and its result is stored in a temporary variable. Then the temporary variable is assigned to the target operands.

```
#TARGET1 := #TARGET2 := #SOURCE1 + 1
is identical to
#TEMP := #SOURCE1 + 1
#TARGET1 := #TEMP
#TARGET2 := #TEMP
```

Example:

```
DEFINE DATA LOCAL
1 #ARRAY(I4/1:3) INIT <2, 0, 9>
1 #INDEX(I4)
1 #RESULT(I4)
END-DEFINE
*
#INDEX := 1
*
#INDEX :=   /* #INDEX is 3
#RESULT :=   /* #RESULT is 3
#ARRAY(#INDEX) + 1
*
#INDEX := 2
*
#INDEX  :=  /* #INDEX is 0
#ARRAY(3) :=   /* returns run time error NAT1316
#ARRAY(#INDEX)
END
```

For further information, see *Rules for Arithmetic Assignment* in the *Programming Guide* and particularly the following sections:

- *Arithmetic Operations with Arrays*
- *Data Transfer* (for information on data transfer compatibility and the rules for data transfer)

Syntax Description

Operand Definition Table:

Operand	Possible Structure						Possible Formats													Referencing Permitted	Dynamic Definition
operand1		S	A		M		A	U	N	P	I	F	B	D	T	L	C	G	O	yes	yes
operand2	C	S	A		N	E	A	U	N	P	I	F	B	D	T	L	C	G	O	yes	no
operand3	C	S							N	P	I		B*							yes	no
operand4	C	S							N	P	I		B*							yes	no

* If *operand3* or *operand4* is a binary variable, it may be used only with a length of less than or equal to 4.

Syntax Element Description:

Syntax Element	Description
`COMPUTE | ASSIGN [:]=`	**Usage of Keywords:** This statement may be issued in short form by omitting the statement keyword `COMPUTE` (or `ASSIGN`). In structured mode, when the statement keyword `COMPUTE` (or `ASSIGN`) is omitted, the equal sign (=) must be preceded by a colon (:). However, when the `ROUNDED` option is used, the statement keyword `COMPUTE` (or `ASSIGN`) must be specified.
`ROUNDED`	**ROUNDED Option:** If you specify the keyword `ROUNDED`, the value will be rounded before it is assigned to *operand1*. For information on rounding, see *Rules for Arithmetic Assignments*, *Field Truncation and Field Rounding* in the *Programming Guide*.
operand1	**Result Field:** *operand1* will contain the result of the arithmetic/assignment operation. For the precision of the result, see *Precision of Results of Arithmetic Operations* in the *Programming Guide*. If *operand1* is a database field, the field in the database is not updated.

Syntax Element	Description
	If `operand1` is a dynamic variable, it is filled with exactly the data and length of `operand2` or the length of the result of the arithmetic-operation, including trailing blanks. The current length of a dynamic variable can be obtained by using the system variable `*LENGTH`. For general information on dynamic variables, see *Using Dynamic and Large Variables*.
arithmetic-expression	**Arithmetic Expression:** An arithmetic expression consists of one or more constants, database fields, and user-defined variables. Natural mathematical functions (described in the *System Functions* documentation) may also be used as arithmetic operands. Operands used in an arithmetic expression must be defined with format N, P, I, F, D, or T. As for the formats of the operands, see also *Performance Considerations for Mixed Formats* in the *Programming Guide*. The following connecting operators may be used: **Operator:** / **Symbol:** Parentheses: `( )` Exponentiation: `**` Multiplication: `*` Division: `/` Addition: `+` Subtraction: `-` Each operator should be preceded and followed by at least one blank so as to avoid any conflict with a variable name that contains any of the above characters. The processing order of arithmetic operations is: 1. Parentheses 2. Exponentiation 3. Multiplication/division (left to right as detected) 4. Addition/subtraction (left to right as detected)
operand2	**Source Field:** `operand2` is the source field. If `operand1` is of format C, `operand2` may also be specified as an attribute constant. See *User-Defined Constants* in the *Programming Guide*.

Syntax Element	Description
SUBSTRING (*operand2*,*operand3*,*operand4*)	**SUBSTRING Option:** Without the substring option, the whole content of *operand2* is moved. If *operand1* and *operand2* are of alphanumeric, Unicode or binary format, you may use the SUBSTRING option to assign a certain part of *operand2* to *operand1*. After the field name (*operand2*) in the SUBSTRING clause, you specify the starting position (*operand3*) and then the length (*operand4*) of the field portion to be moved. For example, to assign the 3rd to 6th position of field #B to field #A, you would specify: `#A := SUBSTRING(#B,3,4)` If you omit *operand3*, the starting position is assumed to be 1. If you omit *operand4*, the length is assumed to range from the starting position to the end of the field. **Note:** ASSIGN with the SUBSTRING option is a byte-by-byte assignment (that is, the rules described under *Rules for Arithmetic Assignment* in the *Programming Guide* do *not* apply). See also MOVE SUBSTRING.

Result Precision of a Division

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

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

Examples

Example 1 - ASSIGN Statement

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

Output of Program ASGEX1S:

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

Equivalent reporting-mode example: **ASGEX1R**.

Example 2 - COMPUTE Statement

```
** Example 'CPTEX1': COMPUTE
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 SALARY    (1:2)
*
1 #A          (P4)
1 #B          (N3.4)
1 #C          (N3.4)
1 #CUM-SALARY (P10)
1 #I          (P2)
END-DEFINE
*
COMPUTE #A = 3 * 2 + 4 / 2 - 1
WRITE NOTITLE 'COMPUTE #A = 3 * 2 + 4 / 2 - 1' 10X '=' #A
*
COMPUTE ROUNDED #B = 3 -4 / 2 * .89
WRITE 'COMPUTE ROUNDED #B = 3 -4 / 2 * .89' 5X '=' #B
*
COMPUTE #C = SQRT (#B)
WRITE 'COMPUTE #C = SQRT (#B)' 18X '=' #C
*
LIMIT 1
READ EMPLOY-VIEW BY PERSONNEL-ID STARTING FROM '20017000'
  WRITE / 'CURRENT SALARY: '  4X SALARY (1)
        / 'PREVIOUS SALARY:'  4X SALARY (2)
  FOR #I = 1 TO 2
    COMPUTE #CUM-SALARY = #CUM-SALARY + SALARY (#I)
  END-FOR
  WRITE 'CUMULATIVE SALARY:' #CUM-SALARY
END-READ
*
END
```

Output of Program CPTEX1:

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

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

34 CREATE OBJECT

```
CREATE OBJECT operand1 OF [CLASS] operand2
  [GIVING operand3]
```

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

Related Statements: `DEFINE CLASS` | `INTERFACE` | `METHOD` | `PROPERTY` | `SEND METHOD`

Belongs to Function Group: *Component Based Programming*

Function

The `CREATE OBJECT` statement is used to create an instance of a class.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1		S																O	no	no
operand2	C	S				A													yes	no
operand3		S			N					I									yes	no

Syntax Element Description:

Syntax Element	Description
operand1	**Object Handle:** *operand1* must be defined as an object handle (`HANDLE OF OBJECT`). The object handle is filled when the object is successfully created. When not successfully returned, *operand1* contains the value `NULL-HANDLE`.
`OF CLASS` *operand2*	**Class-Name:** *operand2* is the name of the class of which the object is to be created. For classes that are not registered as DCOM classes, it must contain the class name defined in the `DEFINE CLASS` statement. For classes that are registered as DCOM classes, it must contain either the `ProgID` of the class or the class GUID. For Natural classes that are registered as DCOM classes, the `ProgID` corresponds to the class name specified in the `DEFINE CLASS` statement.

Syntax Element	Description
	`CREATE OBJECT #O1 OF CLASS "Employee" or` `CREATE OBJECT #O1 OF CLASS "653BCFE0-84DA-11D0-BEB3-10005A66D231"`
`GIVING` *`operand3`*	**GIVING Clause:** If this clause is specified, *`operand3`* contains either the Natural message number if an error occurred, or zero on success. If this clause is not specified, Natural run time error processing is triggered if an error occurs.

35 DECIDE FOR

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

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

Related Statements: DECIDE ON | IF | IF SELECTION | ON ERROR

Belongs to Function Group: ***Processing of Logical Conditions***

Function

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

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

Syntax Description

Syntax Element	Description
FIRST CONDITION	**Processing of First Condition Only:** Only the first true condition is to be processed. See also ***Example 1***.
EVERY CONDITION	**Processing of Every Condition:** Every true condition is to be processed. See also ***Example 2***.
WHEN *logical-condition statement*	**Logical Condition(s) to be Processed:** With this clause, you specify the logical condition(s) to be processed. See the section *Logical Condition Criteria* in the *Programming Guide*.
WHEN ANY *statement*	**WHEN ANY Clause:**

Syntax Element	Description
	With WHEN ANY, you can specify the **statement(s)** to be executed when any of the logical conditions are true.
WHEN ALL *statement*	**WHEN ALL Clause:** With WHEN ALL, you can specify the **statement (s)** to be executed when all logical conditions are true. This clause is applicable only if EVERY has been specified.
WHEN NONE *statement*	**WHEN NONE Clause:** With WHEN NONE, you specify the statement(s) to be executed when none of the logical conditions are true.
END-DECIDE	**End of DECIDE FOR Statement:** The Natural reserved word END-DECIDE must be used to end the DECIDE FOR statement.

Examples

- Example 1 - DECIDE FOR with FIRST Option
- Example 2 - DECIDE FOR with EVERY Option

Example 1 - DECIDE FOR with FIRST Option

```
** Example 'DECEX1': DECIDE FOR (with FIRST option)
************************************************************************
DEFINE DATA LOCAL
1 #FUNCTION (A1)
1 #PARM     (A1)
END-DEFINE
*
INPUT #FUNCTION #PARM
*
DECIDE FOR FIRST CONDITION
  WHEN #FUNCTION = 'A' AND #PARM = 'X'
    WRITE 'Function A with parameter X selected.'
  WHEN #FUNCTION = 'B' AND #PARM = 'X'
    WRITE 'Function B with parameter X selected.'
  WHEN #FUNCTION = 'C' THRU 'D'
    WRITE 'Function C or D selected.'
  WHEN NONE
    REINPUT 'Please enter a valid function.'
            MARK *#FUNCTION
END-DECIDE
```

```
*
END
```

Output of Program DECEX1:

```
#FUNCTION   #PARM
```

After entering A and Y and pressing ENTER:

```
#FUNCTION A #PARM Y

Please enter a valid function.
```

Example 2 - DECIDE FOR with EVERY Option

```
** Example 'DECEX2': DECIDE FOR (with EVERY option)
************************************************************************
DEFINE DATA LOCAL
1 #FIELD1 (N5.4)
END-DEFINE
*
INPUT #FIELD1
*
DECIDE FOR EVERY CONDITION
  WHEN #FIELD1 >= 0
    WRITE '#FIELD1 is positive or zero.'
  WHEN #FIELD1 <= 0
    WRITE '#FIELD1 is negative or zero.'
  WHEN FRAC(#FIELD1) = 0
    WRITE '#FIELD1 has no decimal digits.'
  WHEN ANY
    WRITE 'Any of the above conditions is true.'
  WHEN ALL
    WRITE '#FIELD1 is zero.'
  WHEN NONE
    IGNORE
END-DECIDE
*
END
```

Output of Program DECEX2:

```
#FIELD1 42
```

After pressing ENTER:

```
Page      1                                                   05-01-11  14:56:26

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

36 DECIDE ON

```
DECIDE ON
{ FIRST } [VALUE]      { op1              }
{ EVERY } [OF]         { SUBSTR           }
                       { (op3,op5,op6)    }

{ VALUE { op2           } ,  [ : { op2           } ] statement }
{       { SUBSTR        } ...    { SUBSTR        }   ...       }
{       { (op4,op7,op8) }        { (op4,op7,op8) }             } ...
[ANY [VALUE] statement ...]
[ALL [VALUE] statement ...]
NONE [VALUE] statement ...
END-DECIDE
```

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

Related Statements: DECIDE FOR | IF | IF SELECTION | ON ERROR

Belongs to Function Group: *Processing of Logical Conditions*

Function

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

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

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
op1		S	A		N	A	U	N	P	I	F	B	D	T	L		G	O	yes	no
op2	C	S	A			A	U	N	P	I	F	B	D	T	L		G	O	yes	no
op3		S	A			A	U					B							yes	no
op4	C	S	A			A	U					B							yes	no
op5	C	S						N	P	I		B *							yes	no
op6	C	S						N	P	I		B *							yes	no
op7	C	S						N	P	I		B *							yes	no

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
op8	C	S						N	P	I		B *							yes	no

* Format B of `op5`, `op6`, `op7` and `op8` may be used only with a length of less than or equal to 4.

Syntax Element Description:

Syntax Element	Description
`FIRST/EVERY`	**Processing of Values:** With one of these keywords, you indicate whether only the first or every value that is found is to be processed.
op1	**Selection Field:** As *op1* or *op2* you specify the name of the field whose content is to be checked.
`VALUES` *op2* `[[,`*op2*`] ... [:`*op2*`]`*statement* `...`	**VALUES Clause:** With this clause, you specify the value (*op2*) of the selection field, as well as the *statement(s)* which are to be executed if the field contains that value. You can specify one value, multiple values, or a range of values optionally preceded by one or more values. Multiple values must be separated from one another either by the input delimiter character (as specified with the session parameter `ID`) or by a comma. A comma must not be used for this purpose, however, if the comma is defined as decimal character (with the session parameter `DC`). For a range of values, you specify the starting value and ending value of the range, separated from each other by a colon.
`SUBSTRING`	**SUBSTRING Option:** Without the `SUBSTRING` option, the whole content of a field is checked. The `SUBSTRING` option allows you to check only a certain part of an alphanumeric, Unicode or binary field. After the field name (*op3*), you specify first the starting position (*op5*) and then the length (*op6*) of the field portion to be checked.
`SUBSTRING (`*op4*`,`*op7*`,`*op8*`)`	**SUBSTRING Option:** After the field name (*op4*), you specify first the starting position (*op7*) and then the length (*op8*) of the field portion to be checked.
`ANY` *statement*	**ANY Clause:** With `ANY`, you specify the *statement(s)* which are to be executed if any of the values in the `VALUES` clause are found. These statements are to be executed in addition to the statement specified in the `VALUES` clause.

Syntax Element	Description
`ALL statement`	**ALL Clause:** With `ALL`, you specify the *statement(s)* which are to be executed if all of the values in the `VALUES` clause are found. These statements are to be executed in addition to the statement specified in the `VALUES` clause. The `ALL` clause applies only if the keyword `EVERY` is specified.
`NONE statement`	**NONE Clause:** With `NONE`, you specify the *statement(s)* which are to be executed if none of the specified values are found.
`END-DECIDE`	**End of DECIDE ON Statement:** The Natural reserved word `END-DECIDE` must be used to end the `DECIDE ON` statement.

Examples

- Example 1 - DECIDE ON with FIRST Option
- Example 2 - DECIDE ON with EVERY Option

Example 1 - DECIDE ON with FIRST Option

```
** Example 'DECEX3': DECIDE ON (with FIRST option)
************************************************************************
*
SET KEY ALL
INPUT 'Enter any PF key' /
      'and check result' /
*
DECIDE ON FIRST VALUE OF *PF-KEY
  VALUE 'PF1'
    WRITE 'PF1 key entered.'
  VALUE 'PF2'
    WRITE 'PF2 key entered.'
  ANY VALUE
    WRITE 'PF1 or PF2 key entered.'
  NONE VALUE
    WRITE 'Neither PF1 nor PF2 key entered.'
END-DECIDE
*
END
```

Output of Program DECEX3:

```
Enter any PF key
and check result
```

Output after pressing PF1:

```
Page      1                                                   05-01-11  15:08:50

PF1 key entered.
PF1 or PF2 key entered.
```

Example 2 - DECIDE ON with EVERY Option

```
** Example 'DECEX4': DECIDE ON (with EVERY option)
************************************************************************
DEFINE DATA LOCAL
1 #FIELD (N1)
END-DEFINE
*
INPUT 'Enter any value between 1 and 9:' #FIELD (SG=OFF)
*
DECIDE ON EVERY VALUE OF #FIELD
  VALUE 1 : 4
    WRITE 'Content of #FIELD is 1-4'
  VALUE 2 : 5
    WRITE 'Content of #FIELD is 2-5'
  ANY VALUE
    WRITE 'Content of #FIELD is 1-5'
  ALL VALUE
    WRITE 'Content of #FIELD is 2-4'
  NONE VALUE
    WRITE 'Content of #FIELD is not 1-5'
    END-DECIDE
*
END
```

Output of Program DECEX4:

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

After entering and confirming 4**:**

```
Page      1                                                     05-01-11  15:11:45

Content of #FIELD is 1-4
Content of #FIELD is 2-5
Content of #FIELD is 1-5
Content of #FIELD is 2-4
```

37 DEFINE CLASS

```
DEFINE CLASS class-name
[ OBJECT    { USING { local-data-area     } } ]
            {       { parameter-data-area } } ] ...
            { local-data-definition ...     } ]
[ LOCAL     { USING { local-data-area     } } ]
            {       { parameter-data-area } } ] ...
            { local-data-definition ...     } ]
[ INTERFACE USING
  copycode
  interface-statement ] ...
[property-statement] ...
[method-statement] ...
END-CLASS
```

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

Related Statements: `CREATE OBJECT` | `INTERFACE` | `METHOD` | `PROPERTY` | `SEND METHOD`

Belongs to Function Group: ***Component Based Programming***

Function

The `DEFINE CLASS` statement is used to specify a class from within a Natural class module. A Natural class module consists of one `DEFINE CLASS` statement followed by an `END` statement.

Syntax Description

Syntax Element	Description
class-name	**Class Name:** This is the name that is used by clients to create objects of this class. The name can be up to a maximum of 32 characters long. The name may contain periods: this can be used to construct class names such as *company-name.application-name.class-name* Each part between the periods (...) must conform to the *Naming Conventions for User-Defined Variables*.

Syntax Element	Description
	If the class is planned to be used by clients written in different programming languages, the class name should be chosen in a way that it does not conflict with the naming conventions that apply in these languages.
`OBJECT`	**OBJECT Clause:** This clause is used to define the object data. The syntax of the `OBJECT` clause is the same as for the `LOCAL` clause of the `DEFINE DATA` statement. For further information, see the description of the `LOCAL` clause of the `DEFINE DATA` statement.
`LOCAL`	**LOCAL Clause:** This clause is only used to include globally unique IDs (GUIDs) in the class definition. GUIDs need only be defined if a class is to be registered with DCOM. GUIDs are mostly defined in a local data area. The syntax of the `LOCAL` clause is the same as for the `LOCAL` clause of the `DEFINE DATA` statement. For further information, see the description of the `LOCAL` clause of the `DEFINE DATA` statement.
`INTERFACE USING`	**INTERFACE USING Clause:** This clause is used to include copycode that contains `INTERFACE` statements.
copycode	**Copycode:** The copycode used by the `INTERFACE USING` clause may contain one or more `INTERFACE` statements.
interface-statement	**INTERFACE Statement:** The `INTERFACE` statement is used to define methods and properties for a class.
property-statement	**PROPERTY Statement:** The `PROPERTY` statement is used to assign an object data variable operand as the implementation to a property, outside an interface definition.
method-statement	**METHOD Statement:** The `METHOD` statement is used to assign a subprogram as the implementation to a method, outside an interface definition.
`END-CLASS`	**End of DEFINE CLASS Statement:** The Natural reserved word `END-CLASS` must be used to end the `DEFINE CLASS` statement.

VI DEFINE DATA

```
DEFINE DATA
[GLOBAL USING global-data-area [WITH block [.block] ...]]
[ PARAMETER [ USING parameter-data-area             ] ] ...
              parameter-data-definition ...
[ LOCAL     [ USING { local-data-area     }         ] ] ...
                    { parameter-data-area }
              local-data-definition ...
[INDEPENDENT [aiv-data-definition ...]] ...
[ CONTEXT   [ USING { local-data-area     }         ] ] ...
                    { parameter-data-area }
              context-data-definition ...
[ OBJECT    [ USING { local-data-area     }         ] ] ...
                    { parameter-data-area }
              local-data-definition ...
END-DEFINE
```

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

Related topics in the *Programming Guide*: *Use and Structure of DEFINE DATA Statement* | *Data Areas*

The `DEFINE DATA` documentation is organized under the following headings:

- **Function and Basic Syntax Rules**

Data Definitions:

- **Defining Global Data**
- **Defining Parameter Data**
- **Defining Local Data**

- **Defining Application-Independent Variables**
- **Defining Context Variables for Natural RPC**
- **Defining NaturalX Objects**

Clauses and Options:

- **Variable Definition**
- **View Definition**
- **Redefinition**
- **Array Dimension Definition**
- **Initial-Value Definition**
- **Initial/Constant Values for an Array**
- **EM, HD, PD Parameters for Field/Variable**

Examples:

- **Examples of DEFINE DATA Statement Usage**

38 Function and Basic Syntax Rules

Function

The `DEFINE DATA` statement offers a number of clauses to declare data definitions for use within a Natural program, either by referencing predefined data definitions contained in a local data area (LDA), global data area (GDA) or parameter data area (PDA), or by writing in-line definitions.

General Syntax Rules

- When a `DEFINE DATA` statement is used, it must be the first statement of the program/routine.
- An "empty" `DEFINE DATA` statement is not allowed; at least one clause (`GLOBAL`, `PARAMETER`, `LOCAL`, `INDEPENDENT`, `CONTEXT` or `OBJECT`) must be specified.
- You can specify more than one clause. However, if the `GLOBAL` and the `PARAMETER` clauses are used, `GLOBAL` must be the first clause of the statement and `PARAMETER` must follow `GLOBAL` (without `GLOBAL`, `PARAMETER` comes first if used). All other clauses can be specified in any order.
- The Natural reserved word `END-DEFINE` must be used to end the `DEFINE DATA` statement.

Programming Modes

The `DEFINE DATA` statement is available in structured mode and in reporting mode. Differences are marked accordingly in the `DEFINE DATA` statement description.

Generally, the following applies:

- Structured Mode
- Reporting Mode

Structured Mode

All variables to be used, except **application-independent variables** (AIVs), must be defined in the `DEFINE DATA` statement; they must not be defined elsewhere in the program. If a `DEFINE DATA INDEPENDENT` statement is used, AIVs must not be defined elsewhere in the program.

Reporting Mode

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

39 Defining Global Data

General syntax of `DEFINE DATA GLOBAL`:

```
DEFINE DATA
  GLOBAL USING global-data-area [WITH block[.block...]]
END-DEFINE
```

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

Function

The `DEFINE DATA GLOBAL` statement is used to define data elements using a GDA (see Global Data Area).

Syntax Description

Syntax Element	Description
`USING` *`global-data-area`*	**GDA Name:** Specify the name of a global data area (GDA) to be referenced. A GDA is created with the source editor. It contains predefined data elements which can be included in the `DEFINE DATA GLOBAL` statement. In contrast to an LDA, the data elements defined in a GDA can be referenced by more than one Natural object. For further information, see Global Data Area in the *Programming Guide*.
`WITH` *`block`*	**Data Blocks:** To save data storage space, you can create a global data area with data blocks. Data blocks can overlay one another during program execution, thereby saving storage space. The maximum number of block levels is 8 (including the master block). For further information, see *Data Blocks* in the *Programming Guide*.
`.block`	**Block(s) to be Used:** A single or multiple *`.block`* notations specify the block(s) which are used in the program.
`END-DEFINE`	**End of DEFINE DATA Statement:**

Syntax Element	Description
	The Natural reserved word `END-DEFINE` must be used to end the `DEFINE DATA` statement.

40 Defining Parameter Data

General syntax of DEFINE DATA PARAMETER:

```
DEFINE DATA
   PARAMETER [ USING parameter-data-area          ]
             [ parameter-data-definition ...      ]
END-DEFINE
```

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

Function

The DEFINE DATA PARAMETER statement is used to define the data elements that are to be used as incoming parameters in a Natural subprogram, external subroutine, helproutine or function. These parameters can be defined within the statement itself (see *Parameter Data Definition*); or they can be defined outside the program in a **parameter data area** (PDA), with the statement referencing that data area.

Restrictions

- Parameter data elements must not be assigned initial or constant values, and they must not have edit mask (EM), header (HD) or print mode (PM) definitions; see also *EM, HD, PM Parameters for Field/Variable*.
- The parameter data area and the objects which reference it must be contained in the same library (or in a steplib).

Syntax Description

Syntax Element	Description
USING *parameter-data-area*	**Parameter Data Area (PDA) Name:** The name of the *parameter-data-area* (PDA) that contains data elements which are used as parameters in a subprogram, external subroutine or dialog.
parameter-data-definition	**Parameter Data Definition:** Instead of using a PDA, you can define parameter data directly. See *Parameter Data Definition*.

Syntax Element	Description
END-DEFINE	**End of DEFINE DATA Statement:** The Natural reserved word END-DEFINE must be used to end the DEFINE DATA statement.

Parameter Data Definition

For parameter data definition, the following syntax applies:

```
        { group-name [(array-definition)]                                                              }
        { redefinition                                                                                 }
        {                   { (format-length[/array-definition])                          }            }
{ level {                   {     { A }                                                   } [BY VALUE  }
        { variable-name     { (   { U }    [/array-definition])         DYNAMIC           } [RESULT]]  }
        {                   {     { B }                                                   } [OPTIONAL] }
        {                   { [(array-definition)] HANDLE OF OBJECT                       }            }
```

Syntax Element Description:

Syntax Element	Description
level	**Level Number:** Level number is a 1- or 2-digit number in the range from 01 to 99 (the leading zero is optional) used in conjunction with field grouping. Fields assigned a level number of 02 or greater are considered to be a part of the immediately preceding group which has been assigned a lower level number. The definition of a group enables reference to a series of fields (may also be only 1 field) by using the group name. With certain statements (CALL, CALLNAT, RESET, WRITE, etc.), you may specify the group name as a shortcut to reference the fields contained in the group. A group may consist of other groups. When assigning the level numbers for a group, no level numbers may be skipped.
group-name	**Group Name:** The name of a group. The name must adhere to the rules for defining a Natural variable name. See also the following sections: ■ *Naming Conventions for User-Defined Variables* in *Using Natural*. ■ *Qualifying Data Structures* in the *Programming Guide*.
array-definition	**Array Dimension Definition:**

Syntax Element	Description
	With an *array-definition*, you define the lower and upper bounds of dimensions in an array-definition. For further information, see ***Array Dimension Definition*** and ***Variable Arrays in a Parameter Data Area***.
redefinition	**Redefinition:** A *redefinition* may be used to redefine a group or a single field/variable (that is a scalar or an array). See ***Redefinition***. **Note:** In a *parameter-data-definition*, a redefinition of groups is only permitted within a REDEFINE block.
variable-name	**Variable Name:** The name to be assigned to the variable. Rules for Natural variable names apply. For information on naming conventions for user-defined variables. For further information, see *Naming Conventions for User-Defined Variables* in *Using Natural*.
format-length	**Format/Length Definition:** The format and length of the field. For information on format/length definition of user-defined variables, see *Format and Length of User-Defined Variables* in the *Programming Guide*.
HANDLE OF OBJECT	**Handle of Object:** Used in conjunction with NaturalX. A handle identifies a dialog element in code and is stored in handle variables. For further information, see *NaturalX* in the *Programming Guide*.
A, U or B	**Data Type:** Alphanumeric (A), Unicode (U) or binary (B) for dynamic variable.
DYNAMIC	**DYNAMIC Option:** A parameter may be defined as DYNAMIC. For further information on processing dynamic variables, see *Introduction to Dynamic Variables and Fields* in the *Programming Guide*.
	Call Mode: Depending on whether call-by-reference, call-by-value or call-by-value-result is used, the appropriate transfer mechanism is applicable. For further information, see the CALLNAT statement.
(without BY VALUE)	**Call-by-Reference:** Call-by-reference is active by default when you omit the BY VALUE keywords. In this case, a parameter is passed to a subprogram/subroutine/function by reference (that

<table>
<tr><th>Syntax Element</th><th colspan="2">Description</th></tr>
<tr><td></td><td colspan="2">is, via its address); therefore a field specified as parameter in a <code>CALLNAT</code>/<code>PERFORM</code> statement must have the same format/length as the corresponding field in the invoked subprogram/subroutine/function.</td></tr>
<tr><td rowspan="3"><code>BY VALUE</code></td><td colspan="2">Call-by-Value:

When you specify <code>BY VALUE</code>, a parameter is passed to a subprogram/subroutine/function by value; that is, the actual parameter value (instead of its address) is passed. Consequently, the field in the subprogram/subroutine/function need not have the same format/length as the parameter passed in the <code>CALLNAT</code>/<code>PERFORM</code> statement or in the function call. The formats/lengths must only be data transfer compatible. For data transfer compatibility, the Rules for Arithmetic Assignment and Data Transfer apply (see Programming Guide).

<code>BY VALUE</code> allows you, for example, to increase the length of a field in a subprogram/subroutine/function (if this should become necessary due to an enhancement of the subprogram/subroutine) without having to adjust any of the objects that invoke the subprogram/subroutine/function.</td></tr>
<tr><td colspan="2">Example of <code>BY VALUE</code>:</td></tr>
<tr><td><pre>* Program
DEFINE DATA LOCAL
1 #FIELDA (P5)
...
END-DEFINE
...
CALLNAT 'SUBR01' #FIELDA...</pre></td><td><pre>* Subroutine SUBR01
DEFINE DATA PARAMETER
1 #FIELDB (P9) BY VALUE
END-DEFINE
...</pre></td></tr>
<tr><td><code>BY VALUE RESULT</code></td><td colspan="2">Call-by-Value-Result:

While <code>BY VALUE</code> applies to a parameter passed to a subprogram/subroutine/function, <code>BY VALUE RESULT</code> causes the parameter to be passed by value in both directions; that is, the actual parameter value is passed from the invoking object to the subprogram/subroutine/function and, on return to the invoking object, the actual parameter value is passed from the subprogram/subroutine/function back to the invoking object.

With <code>BY VALUE RESULT</code>, the formats/lengths of the fields concerned must be data transfer compatible in both directions.</td></tr>
<tr><td><code>OPTIONAL</code></td><td colspan="2">Optional Parameters:

For a parameter defined without <code>OPTIONAL</code> (default), a value must be passed from the invoking object.

For a parameter defined with <code>OPTIONAL</code>, a value can, but need not be passed from the invoking object to this parameter.

In the invoking object, the notation n<code>X</code> is used to indicate parameters which are skipped, that is, for which no values are passed.</td></tr>
</table>

Syntax Element	Description
	With the `SPECIFIED` option you can find out at run time whether an optional parameter has been defined or not.

41 Defining Local Data

General syntax of `DEFINE DATA LOCAL`:

```
DEFINE DATA
  LOCAL [ USING { local-data-area      }  ]
                { parameter-data-area  }
          local-data-definition ...
END-DEFINE
```

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

Function

The `DEFINE DATA LOCAL` statement is used to define the data elements that are to be used exclusively by a single Natural module in an application. These elements or fields can be defined in different ways:

- either within the `DEFINE DATA LOCAL` statement itself, using the *`local-data-definition`* syntax (see ***Local Data Definition***)
- or outside the program in a separate LDA (*Local Data Area*) or PDA (*Parameter Data Area*), with the `DEFINE DATA LOCAL USING` statement referencing that data area.

Restriction

The LDA and the objects which reference it must be contained in the same library (or in a steplib).

Syntax Description

Syntax Element	Description
`local-data-area`	**LDA Name:** Specify the name of the local data area (LDA) to be referenced. An LDA is created with the source editor. It contains predefined data elements which can be included in the `DEFINE DATA LOCAL` statement. You may reference more than one data area; in that case you have to repeat the reserved words `LOCAL` and `USING`, for example:

Syntax Element	Description

	`DEFINE DATA LOCAL`
	`  LOCAL USING DATX_L`
	`  LOCAL USING DATX_P`
	
	`  ...`
	`END-DEFINE ;`
	
	For further information, see also *Defining Fields in a Separate Data Area* and *Local Data Area, Example 2* in the *Programming Guide*.

parameter-data-area	**PDA Name:** Specify the name of a parameter data area (PDA). **Note:** A data area referenced with DEFINE DATA LOCAL may also be a parameter data area (PDA). 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. A PDA is created with the source editor. It contains predefined data elements which can be included in the DEFINE DATA LOCAL statement. For further information, see *Parameter Data Area* in the *Programming Guide*.
local-data-definition	**Local Data Definition:** For information on how to define elements or fields within the statement itself, that is, without using an LDA or PDA, see the section ***Local Data Definition*** below.
END-DEFINE	**End of DEFINE DATA Statement:** The Natural reserved word END-DEFINE must be used to end the DEFINE DATA statement.

Local Data Definition

Local data can be defined directly. For local data definition, the following syntax applies:

$$\left\{ \; level \; \left\{ \begin{array}{l} group\text{-}name \; [(array\text{-}definition)] \\ variable\text{-}definition \\ view\text{-}definition \\ redefinition \end{array} \right\} \right\}$$

For further information, see

- ***Example 1 - DEFINE DATA LOCAL (Local Data Definition)***
- *Defining Fields within a DEFINE DATA Statement* in the *Programming Guide*
- *Local Data Area, Example 1* in the *Programming Guide*

Syntax Element Description for Local Data Definition:

Syntax Element	Description
level	**Level Number:** Level number is a 1- or 2-digit number in the range from 01 to 99 (the leading zero is optional) used in conjunction with field grouping. Fields assigned a level number of 02 or greater are considered to be a part of the immediately preceding group which has been assigned a lower level number. The definition of a group enables reference to a series of fields (may also be only 1 field) by using the group name. With certain statements (`CALL`, `CALLNAT`, `RESET`, `WRITE`, etc.), you may specify the group name as a shortcut to reference the fields contained in the group. A group may consist of other groups. When assigning the level numbers for a group, no level numbers may be skipped. A view-definition must always be defined at Level 1.
group-name	**Group Name:** The name of a group. The name must adhere to the rules for defining a Natural variable name. See also the following sections: ■ *Naming Conventions for User-Defined Variables* in *Using Natural*. ■ *Qualifying Data Structures* in the *Programming Guide*.
array-definition	**Array Dimension Definition:** With an *array-definition*, you define the lower and upper bounds of dimensions in an array-definition. See ***Array Dimension Definition***.
variable-definition	**Variable Definition:** A *variable-definition* is used to define a single field/variable that may be single-valued (scalar) or multi-valued (array). See ***Variable Definition***.
view-definition	**View Definition:** A *view-definition* is used to define a view as derived from a data definition module (DDM). See ***View Definition***.
redefinition	**Redefinition:**

Syntax Element	Description
	A *redefinition* may be used to redefine a group, a view, a DDM field or a single field/variable (that is a scalar or an array). See ***Redefinition***.

42 Defining Application-Independent Variables

General syntax of DEFINE DATA INDEPENDENT:

```
DEFINE DATA
  INDEPENDENT [aiv-data-definition ...]
END-DEFINE
```

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

Function

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

An application-independent variable is referenced by its name, and its content is shared by all Natural objects executed within one application that refer to that name. The variable is allocated by the first executed Natural object that references this variable and is deallocated by the LOGON command or a RELEASE VARIABLES statement.

The optional INIT clause is evaluated in each executed Natural object that contains this clause (not only in the Natural object that allocates the variable).

Note: In an RPC server, application-independent variables (AIVs) are not deallocated implicitly, but stay active across RPC requests, because different clients may have access to the same variables on the RPC server. This means they must be deallocated explicitly using the RELEASE VARIABLES statement. See *Application-Independent Variables* in the *Natural RPC (Remote Procedure Call)* documentation.

Syntax Description

Syntax Element	Description
aiv-data-definition	**AIV Data Definition:** The DEFINE DATA INDEPENDENT statement can be used to define a single or multiple application-independent variables (AIVs). For each AIV, the syntax shown in ***AIV Data Definition*** applies.
END-DEFINE	**End of DEFINE DATA Statement:** The Natural reserved word END-DEFINE must be used to end the DEFINE DATA statement.

AIV Data Definition

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

Syntax Element Description:

Syntax Element	Description
level	**Level Number:** An application-independent variable must be defined at Level 01. Other levels are only used in a redefinition.
variable-definition	**Variable Definition** A *variable definition* is used to define a single field/variable that may be single-valued (scalar) or multi-valued (array). For further information, see ***Variable Definition***. **Note:** The name of an application-independent variable must start with a plus (+) character.
redefinition	**Redefinition:** With a *redefinition*, you can partition an application-independent variable into one or more subfields. For further information, see ***Redefinition***. The subfields resulting from the redefinition must not be application-independent variables; that is, their name must not start with a plus sign (+). These fields are treated as local variables.

Note: The first character of the name must be a plus (+). Rules for Natural variable names apply, see *Naming Conventions for User-Defined Variables* in *Using Natural*.

43 Defining Context Variables for Natural RPC

General syntax of DEFINE DATA CONTEXT:

```
DEFINE DATA
            [ USING  { local-data-area     }  ]
  CONTEXT   [        { parameter-data-area }  ]
            [                                 ]
            [ context-data-definition ...     ]
END-DEFINE
```

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

Belongs to Function Group: ***Natural Remote Procedure Call***

Function

The DEFINE DATA CONTEXT statement is used in conjunction with the Natural RPC (Remote Procedure Call). It is used to define variables known as context variables, which are meant to be available to multiple remote subprograms within one conversation, without having to explicitly pass the variables as parameters with the corresponding CALLNAT statements.

A context variable is referenced by its name, and its content is shared by all Natural objects executed in one conversation that refer to that name. The variable is allocated by the first executed Natural object that contains the definition of the variable and is deallocated when the conversation ends.

A context variable is not shared with subprograms that are called within the conversation. If such a subprogram or one of its callees references a context variable, a separate storage area is allocated for this variable.

Context variables can also be used in a non-conversational CALLNAT. In this case, the context variables only exist during a single invocation of this CALLNAT. The variable is allocated when the remote subprogram is started and deallocated when it ends. The content is shared by all Natural objects except subprograms executed by this non-conversational CALLNAT.

The optional INIT clause is evaluated in each executed Natural object that contains this clause (not only in the Natural object that allocates the variable). This is different to the way the INIT works for global variables.

For further information, see *Defining a Conversation Context* in the *Natural RPC (Remote Procedure Call)* documentation.

Restrictions

A context variable must be defined at Level 01. Other levels are only used in a redefinition.

Syntax Description

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td><code>USING</code> local-data-area</td><td>LDA Name:

A local data area (LDA) contains data elements which are to be used in a single Natural module. You may reference more than one data area; in that case you have to repeat the reserved words <code>CONTEXT</code> and <code>USING</code>, for example:

<pre>DEFINE DATA
  CONTEXT USING DATX_L
  CONTEXT USING DATX_P

  ...
END-DEFINE ;</pre>
For further information, see Defining Fields in a Separate Data Area in the Programming Guide.</td></tr>
<tr><td><code>USING</code> parameter-data-area</td><td>PDA Name:

A parameter data area contains data elements which are used as parameters in a subprogram, external subroutine or dialog.</td></tr>
<tr><td>context-data-definition</td><td>Context Data Definition:

Context data can be defined directly within a program or routine. For context data definition, the syntax shown below applies.</td></tr>
<tr><td><code>END-DEFINE</code></td><td>End of DEFINE DATA Statement:

The Natural reserved word <code>END-DEFINE</code> must be used to end the <code>DEFINE DATA</code> statement.</td></tr>
</table>

Context Data Definition

Context data can be defined directly within a program or routine. For context data definition, the following syntax applies:

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

For further information, see *Defining Fields within a DEFINE DATA Statement* in the *Programming Guide*.

Syntax Element	Description
level	**Level Number:** An application-independent variable must be defined at Level 01. Other levels are only used in a redefinition.
variable-definition	**Variable Definition:** A *variable-definition* is used to define a single field/variable that may be single-valued (scalar) or multi-valued (array). For further information, see ***Variable Definition***. **Note:** The `CONSTANT` clause must not be used in this context
redefinition	**Redefinition:** A *redefinition* may be used to redefine a group, a view, a DDM field or a single field/variable (that is a scalar or an array). For further information, see ***Redefinition***.

Note: The fields resulting from the redefinition are not considered a context variable. These fields are treated as local variables.

44 Defining NaturalX Objects

General syntax of `DEFINE DATA OBJECT`:

```
DEFINE DATA
 OBJECT   [ USING { local-data-area      } ]
          [       { parameter-data-area  } ]
          [ local-data-definition ...        ]
END-DEFINE
```

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

Function

The `DEFINE DATA OBJECT` statement is used in a subprogram or class in conjunction with NaturalX.

Syntax Description

Syntax Element	Description

`USING` *local-data-area*	**LDA Name:**
	
	A local data area (LDA) contains data elements which are to be used in a single Natural module. You may reference more than one data area; in that case you have to repeat the reserved words `OBJECT` and `USING`, for example:
	
	`DEFINE DATA`
	`  OBJECT USING DATX_L`
	`  OBJECT USING DATX_P`
	
	`  ...`
	`END-DEFINE ;`
	
	For further information, see also *Defining Fields in a Separate Data Area* in the *Programming Guide*.

`USING` *parameter-data-area*	**PDA Name:** A data area defined with `DEFINE DATA OBJECT` may be a parameter data area (PDA). By using a PDA as an object data area you can avoid the extra effort of creating an object data area that has the same structure as the PDA.
local-data-definition	**Local Data Definition:** Data can also be defined directly using the syntax shown in ***Local Data Definition*** in the section *Defining Local Data*.
`END-DEFINE`	**End of DEFINE DATA Statement:**

Syntax Element	Description
	The Natural reserved word `END-DEFINE` must be used to end the `DEFINE DATA` statement.

45 Variable Definition

```
{ scalar-definition }
{ array-definition  }
```

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

The `variable-definition` option is used to define a single field/variable that may be single-valued (`scalar-definition`) or multi-valued (`array-definition`).

`scalar-definition`

```
              { (format-length)                    }
              {          A                         }
variable-name { (   {    U   }   )   DYNAMIC       } [ { CONSTANT } init-definition ] [emhdpm]
              {          B                         }   { INIT     }
              { HANDLE OF OBJECT                   }
```

`array-definition`

```
              { (format-length/array-definition)                                   }
              {              A                                                     }
variable-name {      (    {  U  }      /array-definition) DYNAMIC                  } [ { CONSTANT } array-init-definition ] [
              {              B                                                     }   { INIT     }
              { (array-definition) HANDLE OF OBJECT                                }
```

Syntax Description

Syntax Element	Description
`variable-name`	**Variable Name:** The name to be assigned to the variable. Rules for Natural variable names apply. With `DEFINE DATA INDEPENDENT`, the variable name must begin with a plus character (+). For information on naming conventions for user-defined variables, see *Naming Conventions for User-Defined Variables* in *Using Natural*.
`format-length`	**Format/Length Definition:** For information on format/length definition of user-defined variables, see *Format and Length of User-Defined Variables* in the *Programming Guide*.
`HANDLE OF OBJECT`	**Handle of Object:** Used in conjunction with NaturalX. A handle identifies a dialog element in code and is stored in handle variables.

Syntax Element	Description
	For further information, see *NaturalX* in the *Programming Guide*.
A, U or B	**Data Type:** Alphanumeric (A), Unicode (U) or binary (B) for dynamic variables.
array-definition	**Array Dimension Definition:** With an *array-definition* you define the lower and upper bounds of dimensions in an array-definition. For further information, see ***Array Dimension Definition***.
DYNAMIC	**DYNAMIC Option:** A field may be defined as DYNAMIC. For more information on processing dynamic variables, see *Introduction to Dynamic Variables and Fields*.
CONSTANT	**CONSTANT Option:** The variable/array is to be treated as a named constant. The constant value(s) assigned will be used each time the variable/array is referenced. The value(s) assigned cannot be modified during program execution. See also *Field Definitions*, *User-Defined Constants*, *Defining Named Constants* in the *Programming Guide*. **Note:** 1. For reasons of internal handling, it is not allowed to mix variable definitions and constant definitions within one group definition; that is, a group may contain either variables only or constants only. 2. The CONSTANT clause must not be used with DEFINE DATA INDEPENDENT and DEFINE DATA CONTEXT. The CONSTANT option cannot be used with X-arrays. 3. The CONSTANT option must not be used with DEFINE DATA INDEPENDENT and DEFINE DATA CONTEXT.
INIT	**INIT Option:** The variable/array is to be assigned an initial value. This value will also be used when this variable/array is referenced in a RESET INITIAL statement. If no INIT specification is supplied, a field will be initialized with a default initial value depending on its format (see table ***Default Initial Values*** below). For further information, see *Field Definitions*, *Initial Values* in the *Programming Guide*.

Syntax Element	Description
	With DEFINE DATA INDEPENDENT and DEFINE DATA CONTEXT, the INIT clause is evaluated in each executed Natural object that contains this clause (not only in the Natural object that allocates the variable). This is different to the way the INIT works for global variables. The INIT option cannot be used with X-arrays.
init-definition	**Initial-Value Definition:** With the *init-definition* option, you define the initial/constant values for a variable. See ***Initial-Value Definition***.
array-init-definition	**Initial/Constant Values for an Array:** The array is to be assigned an initial value. This value will also be used when this array is referenced in a RESET INITIAL statement. With an *array-init-definition*, you define the initial/constant values for an array. For further information, see ***Initial/Constant Values for an Array***.
emhdpm	**EM, HD, PM Parameters for Field/Variable:** With this option, additional parameters to be in effect for a field/variable may be defined. For further information, see ***EM, HD, PM Parameters for Field/Variable***.

Default Initial Values

The following table shows the default initial values that are provided with the various formats:

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

Fields declared as DYNAMIC do not have any initial value because their field length is zero by default.

46 View Definition

```
view-name VIEW [OF] ddm-name
[ level { ddm-field [ { ([format-length][/array-definition])
                        ( { A U B } [/array-definition] ) DYNAMIC } [emhdpm] ]
          redefinition } ] ...
```

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

The `view-definition` option is used to define a data view as derived from a data definition module (DDM).

Note: In a parameter data area, `view-definition` is not permitted.

For further information, see *Accessing Data in an Adabas Database* in the *Programming Guide* and particularly the following topics:

- *Data Definition Modules - DDMs*
- *Database Arrays*
- *Defining a Database View*

Syntax Description

Syntax Element	Description
`view-name`	**View Name:** The name to be assigned to the view. Rules for Natural variable names apply; see *Naming Conventions for User-Defined Variables* in *Using Natural*.
`VIEW [OF]` `ddm-name`	**DDM Name:** The name of the data definition module (DDM) from which the view is to be taken.
`level`	**Level Number:** Level number is a 1- or 2-digit number in the range from 01 to 99 (the leading zero is optional) used in conjunction with field grouping. Fields assigned a level number of 02 or greater are considered to be a part of the immediately preceding group which has been assigned a lower level number. The definition of a group enables reference to a series of fields (may also be only one field) by using the group name. With certain statements (`CALL`, `CALLNAT`, `RESET`,

Syntax Element	Description
	WRITE, etc.), you may specify the group name as a shortcut to reference the fields contained in the group. A group may consist of other groups. When assigning the level numbers for a group, no level numbers may be skipped.
ddm-field	**DDM Field Name:** The name of a field to be taken from the DDM. When you define a view for a HISTOGRAM statement, the view must contain only the descriptor for which HISTOGRAM is to be executed.
redefinition	**Redefinition:** A *redefinition* may be used to redefine a group, a view, a DDM field or a single field/variable (that is a scalar or an array). For further information, see ***Redefinition***.
format-length	**Format/Length Definition:** Format and length of the field. If omitted, these are taken from the DDM. In structured mode, the definition of format and length (if supplied) must be the same as those in the DDM. In reporting mode, the definition of format and length (if supplied) must be type-compatible with those in the DDM.
A, U or B	**Data Type:** Alphanumeric (A), Unicode (U) or binary (B) for dynamic variables. **Note:** 1. For Adabas on mainframe computers, format U is available for LA fields (length <= 16381 bytes), but not for LB fields (length: <= 1 GB). 2. Format B is not available with Adabas.
array-definition	**Array Definition:** Depending on the programming mode used, arrays (periodic-group fields, multiple-value fields) may have to contain information about their occurrences. For further information, see ***Array Definition in a View*** below.
emhdpm	**EM, HD, PM Parameters for Field/Variable:** With this option, additional parameters to be in effect for a field/variable may be defined. See ***EM, HD, PM Parameters for Field/Variable***.
DYNAMIC	**DYNAMIC Option:**

Syntax Element	Description
	Defines a view field as `DYNAMIC`. For further information on processing dynamic variables, see *Introduction to Dynamic Variables and Fields* in the *Programming Guide*.

Array Definition in a View

Depending on the programming mode used, arrays (periodic-group fields, multiple-value fields) may have to contain information about their occurrences.

Structured Mode

If a field is used in a view that represents an array, the following applies:

- An index value must be specified for MU/PE fields
- When no format/length specification is supplied, the values are taken from the DDM.
- When a format/length specification is supplied, it must be the same as in the DDM.

Database-Specific Considerations in Structured Mode:

Adabas:	If MU/PE fields (defined in a DDM) are to be used inside a view, these fields must include an array index specification. For an MU field or ordinary PE field, you specify a one-dimensional index range, e.g. (1:10). For an MU field inside a PE group, you specify a two-dimensional index range, e.g. (1:10,1:5).		
Tamino:	**DDM definition**	**allowed**	**not allowed**

	`A(*:X2)`	`A(*:Y2) Y2=<X2`
		`A(Y1:Y2) Y2>Y1`
		`Y2=<X2 A(Z:Z+Y) Y>=0`
			`A(*:*)`
			`A(Y1:*)`

	`A(X1:*)`	`A(Y1:*) Y1>=X1`
		`A(Y1:Y2) Y2>=X1, Y1>=X1`
		`A(Z:Z+Y) Y>=0`
			`A(*:*)`
			`A(*:Y2)`

	`A(X1:X2)`	`A(Y1:Y2) Y2<Y1`
		`A(Z:Z+Y) 0=<Y>=X2-X1+1`
			`A(*:*)`
			`A(Y1:*)`
			`A(*:Y2)`

Examples of Structured Mode:

```
DEFINE DATA LOCAL
1 EMP1 VIEW OF EMPLOYEES
  2 NAME(A20)
  2 ADDRESS-LINE(A20 / 1:2)

1 EMP2 VIEW OF EMPLOYEES
  2 NAME
  2 ADDRESS-LINE(1:2)

1 EMP3 VIEW OF EMPLOYEES
  2 NAME
  2 ADDRESS-LINE(2)

1 #K (I4)
1 EMP4 VIEW OF EMPLOYEES
  2 NAME
  2 ADDRESS-LINE(#K:#K+1)
END-DEFINE
END
```

Reporting Mode

In this mode, the same rules are valid as for structured mode, however, there are two exceptions:

- An index value needs not be supplied. In this case, the index range for the missing dimensions is set to (1:1).
- The format/length specification may differ from the specification in the DDM. Then the definition of format and length must be type-compatible with those in the DDM.

Examples:

```
DEFINE DATA LOCAL
1 EMP1 VIEW OF EMPLOYEES
  2 NAME(A30)
  2 ADDRESS-LINE(A35 / 5:10)

1 EMP2 VIEW OF EMPLOYEES
  2 NAME
  2 ADDRESS-LINE(A40)        /* ADDRESS LINE (1:1) IS ASSUMED

1 EMP3 VIEW OF EMPLOYEES
  2 NAME
  2 ADDRESS-LINE             /* ADDRESS LINE (1:1) IS ASSUMED

1 #K (I4)
1 EMP4 VIEW OF EMPLOYEES
  2 NAME
  2 ADDRESS-LINE(#K:#K+1)
```

```
END-DEFINE
END
```

47 Redefinition

```
                            ⎧ rgroup [(array-definition)]                       ⎫ ⎫
REDEFINE field-name ⎨ level ⎨ rfield (format-length [/array-definition])  ⎬ ⎬
                            ⎩ FILLER nX                                         ⎭ ⎭ ...
```

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

The `redefinition` option is used to redefine a group, a view, a DDM field or a single field/variable (that is a scalar or an array).

See also *Redefining Fields* in the *Programming Guide*.

Restrictions

- A `redefinition` of a view or a DDM field is not applicable to a `parameter-data-definition`.
- Handles, X-arrays and dynamic variables cannot be redefined and cannot be contained in a redefinition clause.
- A group that contains a handle, X-array or a dynamic variable can only be redefined up to - but not including or beyond - the element in question.

Syntax Description

Syntax Element	Description
field-name	**Name of Field to be Redefined:** The name of the group, view, DDM field or single field that is being redefined.
level	**Level Number of Field being Redefined:** Level number is a 1- or 2-digit number in the range from 01 to 99 (the leading zero is optional) used in conjunction with field grouping. Fields assigned a level number of 02 or greater are considered to be a part of the immediately preceding group, which has been assigned a lower level number.
rgroup	**Name of Resulting Group:** The name of the group resulting from the redefinition. **Note:** In a `redefinition` within a `view-definition`, the name of `rgroup` must be different from any field name in the underlying DDM.
rfield	**Name of Resulting Field:** The name of the field resulting from the redefinition.

Syntax Element	Description
	Note: In a *redefinition* within a *view-definition*, the name of *rfield* must be different from any field name in the underlying DDM.
format-length	**Format/Length of Resulting Field:** The format and length of the resulting field (*rfield*).
array-definition	**Array Dimension Definition:** With an *array-definition*, you define the lower and upper bounds of dimensions in an array-definition. For further information, see ***Array Dimension Definition***.
FILLER *n*X	**Filler Byte Definition:** With this notation, you define *n* filler bytes - that is, segments which are not to be used - in the field that is being redefined. The definition of trailing filler bytes is optional.

Examples of REDEFINE Usage

Example 1:

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

Example 2:

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

Example 3:

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

48 Array Dimension Definition

{[*bound*:] *bound*},... 3

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

The `array-dimension-definition` option is used to define the lower and upper bound of a dimension in an array definition.

You can define up to 3 dimensions for an array.

Syntax Description

Syntax Element	Description
bound	**Lower/Upper Bound:** A bound can be one of the following: ■ a numeric integer constant; ■ a previously defined named constant; ■ (for database arrays) a previously defined user-defined variable; or ■ an asterisk (*) defines an extensible bound, otherwise known as an X-array (eXtensible array). If only one bound is specified, the value represents the upper bound and the lower bound is assumed to be 1.

X-Arrays

If at least one bound in at least one dimension of an array is specified as extensible, that array is then called an X-array (eXtensible array). Only one bound (either upper or lower) may be extensible in any one dimension, but not both. Multi-dimensional arrays may have a mixture of constant and extensible bounds, for example: `#a(1:100, 1:*)`.

Example:

```
DEFINE DATA LOCAL
1 #ARRAY1(I4/1:10)
1 #ARRAY2(I4/10)
1 #X-ARRAY3(I4/1:*)
1 #X-ARRAY4(I4/*,1:5)
1 #X-ARRAY5(I4/*:10)
1 #X-ARRAY6(I4/1:10,100:*,*:1000)
END-DEFINE
```

In the following table you can see the bounds of the arrays in the above program more clearly.

	Dimension 1		Dimension 2		Dimension 3	
	Lower bound	Upper bound	Lower bound	Upper bound	Lower bound	Upper bound
#ARRAY1	1	10	-	-	-	-
#ARRAY2	1	10	-	-	-	-
#X-ARRAY3	1	eXtensible	-	-	-	-
#X-ARRAY4	1	eXtensible	1	5	-	-
#X-ARRAY5	eXtensible	10	-	-	-	-
#X-ARRAY6	1	10	100	eXtensible	eXtensible	1000

Examples of array definitions:

```
#ARRAY2(I4/10)                /* a one-dimensional array with 10 occurrences (1:10)
#X-ARRAY4(I4/*,1:5)             /* a two-dimensional array
#X-ARRAY6(I4/1:10,100:*,*:1000)  /* a three-dimensional array
```

Variable Arrays in a Parameter Data Area

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

Example 1: `#ARR01 (A5/1:V)`

Example 2: `#ARR02 (I2/1:V,1:V)`

A parameter array which contains a variable index notation `1:V` can only be redefined in the length of

- its elementary field length, if the `1:V` index is right-most; for example:

  `#ARR(A6/1:V)` can be redefined up to a length of 6 bytes
  `#ARR(A6/1:2,1:V)` can be redefined up to a length of 6 bytes
  `#ARR(A6/1:2,1:3,1:V)` can be redefined up to a length of 6 bytes

- the product of the right-most fixed occurrences and the elementary field length; for example:

  `#ARR(A6/1:V,1:2)` can be redefined up to a length of 2*6 = 12 bytes
  `#ARR(A6/1:V,1:3,1:2)` can be redefined up to a length of 3*2*6 = 36 bytes
  `#ARR(A6/1:2,1:V,1:3)` can be redefined up to a length of 3*6 = 18 bytes

A variable index notation `1:V` cannot be used within a redefinition.

Example:

```
DEFINE DATA PARAMETER
 1 #ARR(A6/1:V)
 1 REDEFINE #ARR
   2 #R-ARR(A1/1:V)   /* (1:V) is not allowed in a REDEFINE block
END-DEFINE
```

As the number of occurrences is not known at compilation time, it must not be referenced with the index notation (*) in the statements `INPUT`, `WRITE`, `READ WORK FILE`, `WRITE WORK FILE`. Index notation (*) may be applied either to all dimensions or to none.

Valid examples:

```
#ARR01 (*)
#ARR02 (*,*)
#ARR01 (1)
#ARR02 (5,#FIELDX)
#ARR02 (1,1:3)
```

Invalid example:

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

To avoid runtime errors, the maximum number of occurrences of such an array should be passed to the subprogram/subroutine/function via another parameter. Alternatively, you may use the system variable `*OCCURRENCE`.

Notes:

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

49 Initial-Value Definition

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

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

The *init-definition* option is used to define the initial/constant values for a variable.

Note: If, in the *variable-definition* option, the keyword `INIT` was used for the initialization, the value may be modified by any statement that affects the content of a variable. If the keyword `CONST` was used for the initialization, any attempt to change the value will be rejected by the compiler.

See also *Field Definitions*, *Initial Values* in the *Programming Guide*.

Restriction

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

Syntax Description

Syntax Element	Description
<constant>	**Constant Value Option:** The constant value with which the variable is to be initialized; or the constant value to be assigned to the field. For further information, see *User-Defined Constants* in the *Programming Guide*.

<system-variable>	**System Variable Option:**
	
	The initial value for a variable may also be the value of a Natural system variable, for example:
	
	`DEFINE DATA LOCAL`
	`1 #MYDATE (D) INIT <*DATX>`
	`END-DEFINE`
	
	Note: When the variable is referenced in a `RESET INITIAL` statement, the system variable is evaluated again; that is, it will be reset not to the value it contained when program execution started but to the value it contains when the `RESET INITIAL` statement is executed.

Syntax Element	Description

`FULL LENGTH <character-string>`
 	
`LENGTH n <character-string>`
	Character String Option for Alphanumeric/Unicode Variables:
	
	For a variable of the Natural data format A or U, a *character-string* (for example, `'ABC'`) can be used as an initial value which fills all or part of the variable field.
	
	A *character-string* is a constant of the Natural data format A or U as described in *Alphanumeric Constants* and *Unicode Constants* in the *Programming Guide*.
	
	FULL LENGTH Option:
	
	With the `FULL LENGTH` option, a particular *character-string* is repeatedly moved to the specified field until the field is completely filled. In the following example, the entire field is filled with asterisks:
	
	`DEFINE DATA LOCAL`
	`1 #FIELD (A25) INIT FULL LENGTH <'*'>`
	`END-DEFINE`
	
	LENGTH Option:
	
	With the `LENGTH n` option, a particular *character-string* is repeatedly moved to the specified field until the first *n* positions of the field are filled. *n* must be a numeric constant. In the following example, the first four positions of the field are filled with exclamation marks:
	
	`DEFINE DATA LOCAL`
	`1 #FIELD (A25) INIT LENGTH 4 <'!'>`
	`END-DEFINE`

50 Initial/Constant Values for an Array

For selected occurrences:

```
{ [ ( { index[:index] } ,   ) ] { { FULL LENGTH } <character-string, ... >    } } ...
{     {     V         } ...3    { { LENGTH n    }                             } }
{                               { < [ constant        ] , ... >               } }
{                               {   [ system-variable ]                       } }
```

For all occurrences:

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

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

The *array-init-definition* option is used to define the initial/constant values for an array.

Note: If, in the *variable-definition* option, the keyword INIT was used for the initialization, the value may be modified by any statement that affects the content of a variable. If the keyword CONST was used for the initialization, any attempt to change the value will be rejected by the compiler.

See also *Field Definitions* in the *Programming Guide*, particularly the following sections:

- *Initial Values*
- *User-Defined Constants*

Restriction

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

Syntax Description

Syntax Element	Description
`ALL`	**ALL Option:** All occurrences of the array are initialized with the same value. The `ALL` option cannot be combined with any other initialization definitions.

`index`	**Index Option:**
	
	The array occurrences specified by *`index`* are initialized.
	
	If a single index or an index range is used, you can only specify a unique value (*`constant`* or *`system-variable`*) which is assigned to all occurrences.
	
	Examples:
	
	`DEFINE DATA LOCAL`
	`1 #FLD1 (A4/1:4)     INIT (1:3) <'A'>  /* A fills occurrences (1:3)`
	`1 #FLD2 (A4/1:4)    INIT (*)   <'B'>  /* B fills all occurrences`
	`1 #FLD3 (A4/1:2,1:4) INIT (2,3) <'C'>  /* C fills occurrence (2,3)`
	`END-DEFINE`

`V`	**Index Notation V:**
	
	The special index notation `V` is used to fill a consecutive sequence of array occurrences with individual values (*`constant`* or *`system-variable`*).
	
	You can specify the `V` notation for one dimension of an array only. The number of values provided must not exceed the number of occurrences of the specified dimension.
	
	You can omit the `V` notation for a one-dimensional array because the `V` index is then used by default.
	
	Example showing which values fill which occurrences when `V` is used:
	
	`DEFINE DATA LOCAL`
	`1 #FLD4 (A4/1:3)     INIT (V)   <'A','B'>     /* A fills (1) B fills (2)`
	`1 #FLD5 (A4/1:2,1:3) INIT (1,V) <'C','D'>     /* C, D fill (1,1:2)`
	`                          (2,V) <'F','G','H'> /* F, G, H fill (2,1:3)`
	`END-DEFINE`

Syntax Element	Description

constant	**Constant Value Option:**
	
	The constant (value) with which the array is to be initialized.
	
	Occurrences for which no values are specified, are initialized with a **default value**.
	
	In a list of consecutive occurrences, you can skip single occurrences by specifying commas (`,`) only. However, you must end the list with a particular value for the last occurrence.
	
	For further information, see *User-Defined Constants* in the *Programming Guide*.
	
	Note: Multiple constant values/system variables must be separated either by the input delimiter character (as specified with the session parameter `ID`) or by a comma. If numbers are provided in the value list and a comma is defined as the decimal character (with the session parameter `DC`), either separate the comma from the value with an extra blank character or use the input delimiter character.
	
	Example with `ID=;` and `DC=,` delimiter settings:
	
	`DEFINE DATA LOCAL`
	`1 #FLD1 (A4/1:3)   INIT <'A',,'C'>`
	`1 #NUM1 (N4,2/1:3) INIT <1 , 2 , 3>`
	`1 #NUM2 (N4,2/1:3) INIT <1;2;3>`
	`END-DEFINE`

system-variable	**System Variable Option:** The initial value for an array can also be the value of a Natural system variable. See also the *Note* for *constant*.
FULL LENGTH <*character-string*> LENGTH *n* <*character-string*>	**Character String Option for Alphanumeric/Unicode Variables:** For a variable of the Natural data format A or U, a *character-string* (for example, `'ABC'`) can be used as an initial value which fills all or part of the variable field. A *character-string* is a constant of the Natural data format A or U as described in *Alphanumeric Constants* and *Unicode Constants* in the *Programming Guide*. **FULL LENGTH Option:** With the `FULL LENGTH` option, a particular *character-string* is repeatedly moved to the specified array occurrence until the occurrence is completely filled. **LENGTH Option:** With the `LENGTH` *n* option, a particular *character-string* is repeatedly moved to the specified array occurrence until the first *n* positions of the occurrence are filled.

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td></td><td>

Example showing which values fill which occurrences:

```
DEFINE DATA LOCAL
1 #FLD1 (A6/1:3) INIT ALL FULL LENGTH <'X'>    /* XXXXXX in all occ.
1 #FLD2 (A6/1:3) INIT ALL   LENGTH 5  <'NO'>   /* NONON in all occ.
1 #FLD3 (A6/1:3) INIT (1:2) LENGTH 4  <'AB'>   /* ABAB in occ (1:2)
1 #FLD4 (A6/1:3) INIT (V) FULL LENGTH <'X','Y'>/* XXXXXX in occ. (1),
                                               /* YYYYYY in occ. (2)
END-DEFINE
```

Within one *array-init-definition*, only `FULL LENGTH` or `LENGTH n` can be specified; both notations must not be mixed.

</td></tr>
</table>

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

51 EM, HD, PM Parameters for Field/Variable

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

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

The *emhdpm* option is used to define additional parameters to be in effect for a field/variable.

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

Syntax Description

Syntax Element	Description
EM=*value*	**Edit Mask:** The EM parameter may be used to define an edit mask used when the field is displayed with an I/O statement. For further information, see the session parameter EM in the *Parameter Reference*.
EMU=*value*	**Unicode Edit Mask:** The EMU parameter may be used to define a Unicode edit mask used when the field is displayed with an I/O statement. For further information, see the session parameter EMU in the *Parameter Reference*.
HD='*text*'	**Header Definition:** The HD parameter may be used to define the header to be used as the default header for the field. For further information, see the session parameter HD in the *Parameter Reference*.
PM=*value*	**Print Mode:** The PM parameter may be used to set the print mode, which indicates how fields are to be output. For further information, see the session parameter PM in the *Parameter Reference*.

52 Examples of DEFINE DATA Statement Usage

The following topics are covered:

Example 1 - DEFINE DATA LOCAL (Local Data Definition)

```
** Example 'DDAEX1': DEFINE DATA
************************************************************************
DEFINE DATA LOCAL
1 #VAR1     (A15)
1 #VAR2
  2 #VAR2A  (N4.1) INIT <1111>
  2 #VAR2B  (N6.2) INIT <222222>
1 REDEFINE #VAR2
  2 #VAR2C  (A2)
  2 #VAR2D  (A2)
  2 #VAR2E  (A6)
END-DEFINE
*
WRITE NOTITLE '=' #VAR2A / '=' #VAR2B /
              '=' #VAR2C / '=' #VAR2D / '=' #VAR2E
*
END
```

Output of Program DDAEX1:

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

Example 2 - DEFINE DATA LOCAL (Array Definition/Initialization)

```
** EXAMPLE 'DDAEX2': DEFINE DATA (array definition/initialization)
************************************************************************
DEFINE DATA LOCAL
**===============================================
1 #A01 (A5/1:4) INIT       <'A','B',,'D'>
1 #A02 (A5/1:4) INIT  (V)   <'A','B'>
                      (4)   <'D'>
1 #A03 (A5/1:4) INIT  (*)   <'A'>
1 #A04 (A5/1:4) INIT  (2)   <'B'>
                      (3)   <'C'>
1 #A05 (A5/1:4) INIT  (2:3) <'X'>
                      (4)   <'D'>
1 #A06 (A5/1:4) INIT  (*)   <'X'>
```

```
                      (3)   <'C'>
**-----------------------------------------------
1 #A10 (A5/1:4) INIT        FULL LENGTH <'X'>
1 #A11 (A5/1:4) INIT        FULL LENGTH <,'B',,'D'>
1 #A12 (A5/1:4) INIT  (V)   FULL LENGTH <'A','B'>
1 #A13 (A5/1:4) INIT  (2)   FULL LENGTH <'B'>
1 #A14 (A5/1:4) INIT  (*)   FULL LENGTH <'X'>
**-----------------------------------------------
1 #A20 (A5/1:4) INIT        LENGTH 2    <'A'>
1 #A21 (A5/1:4) INIT  (2)   LENGTH 2    <'B'>
                      (3)   LENGTH 3    <'C'>
1 #A22 (A5/1:4) INIT  (3:4) LENGTH 2    <'X'>
1 #A23 (A5/1:4) INIT  (V)   LENGTH 2    <,'B',,'D'>
**-----------------------------------------------
1 #A30 (A5/1:4) INIT  ALL             <'Z'>
1 #A31 (A5/1:4) INIT  ALL FULL LENGTH <'Z'>
1 #A32 (A5/1:4) INIT  ALL LENGTH 2    <'Z'>
**===============================================
1 #B01 (A5/1:2,1:4) INIT (2,V)   <'A','B',,'D'>
1 #B02 (A5/1:2,1:4) INIT (1,*)   <'X'>
                         (1,2)   <'B'>
                         (2,3)   <'F'>
1 #B03 (A5/1:2,1:4) INIT (1,1:2) <'X'>
                         (1,4)   <'Y'>
1 #B04 (A5/1:2,1:4) INIT (V,1)   <'A1','A2'>
1 #B05 (A5/1:2,1:4) INIT (V,*)   <'X','Y'>
**-----------------------------------------------
1 #B10 (A5/1:2,1:4) INIT ALL   <'Z'>
1 #B11 (A5/1:2,1:4) INIT (1,*) FULL LENGTH <'Z'>
1 #B12 (A5/1:2,1:4) INIT (*,*) FULL LENGTH <'Z'>
1 #B13 (A5/1:2,1:4) INIT (1,*) LENGTH 2    <'Z'>
1 #B14 (A5/1:2,1:4) INIT (1,V) FULL LENGTH <'A',,'C'>
                         (2,V) FULL LENGTH <'E','F'>
1 #B15 (A5/1:2,1:4) INIT (1,*) FULL LENGTH <'X'>
                         (2,*) FULL LENGTH <'Y'>
                         (2,4) FULL LENGTH <'Z'>
*
END-DEFINE
**===============================================
WRITE 7X '(1)   (2)   (3)   (4)'
WRITE (AD=V) '=' #A01(*)
WRITE (AD=V) '=' #A02(*)
WRITE (AD=V) '=' #A03(*)
WRITE (AD=V) '=' #A04(*)
WRITE (AD=V) '=' #A05(*)
WRITE (AD=V) '=' #A06(*)
SKIP 1
WRITE (AD=V) '=' #A10(*)
WRITE (AD=V) '=' #A11(*)
WRITE (AD=V) '=' #A12(*)
WRITE (AD=V) '=' #A13(*)
WRITE (AD=V) '=' #A14(*)
```

```
SKIP 1
WRITE (AD=V) '=' #A20(*)
WRITE (AD=V) '=' #A21(*)
WRITE (AD=V) '=' #A22(*)
WRITE (AD=V) '=' #A23(*)
SKIP 1
WRITE (AD=V) '=' #A30(*)
WRITE (AD=V) '=' #A31(*)
WRITE (AD=V) '=' #A32(*)
SKIP 1
**==============================================
WRITE 6X '(1,1) (1,2) (1,3) (1,4)  (2,1) (2,2) (2,3) (2,4)'
WRITE (AD=V) '=' #B01(1,*) 2X #B01(2,*)
WRITE (AD=V) '=' #B02(1,*) 2X #B02(2,*)
WRITE (AD=V) '=' #B03(1,*) 2X #B03(2,*)
WRITE (AD=V) '=' #B04(1,*) 2X #B04(2,*)
WRITE (AD=V) '=' #B05(1,*) 2X #B05(2,*)
SKIP 1
WRITE (AD=V) '=' #B10(1,*) 2X #B10(2,*)
WRITE (AD=V) '=' #B11(1,*) 2X #B11(2,*)
WRITE (AD=V) '=' #B12(1,*) 2X #B12(2,*)
WRITE (AD=V) '=' #B13(1,*) 2X #B13(2,*)
WRITE (AD=V) '=' #B14(1,*) 2X #B14(2,*)
WRITE (AD=V) '=' #B15(1,*) 2X #B15(2,*)
**==============================================
END
```

Output of Program DDAEX2:

```
Page      1

          (1)   (2)   (3)   (4)
#A01:    A     B           D
#A02:    A     B           D
#A03:    A     A     A     A
#A04:          B     C
#A05:          X     X     D
#A06:    X     X     C     X

#A10:    XXXXX
#A11:          BBBBB       DDDDD
#A12:    AAAAA BBBBB
#A13:          BBBBB
#A14:    XXXXX XXXXX XXXXX XXXXX

#A20:    AA
#A21:          BB    CCC
#A22:                XX    XX
#A23:          BB          DD

#A30:    Z     Z     Z     Z
#A31:    ZZZZZ ZZZZZ ZZZZZ ZZZZZ
#A32:    ZZ    ZZ    ZZ    ZZ

         (1,1) (1,2) (1,3) (1,4) (2,1) (2,2) (2,3) (2,4)
#B01:                            A     B           D
#B02:    X     B     X     X                 F
#B03:    X     X           Y
#B04:    A1                      A2
#B05:    X     X     X     X     Y     Y     Y     Y

#B10:    Z     Z     Z     Z     Z     Z     Z     Z
#B11:    ZZZZZ ZZZZZ ZZZZZ ZZZZZ
#B12:    ZZZZZ ZZZZZ ZZZZZ ZZZZZ ZZZZZ ZZZZZ ZZZZZ ZZZZZ
#B13:    ZZ    ZZ    ZZ    ZZ
#B14:    AAAAA       CCCCC       EEEEE FFFFF
#B15:    XXXXX XXXXX XXXXX XXXXX YYYYY YYYYY YYYYY ZZZZZ
```

Example 3 - DEFINE DATA (View Definition, Array Redefinition)

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

Output of Program DDAEX3:

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

SMITH                    SMITH
5 HAWTHORN               13002 NEW ARDEN COUR
OAK BROOK                SILVER SPRING
150-9351                 639-8963
```

Example 4 - DEFINE DATA (Global, Parameter and Local Data Areas)

```
** Example 'DDAEX4': DEFINE DATA (global and local data area definition)
************************************************************************
DEFINE DATA
GLOBAL
  USING DDAEX4G
LOCAL
1 #FIELD1 (A10)
1 #FIELD2 (N5)
END-DEFINE
*
MOVE 'HELLO' TO #FIELD1
MOVE 123     TO #FIELD2
*
CALLNAT 'DDAEX4N' #FIELD1 #FIELD2
*
END
```

Global Data Area DDAEX4G Used by Program DDAEX4:

```
1 GLOBAL-FIELD                     A   10
```

Subprogram DDAEX4N Called by Program DDAEX4:

```
** Example 'DDAEX4N': DEFINE DATA PARAMETER (called by DDAEX4)
************************************************************************
DEFINE DATA
PARAMETER
1 #FIELDA (A10)
1 #FIELDB (N5)
END-DEFINE
*
WRITE '=' #FIELDA '=' #FIELDB
END
```

Output of Program DDAEX4:

```
Page      1                                                  05-01-12  08:55:53

#FIELDA: HELLO      #FIELDB:    123
```

Example 5 - DEFINE DATA (Initialization)

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

Output of Program DDAEX5:

```
2005-01-12
__________________________________________________
....+..../....+..../....+..../....+..../....+..../....+..../....+
```

Example 6 - DEFINE DATA (Variable Array)

```
** Example 'DDAEX6': DEFINE DATA (variable array with (1:V))
************************************************************************
DEFINE DATA LOCAL
1 #ARRAY   (A1/1:10)
1 #MAX-ARR (P3)
END-DEFINE
*
#ARRAY (1) := 'R'
#ARRAY (2) := 'E'
#ARRAY (3) := 'D'
#MAX-ARR   := 4
*
WRITE #ARRAY(*)
*
CALLNAT 'DDAEX6N' #ARRAY(1:4) #MAX-ARR
*
```

```
WRITE #ARRAY(*)
*
*
#MAX-ARR   := 5
*
CALLNAT 'DDAEX6N' #ARRAY(1:5) #MAX-ARR
*
WRITE #ARRAY(*)
*
END
```

Subprogram DDAEX6N Called by Program DDAEX6:

```
** Example 'DDAEX6N': DEFINE DATA (variable array with (1:V))
************************************************************************
DEFINE DATA
PARAMETER
1 #STRING (A1/1:V)
1 #MAX    (P3)
END-DEFINE
*
IF #MAX = 4
  MOVE 'B' TO #STRING (1)
  MOVE 'L' TO #STRING (2)
  MOVE 'U' TO #STRING (3)
  MOVE 'E' TO #STRING (4)
END-IF
*
IF #MAX = 5
  MOVE 'W' TO #STRING (1)
  MOVE 'H' TO #STRING (2)
  MOVE 'I' TO #STRING (3)
  MOVE 'T' TO #STRING (4)
  MOVE 'E' TO #STRING (5)
END-IF
END
```

Output of Program DDAEX4:

```
Page      1                                                  05-01-12  09:06:43

R E D
B L U E
W H I T E
```

VII

53 DEFINE FUNCTION

```
DEFINE FUNCTION function-name
  [return-data-definition]
  [function-data-definition]
  statement...
END-FUNCTION
```

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

Related Statement: `DEFINE PROTOTYPE`

Function

The `DEFINE FUNCTION` statement is used to define a function which is stored as a Natural object of the type function. A function object may contain only one `DEFINE FUNCTION` statement.

The `DEFINE FUNCTION` statement defines the function name, the parameters, the local and application-independent variables, the function result and the statements forming the operation logic. These statements are executed when the function is called.

For further information, see the following sections in the *Programming Guide*:

- Natural object type Function
- *Function Call*

Syntax Description

Syntax Element	Description
function-name	**Function Name:** *function-name* is the name of the function to be called. It must comply with the naming conventions for user-defined variables described in the *Using Natural* documentation. *function-name* is not necessarily the same as the name of the stored object that contains the function definition. You may not use the same function name twice in one library.
return-data-definition	**Return Data Definition Clause:** For details on this clause, see ***Return Data Definition***.
function-data-definition	**Function Data Definition Clause:**

Syntax Element	Description
	For details on this clause, see *Function Data Definition*.
statement...	**Statement(s) to be Executed:** Defines the operation section which is executed when the function is called. It forms the function logic.
END-FUNCTION	**End of DEFINE FUNCTION Statement:** The Natural reserved word END-FUNCTION must be used to terminate the DEFINE FUNCTION statement.

Return Data Definition

```
                       (format-length[/array-definition])
 RETURNS             { [(array-definition)] HANDLE OF { dialog-element-type }          }  [BY
 [variable-name]     {                                { OBJECT              }          }  VALUE]
                     {                                                                 }
                     {             { A }                                               }
                     { (           { U }             [/array-definition])  DYNAMIC     }
                     {             { B }                                               }
```

The *return-data-definition* clause defines the format/length and, if applicable, the array structure of the result value returned by the function.

Syntax Element Description:

Syntax Element	Description
variable-name	**Return Value Name:** Optionally, you may specify a name which is used to access the return field within the function coding. If no such name is specified, the function name is used instead.
format-length	**Format/Length Definition:** The format and length of the result field. For information on format/length definition of user-defined variables, see *Format and Length of User-Defined Variables* in the *Programming Guide*.
array-definition	**Array Dimension Definition:** With *array-definition*, you define the lower and upper bounds of a dimension in an array-definition, if the function result is an array field. For further information, see DEFINE DATA statement, *Array Dimension Definition*.
HANDLE OF *dialog-element-type*	**Dialog Element Type:**

Syntax Element	Description
	The type of dialog element. Its possible values are the values of the `TYPE` attribute.
`HANDLE OF OBJECT`	**Handle of Object:** Used in conjunction with NaturalX. For further information, see *NaturalX* in the *Programming Guide*.
`A`, `U` or `B`	**Data Type:** Alphanumeric (A), Unicode (U) or binary (B) for a dynamic result.
`DYNAMIC`	**Dynamic Variable:** The function result may be defined as `DYNAMIC`. For information on processing dynamic variables, see *Introduction to Dynamic Variables and Fields* in the *Programming Guide*.
`BY VALUE`	**BY VALUE Option:** If `BY VALUE` is specified, the format/length of the "sending" field (defined inside the *return-data-definition* clause) and the "receiving" field (which receives the result at the place where the function is called) must only be transfer compatible. The format/length of the "receiving" field is either ■ defined via an explicit `(IR=)` clause in the function call; or ■ defined with a `DEFINE PROTOTYPE` statement; or ■ taken over from the `RETURNS` field of the function object, which must already exist. For data transfer compatibility the rules outlined in *Rules for Arithmetic Assignment* and *Data Transfer* in the *Programming Guide* apply. If `BY VALUE` is not specified, the format and length of the "receiving" field must exactly match the characteristics of the "sending" field.

Function Data Definition

```
DEFINE DATA
[ PARAMETER { USING parameter-data-area           } ] ...
            { parameter-data-definition ...       }
[ LOCAL { USING { local-data-area     }  } ] ...
        {       { parameter-data-area }  }
        { local-data-definition ...      }
[INDEPENDENT aiv-data-definition ...]
```

```
END-DEFINE
```

The *function-data-definition* clause defines the parameters which are to be provided when the function is called, and the data fields used by the function, such as local and application-independent variables. A global data area (GDA) cannot be referenced inside the function definition.

Syntax Element Description:

Syntax Element	Description
PARAMETER USING *parameter-data-area*	**PDA Name:** Specify the name of the parameter data area (PDA) that contains data elements which are used as parameters in a function call. See also ***Defining Parameter Data*** in the DEFINE DATA statement description.
PARAMETER *parameter-data-definition*	**Parameter Data Definition:** Instead of defining a parameter data area, parameter data can also be defined directly within a function call. See also ***Parameter Data Definition*** in the DEFINE DATA statement description.
LOCAL USING *local-data-area*	**LDA Name:** Specify the name of the local data area (LDA) to be referenced. See also ***Defining Local Data*** in the DEFINE DATA statement description.
LOCAL USING *parameter-data-area*	**PDA Name:** Specify the name of a parameter data area (PDA). **Note:** A data area referenced with DEFINE DATA LOCAL may also be a parameter data area (PDA). 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. See also ***Defining Local Data*** in the DEFINE DATA statement description.
LOCAL *local-data-definition*	**Local Data Definition:** For information on how to define elements or fields within the statement itself, that is, without using an LDA or PDA, see the section ***Local Data Definition*** in the DEFINE DATA statement description.
INDEPENDENT *aiv-data-definition*	**AIV Data Definition:** Can be used to define a single or multiple application-independent variables (AIVs). See ***Defining Application-Independent Variables*** in the DEFINE DATA statement description.

Syntax Element	Description
END-DEFINE	**End of Clause:** The Natural reserved word END-DEFINE must be used to end the *function-data-definition* clause.

Examples

- Example 1 - DEFINE FUNCTION
- Example 2 - DEFINE FUNCTION with Result Value Array

Example 1 - DEFINE FUNCTION

```
** Example 'DFUEX1': DEFINE FUNCTION
************************************************************************
DEFINE FUNCTION F#FIRST-CHAR
  RETURNS #RESULT (A1)
  DEFINE DATA PARAMETER
    1 #PARM (A10)
  END-DEFINE
  /*
  #RESULT := #PARM        /* First character as return value.
END-FUNCTION
*
END
```

The function F#FIRST-CHAR is used in the example program DPTEX2 in library SYSEXSYN. See ***Examples*** in the DEFINE PROTOTYPE statement description.

Example 2 - DEFINE FUNCTION with Result Value Array

```
** Example 'DFUEX2': DEFINE FUNCTION
************************************************************************
DEFINE FUNCTION F#FACTOR
  RETURNS (I2/1:3)
  DEFINE DATA PARAMETER
    1 #VALUE (I2)
  END-DEFINE
  /*
  F#FACTOR(1) := #VALUE * 1
  F#FACTOR(2) := #VALUE * 2
  F#FACTOR(3) := #VALUE * 3
  /*
END-FUNCTION
*
END
```

The function F#FACTOR is used in the example program DPTEX1 in library SYSEXSYN. See ***Examples*** in the DEFINE PROTOTYPE statement description.

54 DEFINE PRINTER

```
DEFINE PRINTER([logical-printer-name=]n)
[OUTPUT operand1]
[  PROFILE operand2  ]
[  DISP operand2     ]
[  COPIES operand3   ] ...
```

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

Related Statements: AT END OF PAGE | AT TOP OF PAGE | CLOSE PRINTER | DISPLAY | EJECT | FORMAT | NEWPAGE | PRINT | SKIP | SUSPEND IDENTICAL SUPPRESS | WRITE | WRITE TITLE | WRITE TRAILER

Belongs to Function Group: ***Creation of Output Reports***

Function

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

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

Syntax Description

Operand Definition Table:

Operand	Possible Structure			Possible Formats	Referencing Permitted	Dynamic Definition
operand1	C	S		A	yes	no
operand2	C	S		A	yes	no

Syntax Element Description:

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td>(n)</td><td>Printer Number (Report Number):

The report number n may be a value in the range of 0 - 31. This is the number also to be used in a DISPLAY / WRITE or CLOSE PRINTER statement.

Report number 0 indicates the output channel of the main report. Only output statements such as PRINT, WRITE or DISPLAY are affected. The INPUT statement is not affected.</td></tr>
<tr><td>logical-printer-name</td><td>Logical Printer Name:

Optionally you can assign a logical name logical-printer-name to printer n. This name can be used for the rep notation in a DISPLAY / WRITE statement.

Naming conventions for logical-printer-name are the same as for user-defined variables. Multiple logical names may be assigned to the same printer number. Unlike the value of the OUTPUT operand (see below), logical-printer-name is evaluated at compile time and therefore independent of the program control flow.</td></tr>
<tr><td>OUTPUT operand1</td><td>Printer Name:

If operand1 is a variable, its format/length must be A8 or one of the following. The name must be specified as LPTnn, where nn may be a number in the range of 1 - 31. See also Example 1.

Note: If the output data written to a report is to be sent to an Entire Connection terminal and then be written to an NCD file on a PC, one of the printer names LPTnn (where nn is a number in the range of 1 - 31) must be specified as operand1.

Note: The device assignments of logical printer LPTnn must be set in the Configuration Utility; see Device/Report Assignments. As Device Destination of the Physical Output Device, the value E (send data to an Entire Connection terminal) must be specified.

Additional reports can be assigned with the following names:
<table>
<tr><th>Report</th><th>Function</th></tr>
<tr><td>DUMMY</td><td>Output to be deleted.</td></tr>
<tr><td>INFOLINE</td><td>Output to the Natural infoline. For details on the infoline, see the Natural terminal command %X in the Terminal Commands documentation. See also Example 2.</td></tr>
<tr><td>SOURCE</td><td>Output to the Natural source area.</td></tr>
<tr><td>NOM</td><td>Output to Entire Output Management. Refer to the Entire Output Management documentation for details.</td></tr>
</table></td></tr>
<tr><td>PROFILE operand2</td><td>Name of Printer Control Characters Table:</td></tr>
</table>

Syntax Element	Description
	With the `PROFILE` clause, you specify as *operand2* the name of a printer control characters table. The maximum length allowed for *operand2* is 8. Such a table is defined in the global configuration file. See *Printer Profiles* in the *Configuration Utility* documentation for details on how to set printer profiles.
`DISP` *operand2*	**Disposition:** Maximum length of operand: 4 bytes. Possible values for *operand2*: `DEL` – The temporary spool file is deleted after its content has been printed. This is the default value. `KEEP` – The temporary spool file is *not* deleted after its content has been printed. `HOLD` – The temporary spool file is neither deleted nor printed.
`COPIES` *operand3*	**Number of Copies:** *operand3* must be an integer value.

Examples

- Example 1 - Printer Name Definition
- Example 2 - Print Output to Infoline

Example 1 - Printer Name Definition

```
/* PRINTER NAME DEFINED FOR WINDOWS
*
DEFINE PRINTER (REPORT1 = 1) OUTPUT 'LPT1'
WRITE (REPORT1) 'REPORT 1 PRINTED ON PRINTER LPT1'
END
```

Example 2 - Print Output to Infoline

```
** Example 'DPIEX1': DEFINE PRINTER
************************************************************************
*
SET CONTROL 'XI+'        /* SWITCH INFOLINE MODE ON
SET CONTROL 'XT'         /* INFOLINE TOP
*
DEFINE PRINTER (1) OUTPUT 'INFOLINE'
WRITE (1) 'EXECUTING' *PROGRAM 'BY' *INIT-USER
WRITE 'TEST OUTPUT'
```

```
EJECT                    /* FORCE PHYSICAL I/O
*
SET CONTROL 'X'          /* SWITCH BACK TO NORMAL
*
END
```

Output of Program DPIEX1:

```
EXECUTING DPIEX1   BY HTR
Page      1                                                05-01-13  14:54:33

TEST OUTPUT
```

55 DEFINE PROTOTYPE

```
DEFINE        [FOR]         prototype-name
PROTOTYPE     VARIABLE

  [ UNKNOWN                                 ]
  [ [return-data-definition]                ]
  [ [parameter-definition]                  ]
  [ same-as-clause                          ]
  [ USING FUNCTION [DEFINITION [OF]]        ]
  [ function-name                           ]

END-PROTOTYPE
```

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

Related Statement: `DEFINE FUNCTION`

Function

The `DEFINE PROTOTYPE` statement is used to specify the properties for calling a function including the following:

- the parameters to be passed in the function call,
- the result value to be returned by the function call, and
- whether the function is called with the function name defined in the `DEFINE FUNCTION` statement, or with an alphanumeric variable that contains the function name.

This information is used to resolve a function call within a Natural object at compile time.

A `DEFINE PROTOTYPE` statement is only needed for a function call if any of the following is true:

- The specified function name is an alphanumeric variable which contains the name of the function to be called at execution time.
- An `(IR=)` clause is not specified in the function call and a cataloged object of the called function is not available.
- The parameters provided in the function call are to be validated and the cataloged object of the called function is not available.

The `DEFINE PROTOTYPE` statement can be included in a copycode object if the function is to be called from multiple objects.

For further information, see the following sections in the *Programming Guide*:

- Natural object type Function

- *Function Call*

Syntax Description

Syntax Element	Description
`[VARIABLE]` *`prototype-name`*	**Prototype Name:** *`prototype-name`* is either of the following: - the name of the prototype whose parameter and result field definitions are to be used. This name typically matches the *`function-name`* in the `DEFINE FUNCTION` statement of the referenced function; - the name of an alphanumeric field specified as *`function-name`* in a function call if the keyword `VARIABLE` is specified. 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.
`UNKNOWN`	**UNKNOWN Option:** The keyword `UNKNOWN` specifies that the function interface is currently undefined. In this case, the cataloged object (if available) will not be used to extract the function result and the parameter description. When a function call is embedded in a Natural statement, this requires to give the result layout explicitly with an `(IR=)` clause. In addition, parameters provided in the function call are not checked.
`return-data-definition`	See ***Return Data Definition*** below.
`parameter-definition`	See ***Parameter Definition*** below.
`same-as-clause`	See ***SAME AS Clause*** below.
`USING FUNCTION [DEFINITION [OF]]` *`function-name`*	**USING FUNCTION Clause:** *`function-name`* is the name of an existing cataloged object of the type function. The parameters and the result field definitions of this function are used to resolve the function call.
`END-PROTOTYPE`	**End of DEFINE PROTOTYPE Statement:** The Natural reserved word `END-PROTOTYPE` must be used to terminate the `DEFINE PROTOTYPE` statement.

Return Data Definition

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

The `return-data-definition` clause defines the format/length and, if applicable, the array structure of the return value.

When no return data definition is specified, a function call can only be used within a statement if an explicit `(IR=)` clause is provided. If such a clause is missing, the function can only be called as a statement, but not in place of an operand within a statement.

Syntax Element Description:

Syntax Element	Description
variable-name	**Return Value Name:** The optional *variable-name* has no meaning. It is just there to have a syntax structure similar to the ***Return Data Definition*** clause of the `DEFINE FUNCTION` statement.
format-length	**Format/Length Definition:** The format and length of the result field. For information on format/length definition of user-defined variables, see *Format and Length of User-Defined Variables* in the *Programming Guide*.
array-definition	**Array Dimension Definition:** With *array-definition*, you define the lower and upper bounds of a dimension in an array-definition, if the function result is an array field. For further information, see ***Array Dimension Definition*** in the description of the `DEFINE DATA` statement.
HANDLE OF OBJECT	**Handle of Object:** Used in conjunction with NaturalX.
A, U or B	**Data Type:** Alphanumeric (A), Unicode (U) or binary (B) for a dynamic result.
DYNAMIC	**Dynamic Variable:** The function result may be defined as `DYNAMIC`. For information on processing dynamic variables, see *Introduction to Dynamic Variables and Fields* in the *Programming Guide*.

Parameter Definition

```
DEFINE DATA
  { PARAMETER UNKNOWN
    { PARAMETER [ USING parameter-data-area
                  parameter-data-definition ] } ...
                  ...                            }
END-DEFINE
```

The *parameter-definition* clause defines the parameters which are to be provided in a function call. This definition layout is checked against the parameters given in a function call. If this clause is omitted, this declares the function as free of parameters. In this case, every attempt to provide parameters in the function call is rejected.

The identifiers used to name the parameter fields have no meaning. They are just there to have a syntax structure similar to the DEFINE DATA PARAMETER syntax.

Syntax Element Description:

Syntax Element	Description
PARAMETER UNKNOWN	**UNKNOWN Option:** With this option, no parameter is specified and the parameter check in the function call is disabled. As a consequence, any number of parameters in the function call will be accepted.
USING *parameter-data-area*	**PDA Name:** The name of the *parameter-data-area* that contains data elements which are used as parameters in a function call. See also ***Defining Parameter Data*** in the DEFINE DATA statement description.
parameter-data-definition	**Parameter Data Definition:** Instead of defining a parameter data area, parameter data can also be defined directly within a function call. See also ***Parameter Data Definition*** in the DEFINE DATA statement description.
END-DEFINE	**End of Clause:** The Natural reserved word END-DEFINE must be used to end the *parameter-definition* clause.

SAME AS Clause

```
SAME AS [PROTOTYPE] prototype-name
```

With the `SAME AS` clause you can use the parameter and result field definitions of another prototype which has been defined before in the same Natural object.

Examples

Example 1 - DEFINE PROTOTYPE with a Defined Function Name

This is a prototype definition for a function named `F#FACTOR` where the *`prototype-name`* corresponds to the *`function-name`* specified in the referenced `DEFINE FUNCTION` statement. The result returned by the function is of format (I2/1:3), and a single parameter of format (I2) is required.

```
** Example 'DPTEX1': DEFINE PROTOTYPE and function call
************************************************************************
DEFINE DATA LOCAL
  1 #NUM (I2)
END-DEFINE
*
DEFINE PROTOTYPE F#FACTOR
  RETURNS (I2/1:3)
  DEFINE DATA PARAMETER
    1 #VALUE (I2)
  END-DEFINE
END-PROTOTYPE
*
#NUM := 3
*
WRITE 'Function call:' F#FACTOR(<#NUM>)(*)
*
END
```

The function `F#FACTOR` is defined in the example function `DFUEX2` in library `SYSEXSYN`. See ***Examples*** in the `DEFINE FUNCTION` statement description.

Output of Program `DPTEX1`**:**

```
Function call:       3       6       9
```

Example 2 - DEFINE PROTOTYPE with a Variable Function Name

Due to the keyword `VARIABLE`, this prototype specifies a function call where the referenced *`prototype-name`* is an alphanumeric variable which contains the function name at execution time.

```
** Example 'DPTEX2': DEFINE PROTOTYPE and function call
************************************************************************
DEFINE DATA LOCAL
  1 #NAME (A20)
  1 #TEXT (A10)
END-DEFINE
*
DEFINE PROTOTYPE VARIABLE #NAME
  RETURNS #RETURN (A1)
  DEFINE DATA PARAMETER
    1 #IN (A10)
  END-DEFINE
END-PROTOTYPE
*
#NAME := 'F#FIRST-CHAR'
#TEXT := 'ABCDEFGHIJ'
*
WRITE 'First character:' #NAME(<#TEXT>)
*
END
```

The function `F#FIRST-CHAR` is defined in the example function `DFUEX1` in library `SYSEXSYN`. See *Examples* in the `DEFINE FUNCTION` statement description.

Output of Program `DPTEX2`**:**

```
First character: A
```

56 DEFINE SUBROUTINE

```
DEFINE [SUBROUTINE] subroutine-name
            statement ...
        { END-SUBROUTINE (structured mode only) }
        { RETURN (reporting mode only)          }
```

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

Related Statements: CALL | CALL FILE | CALL LOOP | CALLNAT | ESCAPE | FETCH | PERFORM

Belongs to Function Group: ***Invoking Programs and Routines***

Function

The DEFINE SUBROUTINE statement is used to define a Natural subroutine. A subroutine is invoked with a PERFORM statement.

Inline/External Subroutines

A subroutine may be defined within the object which contains the PERFORM statement that invokes the subroutine (**inline subroutine**); or it may be defined external to the object that contains the PERFORM statement (**external subroutine**). An inline subroutine may be defined before or after the first PERFORM statement which references it.

Note: Although the structuring of a program function into multiple external subroutines is recommended for achieving a clear program structure, please note that a subroutine should always contain a larger function block because the invocation of the external subroutine represents an additional overhead as compared with inline code or subroutines.

Data Available in a Subroutine

Inline Subroutines

No explicit parameters can be passed from the invoking program via the PERFORM statement to an internal subroutine.

An inline subroutine has access to the currently established global data area as well as to the local data area used by the invoking program.

External Subroutines

An external subroutine has access to the currently established global data area. In addition, parameters can be passed directly with the PERFORM statement from the invoking object to the external subroutine; thus, you may reduce the size of the global data area.

An external subroutine has no access to the local data area defined in the calling program; however, an external subroutine may have its own local data area.

Restrictions

- Any processing loop initiated within a subroutine must be closed before `END-SUBROUTINE` is issued.
- An inline subroutine must not contain another `DEFINE SUBROUTINE` statement (see ***Example 1*** below).
- An external subroutine (that is, an object of type subroutine) must not contain more than one `DEFINE SUBROUTINE` statement block (see ***Example 2*** below). However, an external `DEFINE SUBROUTINE` block may contain further inline subroutines (see ***Example 1*** below).
- You may not use the name of an external subroutine twice in one library.

Example 1

The following construction is possible in an object of type subroutine, but not in any other object (where `SUBR01` would be considered an inline subroutine):

```
...
DEFINE SUBROUTINE SUBR01
   ...
   PERFORM SUBR02
   PERFORM SUBR03
   ...
   DEFINE SUBROUTINE SUBR02
   /* inline subroutine...
   END-SUBROUTINE
 ...
   DEFINE SUBROUTINE SUBR03
   /* inline subroutine...
   END-SUBROUTINE
END-SUBROUTINE
END
```

Example 2 (invalid):

The following construction is *not* allowed in an object of type subroutine:

```
...
DEFINE SUBROUTINE SUBR01
...
END-SUBROUTINE
DEFINE SUBROUTINE SUBR02
...
END-SUBROUTINE
END
```

Syntax Description

Syntax Element	Description
subroutine-name	**Name of Subroutine:** For a subroutine name (maximum 32 characters), the same naming conventions apply as for user-defined variables; see *Naming Conventions for User-Defined Variables* in the *Using Natural* documentation. The subroutine name is independent of the name of the module in which the subroutine is defined (it may but need not be the same).
statement	**Statement(s) to be Executed:** In place of *statement*, you must supply one or several suitable statements, depending on the situation. For an example of a statement, see ***Examples*** below.
END-SUBROUTINE RETURN	**End of DEFINE SUBROUTINE Statement:** In structured mode, the subroutine definition is terminated with END-SUBROUTINE. In reporting mode, RETURN may be used to terminate a subroutine.

Examples

- Example 1 - Define Subroutine

- Example 2 - Sample Structure for External Subroutine Using GDA Fields

Example 1 - Define Subroutine

```
** Example 'DSREX1S': DEFINE SUBROUTINE (structured mode)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 ADDRESS-LINE  (A20/2)
  2 PHONE
*
1 #ARRAY    (A75/1:4)
1 REDEFINE #ARRAY
  2 #ALINE  (A25/1:4,1:3)
1 #X        (N2) INIT <1>
1 #Y        (N2) INIT <1>
END-DEFINE
*
FORMAT PS=20
LIMIT 5
FIND EMPLOY-VIEW WITH NAME = 'SMITH'
  MOVE NAME            TO #ALINE (#X,#Y)
  MOVE ADDRESS-LINE(1) TO #ALINE (#X+1,#Y)
  MOVE ADDRESS-LINE(2) TO #ALINE (#X+2,#Y)
  MOVE PHONE           TO #ALINE (#X+3,#Y)
  IF  #Y = 3
    RESET INITIAL #Y
    PERFORM PRINT
  ELSE
    ADD 1 TO #Y
  END-IF
  AT END OF DATA
    PERFORM PRINT
  END-ENDDATA
END-FIND
*
DEFINE SUBROUTINE PRINT
  WRITE NOTITLE (AD=OI) #ARRAY(*)
  RESET #ARRAY(*)
  SKIP 1
END-SUBROUTINE
*
END
```

Output of Program DSREX1S:

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

SMITH                    SMITH
5 HAWTHORN               13002 NEW ARDEN COUR
OAK BROOK                SILVER SPRING
150-9351                 639-8963
```

Equivalent reporting-mode example: **DSREX1R**.

Example 2 - Sample Structure for External Subroutine Using GDA Fields

```
** Example 'DSREX2': DEFINE SUBROUTINE (using GDA fields)
************************************************************************
DEFINE DATA
GLOBAL
  USING DSREX2G
END-DEFINE
*
INPUT 'Enter value in GDA field' GDA-FIELD1
*
*  Call external subroutine in DSREX2S
*
PERFORM DSREX2-SUB
*
END
```

Global Data Area DSREX2G Used by Program DSREX2:

```
1 GDA-FIELD1                       A    2
```

Subroutine DSREX2S Called by Program DSREX2:

```
** Example 'DSREX2S': SUBROUTINE (external subroutine using global data)
************************************************************************
DEFINE DATA
GLOBAL
  USING DSREX2G
END-DEFINE
*
DEFINE SUBROUTINE DSREX2-SUB
*
  WRITE 'IN SUBROUTINE' *PROGRAM '=' GDA-FIELD1
*
END-SUBROUTINE
```

```
*
END
```

57 DEFINE WINDOW

```
DEFINE WINDOW window-name
[ SIZE { AUTO
         QUARTER
         operand1 * operand2 } ]
[ BASE { CURSOR
         { TOP    }  { LEFT  }
         { BOTTOM }  { RIGHT }
         operand3 / operand4 } ]
[REVERSED [(CD=background-color)]]
[TITLE operand5]
[ CONTROL { WINDOW
            SCREEN } ]
[ FRAMED { [ON] [(CD=frame-color)] [position-clause]
           OFF } ]
```

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

Related Statements: `INPUT` | `REINPUT` | `SET WINDOW`

Belongs to Function Group: ***Screen Generation for Interactive Processing***

Function

The `DEFINE WINDOW` statement is used to specify the size, position and attributes of a window.

A window is that segment of a logical page, built by a program, which is displayed on the terminal screen. There is always a window present, although you may not be aware of its existence: unless specified differently, the size of the window is identical to the physical size of your terminal screen.

A `DEFINE WINDOW` statement does not activate a window; this is done with a `SET WINDOW` statement or with the `WINDOW` clause of an `INPUT` statement.

Note: There is always only *one* Natural window, that is, the most recent window. Any previous windows may still be visible on the screen, but are no longer active and are ignored by Natural. You may enter input only in the most recent window. If there is not enough space to enter input, the window size must be adjusted first.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S						N	P	I									yes	no
operand2	C	S						N	P	I									yes	no
operand3	C	S						N	P	I									yes	no
operand4	C	S						N	P	I									yes	no
operand5	C	S				A	U												yes	no

Syntax Element Description:

Syntax Element	Description
window-name	The *window-name* identifies the window. The name may be up to 32 characters long. For a window name, the same naming conventions apply as for user-defined variables, see *Naming Conventions for User-Defined Variables* in the *Using Natural* documentation.
SIZE	With the SIZE clause, you specify the size of the window. **Note:** On mainframe computers, Natural requires additional columns for so-called attribute bytes to be able to display data on the screen (on other platforms, such attribute bytes are not needed). Consequently, on mainframe computers the screen area overlaid by a window is wider, and the size of the page segment visible inside a window is smaller than on other platforms. Example: Assume a window whose size is defined as SIZE 5 * 15 (that is, with a width of 15 columns): ■ On mainframe computers, the screen area overlaid by the window is 16 columns; the size of what is visible inside the window is 14 columns without frame, and 10 columns with frame respectively. ■ On other platforms, the screen area overlaid by the window is 15 columns; the size of what is visible inside the window is 15 columns without frame, and 13 columns with frame respectively.
SIZE AUTO	The size of the window is determined automatically by Natural at runtime. The size is determined by the data generated into the window as follows: ■ The number of window lines will be the number of INPUT lines generated (plus possibly the PF-key lines, message line, and infoline/statistics line). ■ The number of window columns is determined by the longest INPUT line: Natural scans, starting from the ends of the lines, for the rightmost significant byte in a

Syntax Element	Description
	line. This may cause an input-only or modifiable field (`AD=A` or `AD=M`) to be truncated; to avoid this, you either put a single-character text string after such a field or explicitly set the window size with the following: `SIZE operand1 * operand2` If you omit the `SIZE` clause, `SIZE AUTO` applies by default. **Note:** The title is not part of the window data. Therefore, if the window size has been determined as described above *and* the title is longer than the window, it will be truncated.
`SIZE QUARTER`	The size of the window will be one quarter of the physical screen.
`SIZE` *`operand1`* `*` *`operand2`*	The size of the window will be *n* lines by *n* columns. The number of lines is determined by *operand1*, the number of columns by *operand2*. Neither of the two operands must contain decimal digits. If the window is `FRAMED`, the specified size will be inclusive of the frame. The minimum possible window size is: ■ without frame: 2 lines by 10 columns, ■ with frame: 4 lines by 13 columns. The maximum possible window size is the size of the physical screen.
`BASE`	With the `BASE` clause, you determine the position of the window on the physical screen. If you omit the `BASE` clause, `BASE CURSOR` applies by default.
`BASE CURSOR`	Places the top left corner of the window at the current cursor position. The cursor position is the physical position of the cursor on the screen. If the size of the window makes it impossible to place the window at the cursor position, Natural automatically places the window as close as possible to the desired position.
`BASE TOP/BOTTOM LEFT/RIGHT`	Places the window at the top-left, bottom-left, top-right, or bottom-right corner respectively of the physical screen.
`BASE` *`operand3/operand4`*	This places the top left corner of the window at the specified line/column of the physical screen. The line number is determined by *operand3*, the column number by *operand4*. Neither of the two operands must contain decimal digits. If the size of the window makes it impossible to place the window at the specified position, you will get an error message.
`REVERSED`	`REVERSED` will cause the window to be displayed in reverse video (if the screen used supports this feature; if it does not, `REVERSED` will be ignored).
`REVERSED CD=` *`background-color`*	This will cause the window to be displayed in reverse video and the background of the window in the specified color (if the screen used supports these features; if it does not, the respective specification will be ignored).

Syntax Element	Description
	For information on valid color codes, see session parameter `CD` in the *Parameter Reference*.
`TITLE` *operand5*	With the `TITLE` clause, you may specify a heading for the window. The specified title (*operand5*) will be displayed centered in the top frame-line of the window. The title can be specified either as a text constant (in apostrophes) or as the content of a user-defined variable. If the title is longer than the window, it will be truncated. The title is only displayed if the window is `FRAMED`; if `FRAMED OFF` is specified for the window, the `TITLE` clause will be ignored. **Note:** If the title contains trailing blanks, these will be removed. If the first character of the title is a blank, one blank will automatically be appended to the title.
`CONTROL`	With the `CONTROL` clause, you determine whether the PF-key lines, the message line and the statistics line are displayed in the window or on the full physical screen.
`CONTROL WINDOW`	`CONTROL WINDOW` causes the lines to be displayed inside the window. If you omit the `CONTROL` clause, `CONTROL WINDOW` applies by default.
`CONTROL SCREEN`	`CONTROL SCREEN` causes the lines to be displayed on the full physical screen outside the window.
`FRAMED`	By default, that is, if you omit the `FRAMED` clause, the window is framed. The top and bottom frame lines are cursor-sensitive: where applicable, you can page forward, backward, left or right within the window by simply placing the cursor over the appropriate symbol (<, -, +, or >; see *position-clause* below) and then pressing ENTER. If no symbols are displayed, you can page backward and forward within the window by placing the cursor in the top frame line (for backward positioning) or bottom frame line (for forward positioning) and then pressing ENTER. **Note:** If the window size is smaller than 4 lines by 12 (or 13 on mainframe computers) columns, the frame will not be visible.
`FRAMED OFF`	If you specify `FRAMED OFF`, the framing and everything attached to the frame (window title and position information) will be switched off.
`FRAMED (CD=`*frame-color*`)`	This causes the frame of the window to be displayed in the specified color (if the screen used is a color screen; if it is not, the color specification will be ignored). For information on valid color codes, see session parameter `CD` (in the *Parameter Reference*). **Note:** In Natural for Windows, this specification is ignored.
position-clause	The `POSITION` clause is only evaluated on mainframe computers; on all other platforms it is ignored. For details, refer to ***Position Clause*** below.

POSITION Clause

The POSITION clause is only evaluated on mainframe computers; on all other platforms it is ignored.

```
          { SYMBOL   [ TOP    ]  [AUTO] [SHORT]   [ LEFT  ] }
          {          [ BOTTOM ]                   [ RIGHT ] }
POSITION  {                                                 }
          {          TEXT        [MORE] [ LEFT  ]           }
          {                             [ RIGHT ]           }
          {                      OFF                        }
```

The POSITION clause causes information on the position of the window on the logical page to be displayed in the frame of the window. This applies only if the logical page is larger than the window; if it is not, the POSITION clause will be ignored. The position information indicates in which directions the logical page extends above, below, to the left and to the right of the current window.

If the POSITION clause is omitted, POSITION SYMBOL TOP RIGHT applies by default.

Syntax Element Description:

Syntax Element	Description
POSITION SYMBOL	Causes the position information to be displayed in form of symbols: More: < - + >. The information is displayed in the top and/or bottom frame line.
TOP/BOTTOM	Determines whether the position information is displayed in the top or bottom frame line.
AUTO	Is only applicable if the logical page is fully visible in the window as far as its horizontal size is concerned, that is, if only a minus sign character (-) and/or a plus sign character (+) are to be displayed. In this case, AUTO automatically switches from the symbols to the words Top, Bottom and More respectively.
SHORT	Causes the word More: before the symbols < - + > to be suppressed.
LEFT/RIGHT	Determines whether the position information is displayed in the left or right part of the frame line.
POSITION TEXT	Causes the position information to be displayed in text form. The information is displayed in the top and/or bottom frame line with the words More,Top and Bottom. The text is language-dependent and may also be displayed in another language if the language code is set accordingly.
POSITION TEXT MORE	Suppresses the words Top and Bottom and only displays the word More where applicable, i.e., in the top or bottom frame line or both.
LEFT/RIGHT	Determines whether the position information is displayed in the left or right part of the top frame line.
POSITION OFF	Causes the position information to be suppressed; no position information will be displayed.

Protection of Input Fields in a Window

The following rules apply to input fields (with `AD=A` or `AD=M`) which are not entirely within the window:

- Input fields whose beginning is not inside the window are always made protected.
- Input fields which begin inside and end outside the window are only made protected if the values they contain cannot be displayed completely in the window. Please note that in this case it is decisive whether the *value length*, not the *field length*, exceeds the window size. Filler characters (as specified with the profile parameter `FC`) do not count as part of the value.

If you wish to access input fields thus protected, you have to adjust the window size accordingly so that the beginning of the field/end of the value is within the window.

Invoking Different Windows

A `DEFINE WINDOW` statement must not be placed within a logical condition statement block. To invoke different windows depending on a condition, use different `SET WINDOW` statements (or `INPUT` statements with a `WINDOW` clause respectively) in a condition.

Example

```
** Example 'DWDEX1': DEFINE WINDOW
************************************************************************
DEFINE DATA LOCAL
01 #I (P3)
END-DEFINE
*
SET KEY PF1='%W<<' PF2='%W>>' PF4='%W--' PF5='%W++'
*
DEFINE WINDOW WIND1
       SIZE QUARTER
       BASE TOP RIGHT
       FRAMED ON POSITION SYMBOL AUTO
*
SET WINDOW 'WIND1'
FOR #I = 1 TO 10
  WRITE 25X #I 'THIS IS SOME LONG TEXT' #I
END-FOR
*
END
```

Output of Program DWDEX1:

```
                                        +-----------------------More:     + >+
> r                                     ! Page      1                        !
All    ....+....1....+....2....+....3.. !                                    !
  0010 ** Example 'DWDEX1': DEFINE WIND !                             1 THIS  !
  0020 ******************************** !                             2 THIS  !
  0030 DEFINE DATA LOCAL                !                             3 THIS  !
  0040 01 #I (P3)                       !                             4 THIS  !
  0050 END-DEFINE                       !                             5 THIS  !
  0060 *                                !                             6 THIS  !
  0070 SET KEY PF1='%W<<' PF2='%W>>' PF !                             7 THIS  !
  0080 *                                ! MORE                               !
  0090 DEFINE WINDOW WIND1              +------------------------------------+
  0100        SIZE QUARTER
  0110        BASE TOP RIGHT
  0120        FRAMED ON POSITION SYMBOL AUTO
  0130 *
  0140 SET WINDOW 'WIND1'
  0150 FOR #I = 1 TO 10
  0160   WRITE 25X #I 'THIS IS SOME LONG TEXT' #I
  0170 END-FOR
  0180 *
  0190 END
  0200
       ....+....1....+....2....+....3....+....4....+....5....+... S 19   L 1
```

58 DEFINE WORK FILE

DEFINE WORK FILE *work-file-number* { *operand1* [TYPE *operand2*] | TYPE *operand2* } [ATTRIBUTES {*operand3*}...]

Note: The elements shown in square brackets [...] are optional, however, at least one of them must be specified with this statement.

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

Related Statements: CLOSE WORK FILE | READ WORK FILE | WRITE WORK FILE

Belongs to Function Group: ***Control of Work Files / PC Files***

Function

The statement DEFINE WORK FILE is used to assign a file name to a Natural work file number within a Natural application. This allows you to make or change work file assignments dynamically within a Natural session or overwrite work file assignments made at another level. See also *Work Files* in the *Operations* documentation.

When this statement is executed and the specified work file is already open, the statement will implicitly close the work file.

Note: For Unicode and code page support on Windows and Linux platforms, see *Work Files and Print Files* in the *Unicode and Code Page Support* documentation.

Syntax Description

Operand Definition Table:

Operand	Possible Structure	Possible Formats	Referencing Permitted	Dynamic Definition
operand1	C S	A U	yes	no
operand2	C S	A U	yes	no
operand3	C S	A U	yes	no

Note: If a format U operand is specified in Unicode (UTF-16), it is converted to session code page characters before it is evaluated.

Syntax Element Description:

<table>
<tr><th>Syntax Element</th><th colspan="2">Description</th></tr>
<tr><td rowspan="1">work-file-number</td><td colspan="2">Work File Number:

The work file number is to be specified.

The work file number is either

- a numeric constant in the value range (1:32) or
- a numeric variable of type (B/N/P/I) defined with a CONST clause assigning a value in range (1:32). Variable is a scalar (non-array) without precision digits for type (N/P), length in between 1-4 for type (B), and no redefinition field.

This is the number to be used in a WRITE WORK FILE, READ WORK FILE or CLOSE WORK FILE statement.</td></tr>
<tr><td>operand1</td><td colspan="2">Work File Name:

operand1 is the name of the work file.

The file name (operand1) may contain environment variables. It is possible to use physical work file names. If a file with the specified name does not exist, it will be created.

If operand1 is not specified, the value of operand1 is determined by taking the work file name stored with the Configuration Utility in the parameter file for the corresponding work file number.

Note: If operand1 is not specified, the behavior of Natural for Mainframes and Natural for Windows/Linux is different.</td></tr>
<tr><td rowspan="3">TYPE operand2</td><td colspan="2">TYPE Clause:

operand2 specifies the type of work file. See also Handling of Large and Dynamic Variables in the description of the WRITE WORK FILE statement.

The value of operand2 is handled in a case insensitive way and must be enclosed in quotes or provided in an alphanumeric variable.</td></tr>
<tr><td>DEFAULT</td><td>Determines the file type from the extension.

Format: Depends on the work file type.

Note: The file type TRANSFER cannot be determined by the work file type DEFAULT. You must explicitly define TRANSFER as the file type you wish to use.</td></tr>
<tr><td>TRANSFER</td><td>Is used to transfer data to and from a PC with Entire Connection or NaturalONE.

This work file type represents a data connection between a Natural session on Linux and an Entire Connection terminal or NaturalONE on a PC.

Format: ENTIRE CONNECTION</td></tr>
</table>

Syntax Element	Description	
		Note: 1. This work file type cannot be used in conjunction with the *ATTRIBUTES Clause*. 2. This work file type is not available under Windows.
	`SAG`	Format: binary
	`ASCII`	Files in ASCII are "text" files with records terminated by [a carriage return] line feed. Format: ASCII
	`ASCII-COMPRESSED`	Is a file in ASCII format, with the exception that all trailing blanks are removed. Format: ASCII
	`ENTIRECONNECTION`	With this work file type, you can read and write (using the statements `READ` and `WRITE`, for example) directly from/to a work file in Entire Connection format on the local disc. Format: `ENTIRE CONNECTION` **Note:** This work file type is available on PCs and on Linux. No transfer to PC is possible. The Entire Connection terminal is not used in this process.
	`UNFORMATTED`	A completely unformatted file. No formatting information is written (neither for fields nor for records). Format: `UNFORMATTED`
	`PORTABLE`	Files which can handle dynamic variables exactly and can also be transported: for example, from a Little Endian machine to a Big Endian machine, and vice versa. Format: `PORTABLE`
	`CSV`	Comma-separated values. Each record is written to one line in the file. By default, a header is not written. The default character which is used to separate the data fields is a semicolon (;). For further information, see *Work Files* in the *Configuration Utility* documentation.
`ATTRIBUTES {operand3}...`	**ATTRIBUTES Clause:** *operand3* specifies a work file attribute. Several attributes separated by a comma or a blank can be specified, for example:	

Syntax Element	Description
	(continued)

```
DEFINE WORK FILE ATTRIBUTES 'APPEND,KEEP'
```

If multiple values for the same attribute type are specified, the last value is taken, for example:

```
DEFINE WORK FILE ATTRIBUTES 'APPEND,NOAPPEND'
```

In this case, `NOAPPEND` will be taken.

Example for `BOM/NOBOM` usage:

```
...
DEFINE WORK FILE 11 'x.tmp' ATTRIBUTES 'BOM'
*
* write work file with BOM
*
DEFINE WORK FILE 11 'x.tmp' ATTRIBUTES 'NOBOM'
*
* write work file without BOM
...
```

Note: If *operand3* is omitted, the corresponding value defined in the parameter file, as created by the Configuration Utility, is implicitly used.

The following is an overview of the attribute types and their possible values:

Append Mode:	
`NOAPPEND`	Deactivates the append mode. The file is rewritten from the start. This is the default value.
`APPEND`	Activates the append mode. In this mode, new records are added at the end of the file.
Keep/Delete File after Work File Close:	
`DELETE`	The work file is deleted after a close work file operation.
`KEEP`	The work file is kept after a close work file operation. This is the default value.
Write Byte Order Mark (BOM):	
`BOM`	A byte order mark is written in front of the work file data. Only available for the work file types which write code page data: `ASCII`, `ASCII-COMPRESSED`, `UNFORMATTED` and `CSV`. For these work file types, the attribute `BOM` can only be set, if the code page UTF-8 is defined for the work file (see the description of the `TYPE` clause). If a work file of another type is written or a code page other than UTF-8 is defined, the specification of the attribute `BOM` is ignored during runtime.

Syntax Element	Description	
		See also *Work Files and Print Files on Windows and Linux Platforms* in the *Unicode and Code Page Support* documentation.
	NOBOM	No byte order mark is written in front of the work file data. This is the default value.
	Remove/Keep Carriage Return:	
	KEEPCR	Carriage return characters are kept when reading an ASCII work file. This attribute is only relevant for ASCII work files. If a work file of another type than ASCII or ASCII-COMPRESSED is read, the specification of the attribute KEEPCR is ignored during runtime. **Caution:** Use KEEPCR with care. ASCII format is only recommended for alphanumeric data. Binary data should not be processed with ASCII work files. When you use KEEPCR, the work file record may include carriage return characters. The use of KEEPCR only makes sense when reading ASCII work files which have been written on Linux. It does not make sense to use KEEPCR with ASCII work files which have been written on Windows.
	REMOVECR	Carriage return characters are removed when reading an ASCII work file. This is the default value. This attribute is only relevant for ASCII work files. If a work file of another type than ASCII or ASCII-COMPRESSED is read, the specification of the attribute REMOVECR is ignored during runtime.

VIII

59 DELETE

```
DELETE [RECORD] [IN] [STATEMENT] [(r)]
```

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

Related Statements: ACCEPT/REJECT | AT BREAK | AT START OF DATA | AT END OF DATA | BACKOUT TRANSACTION | BEFORE BREAK PROCESSING | END TRANSACTION | FIND | GET | GET SAME | GET TRANSACTION DATA | HISTOGRAM | LIMIT | PASSW | PERFORM BREAK PROCESSING | READ | RETRY | STORE | UPDATE

Belongs to Function Group: ***Database Access and Update***

Function

The DELETE statement is used to delete a record from a database.

Hold Status

The use of the DELETE statement causes each record selected in the corresponding FIND or READ statement to be placed in exclusive hold.

Record hold logic is explained in the section *Database Update - Transaction Processing* (in the *Programming Guide*).

Restriction

A DELETE statement cannot be specified in the same statement line as a FIND, READ, or GET statement.

Syntax Description

Syntax Element	Description
(*r*)	**Statement Reference:** The notation (*r*) is used to reference the statement which was used to select/read the record to be deleted. If no statement reference is specified, the DELETE statement will reference the innermost active processing loop in which a database record was selected/read.

Database-Specific Considerations

SQL Databases	The `DELETE` statement is used to delete a row from the database table. It corresponds with the SQL statement `DELETE WHERE CURRENT OF CURSOR-NAME`, that is, only the row which was read last can be deleted. With most SQL databases, a row that was read with a `FIND SORTED BY` or `READ LOGICAL` statement cannot be deleted.
XML Databases	The `DELETE` statement is used to delete an XML object from a database. For XML databases, this implies that only the record which was read last can be deleted.

Examples

- Example 1
- Example 2

Example 1

In this example, all records with the name `ALDEN` are deleted.

```
** Example 'DELEX1': DELETE
**
**
CAUTION: Executing this example will modify the database records!
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
END-DEFINE
*
FIND EMPLOY-VIEW WITH NAME = 'ALDEN'
  /*
  DELETE
  END TRANSACTION
  /*
  AT END OF DATA
    WRITE NOTITLE *NUMBER 'RECORDS DELETED'
  END-ENDDATA
END-FIND
END
```

Example 2

If no records are found in the `VEHICLES` file for the person named ALDEN, the `EMPLOYEE` record for ALDEN is deleted.

```
** Example 'DELEX2': DELETE
**
**
CAUTION: Executing this example will modify the database records!
***********************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
1 VEHIC-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
END-DEFINE
*
EMPL. FIND EMPLOY-VIEW WITH NAME = 'ALDEN'
  /*
 VEHC. FIND VEHIC-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (EMPL.)
    IF NO RECORDS FOUND
      /*
      DELETE (EMPL.)
      /*
      END TRANSACTION
    END-NOREC
  END-FIND
  /*
END-FIND
END
```

60 DELETE (SQL)

Belongs to Function Group: ***Database Access and Update***

Function

The SQL `DELETE` statement is used to delete either rows in a table without using a cursor (**“searched”** `DELETE`) or rows in a table to which a cursor is positioned (**“positioned”** `DELETE`).

Two different structures are possible.

Syntax 1 - Searched DELETE

The “searched” `DELETE` statement is a stand-alone statement not related to any `SELECT` statement. With a single statement you can delete zero, one, multiple or all rows of a table. The rows to be deleted are determined by a *`search-condition`* that is applied to the table. Optionally, the table name can be assigned a *`correlation-name`*.

Note: The number of rows that have actually been deleted with a “searched” `DELETE` can be ascertained by using the system variable `*ROWCOUNT`; see *System Variables* documentation.

Common Set Syntax:

```
DELETE FROM table-name [correlation-name] [WHERE search-condition]
```

Extended Set Syntax:

```
DELETE FROM table-name [correlation-name]
[WHERE search-condition]
[            { RR }  ]
[   WITH     { RS }  ]   [QUERYNO integer]
[            { CS }  ]
```

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

Syntax Element Description:

Syntax Element	Description
`FROM` *`table-name`*	**FROM Clause:** Specifies the table from which the rows are to be deleted.
`correlation-name`	**Correlation Name:** Optional. The table name can be assigned a *`correlation-name`*.
`WHERE` *`search-condition`*	**WHERE Clause:** Specifies the selection criteria for the rows to be deleted. If no `WHERE` clause is specified, the entire table is deleted.
`WITH`	**WITH Isolation Level Clause:** Enables the explicit specification of the isolation level used when locating the row to be deleted. This clause belongs to the **SQL Extended Set**. It is only valid against Db2 databases. When used against other databases, it will cause runtime errors. `CS` – Cursor Stability `RR` – Repeatable Read `RS` – Read Stability
`QUERYNO` *`integer`*	**QUERYNO Clause:** This clause belongs to the **SQL Extended Set**. This clause is not currently supported and will be ignored.

Syntax 2 - Positioned DELETE

The "positioned" `DELETE` statement always refers to a cursor within a database loop. Therefore the table referenced by a positioned `DELETE` statement must be the same as the one referenced by the corresponding `SELECT` statement, otherwise an error message is returned. A positioned `DELETE` cannot be used with a non-cursor selection.

The functionality of the positioned `DELETE` statement corresponds to that of the **"native" Natural `DELETE` statement**.

```
DELETE FROM table-name WHERE CURRENT OF CURSOR [(r)]
```

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

Syntax Element Description:

Syntax Element	Description
`FROM table-name WHERE CURRENT OF CURSOR`	**FROM Clause:** This clause specifies the table from which the rows are to be deleted.
`(r)`	**Statement Reference:** The `(r)` notation is used to reference the statement which was used to select the row to be deleted. If no statement reference is specified, the `DELETE` statement is related to the innermost active processing loop in which a database record was selected.
`FOR ROW ... OF ROWSET`	**FOR ROW ... OF ROWSET Clause:** This clause belongs to the **SQL Extended Set**. The optional `FOR ROW ... OF ROWSET` clause for positioned `SQL DELETE` statements specifies which row of the current rowset has to be deleted. It should only be specified if the `DELETE` statement is related to a `SELECT` statement which uses rowset positioning and which has column arrays in its `INTO` clause, see ***into-clause***. If this clause is omitted, all rows of the current rowset are deleted.

61 DISPLAY

```
DISPLAY [(rep)] [options] {[/ ...] [output-format] output-element} ...
```

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

Related Statements: AT END OF PAGE | AT TOP OF PAGE | CLOSE PRINTER | DEFINE PRINTER EJECT | FORMAT | NEWPAGE | PRINT | SKIP | SUSPEND IDENTICAL SUPPRESS | WRITE | WRITE TITLE | WRITE TRAILER

Belongs to Function Group: ***Creation of Output Reports***

Function

The DISPLAY statement is used to specify the fields to be output on a report in column format. A column is created for each field and a field header is placed over the column.

Note: The statements WRITE and PRINT can be used to produce output in free (non-column) format.

See also the following topics (in the *Programming Guide*):

- *Report Format and Control*
- *Statements DISPLAY and WRITE*
- *Index Notation for Multiple-Value Fields and Periodic Groups*
- *Column Headers*
- *Layout of an Output Page*

Syntax Description

Syntax Element	Description
(*rep*)	**Report Specification:** The notation (*rep*) may be used to specify the identification of the report for which the DISPLAY statement is applicable. As report identification, a value in the range 0 - 31 or a logical name which has been assigned using the DEFINE PRINTER statement may be specified. If (*rep*) is not specified, the statement will apply to the first report (Report 0). If this printer file is defined to Natural as PC, the report will be downloaded to the PC, see ***Example 8***.

Syntax Element	Description
	For information on how to control the format of an output report created with Natural, see *Report Format and Control* in the *Programming Guide*.
options	**Display Options:** For details, see ***Display Options*** below.
output-format	**Output Format Definitions:** For details, see ***Output Format Definitions*** below.
/	**Line Advance - Slash Notation:** When specified within a text element, a slash (/) causes a line advance for the text displayed. When specified between output elements, it causes the output element specified by the slash (/) to be placed vertically within the same column. The header for this column will be constructed by placing the headers of the vertically displayed elements vertically above the column. See also the following topics in the *Programming Guide*: ■ *Line Advance - Slash Notation* ■ *Example 1 - Line Advance in DISPLAY Statement* ■ *Suppressing Column Headers - Slash Notation*
output-element	**Output Element:** For details, see ***Output Element*** below.

Display Options

```
[NOTITLE] [NOHDR] [ [AND] [GIVE] [SYSTEM] FUNCTIONS ] [(statement-parameters)]
```

Syntax Element Description:

Syntax Element	Description
NOTITLE	**Default Page Title Suppression:** By default, Natural generates a single title line for each page resulting from a `DISPLAY` statement. This title contains the page number, the time of day, and the date. Time of day is set at the beginning of the program execution or at the beginning of the job (batch mode). The default title line may be overridden by using a `WRITE TITLE` statement, or it may be suppressed by specifying the keyword `NOTITLE` in the `DISPLAY` statement. Examples:

Syntax Element	Description

	■ Default title will be produced:
	`DISPLAY NAME`
	■ User title will be produced:
	`DISPLAY NAME WRITE TITLE 'user-title'`
	■ No title will be produced:
	`DISPLAY NOTITLE NAME`
	Note: If the NOTITLE option is used, it applies to all DISPLAY, PRINT and WRITE statements within the same object which write data to the same report.

NOHDR	**Column Headers:**
	Column headers are produced for each field specified in the DISPLAY statement using the following rules:
	■ The header text may be explicitly specified in the DISPLAY statement before the field name. For example:
	`DISPLAY 'EMPLOYEE' NAME 'SALARY' SALARY`
	■ If you do not specify an explicit header for a field, the **header** as defined in the DEFINE DATA statement will be used.
	■ If for a database field no header is defined in the DEFINE DATA statement, the default header as defined in the DDM will be used.
	■ If no default header is defined in the DDM, the field name will be used as header.
	■ If for a user-defined variable no header is defined in the DEFINE DATA statement, the variable name will be used as header. See also the DEFINE DATA statement for **header definition**.
	`DISPLAY NAME SALARY #NEW-SALARY`
	■ Natural always underlines column headings and generates one blank line between the underlining and the data being displayed.
	■ If there are multiple DISPLAY statements in a program, the first DISPLAY statement determines the column header(s) to be used; this is evaluated at compilation time.
	Column Header Suppression:
	To suppress the column header for a single field

Syntax Element	Description

	■ Specify the following characters (apostrophe-slash-apostrophe) before the field name:
	`'/'`
	For example:
	`DISPLAY '/' NAME 'SALARY' SALARY`
	To suppress all column headers
	■ Specify the keyword `NOHDR`:
	`DISPLAY NOHDR NAME SALARY`
	Note:
	1. `NOHDR` only takes effect for the first `DISPLAY` statement, as subsequent `DISPLAY` statements cannot create column headers anyhow.
	2. If both `NOTITLE` and `NOHDR` are used, they must be specified in the following order: `DISPLAY NOTITLE NOHDR NAME SALARY`

`GIVE SYSTEM FUNCTIONS`	**Natural System Function Usage:** The `GIVE SYSTEM FUNCTIONS` clause is used to make available the following Natural system functions: `AVER`, `COUNT`, `MAX`, `MIN`, `NAVER`, `NCOUNT`, `NMIN`, `SUM`, `TOTAL`. These are evaluated when the `DISPLAY` statement containing the `GIVE SYSTEM FUNCTIONS` clause is executed. These functions may then be referred to in a statement executed as a result of an end-of-page condition. **Note:** 1. Only one `DISPLAY` statement per report may contain a `GIVE SYSTEM FUNCTIONS` clause. When system functions are evaluated from a `DISPLAY` statement, they are evaluated on a page basis, which means that all functions (except `TOTAL`) are reset to zero when a new page is initiated. 2. When system functions are used within a `DISPLAY` statement within a subroutine, the end-of-page processing must occur within the same routine. 3. In place of the keyword `GIVE`, the keyword `GIVING` may be used. See also *Example 2 - DISPLAY Statement Using GIVE SYSTEM FUNCTIONS Clause*.
statement-parameters	**Parameter Definition at Statement Level:**

Syntax Element	Description
	One or more parameters, enclosed within parentheses, may be specified at statement level, that is, immediately after the `DISPLAY` statement. Each parameter specified will override the corresponding parameter previously specified in a `GLOBALS` command, `SET GLOBALS` (Reporting Mode only) or `FORMAT` statement. If more than one parameter is specified, they must be separated by one or more blanks from one another. Each parameter specification must not be split between two statement lines. **Note:** The parameter settings applied here will only be regarded for variable fields, but they have no effect on text-constants. If you would like to set field attributes for a text-constant, they have to be set explicitly for this element, see *Parameter Definition at Element (Field) Level*. See also: ■ *List of Parameters* ■ *Example of Parameter Usage at Statement and Element (Field) Level* ■ *Example 7 - DISPLAY Statement Using Parameters on Statement/Element Level*

List of Parameters

The following parameters can be specified with the `DISPLAY` statement

Parameter Name	Explanation	Specification possible at statement level (S), at element level (E) or both (SE)
`AD`	Attribute Definition	SE
`AL`	Alphanumeric Length for Output	SE
`CD`	Color Definition	SE
`CV`	Control Variable	SE
`DF`	Date Format	SE
`DL`	Display Length for Output	SE
`DY`	Dynamic Attributes	SE
`EM`	Edit Mask	SE
`EMU`	Unicode Edit Mask	E
`ES`	Empty Line Suppression	S
`FC`	Filler Character	SE
`FL`	Floating Point Mantissa Length	SE
`GC`	Filler Character for Group Headers	SE

Parameter Name	Explanation	Specification possible at statement level (S), at element level (E) or both (SE)
HC	Header Centering	SE
HW	Heading Width	SE
IC	Insertion Character	SE
ICU	Unicode Insertion Character	SE
IS	Identical Suppress	SE
LC	Leading Characters	SE
LCU	Unicode Leading Characters	SE
LS	Line Size	S
MC	Multiple-Value Field Count	S
MP	Maximum Number of Pages of a Report	S
NL	Numeric Length for Output	SE
PC	Periodic Group Count	S
PM	Print Mode	SE
PS	Page Size	S
SF	Spacing Factor	SE
SG	Sign Position	SE
TC	Trailing Characters	SE
TCU	Unicode Trailing Characters	SE
UC	Underlining Character	SE
ZP	Zero Printing	SE

The individual parameters are described in the *Parameter Reference* (session parameters).

See also the following topics in the *Programming Guide*:

- *Centering of Column Headers - HC Parameter*
- *Width of Column Headers - HW Parameter*
- *Filler Characters for Headers - Parameters FC and GC*
- *Underlining Character for Titles and Headers - UC Parameter*

Example of Parameter Usage at Statement and Element (Field) Level

```
DEFINE DATA LOCAL
1 VARI (A4)     INIT <'1234'>                          /*     Output
END-DEFINE                                             /*    Produced
*                                                      /*    ---------
DISPLAY NOHDR           'Text'          '='    VARI        /*    Text 1234
DISPLAY NOHDR (AD=U)    'Text'          '='    VARI        /*    Text 1234
DISPLAY NOHDR           'Text' (AD=U)   '='    VARI (AD=U)/*    Text 1234
DISPLAY NOHDR           'Text' (AD=U)   '='    VARI        /*    Text 1234
END
```

Output Format Definitions

```
[ nX            ]
[ nT            ]
[ x/y           ]  [ { 'text'  }    [(attributes)] ]   ...
[ T*field-name  ]  [ { 'c'(n)  }                   ]
[ P*field-name  ]

[ VERTICALLY      [ AS { 'text' [(attributes)] [CAPTIONED] } ]  [/ ...] ]
[                 [    { [CAPTIONED]                       } ]          ]
[ [HORIZONTALLY]                                                        ]
```

Field Positioning Notations

Syntax Element	Description
*n*X	**Column Spacing:** This notation inserts *n* spaces between columns. Example: `DISPLAY NAME 5X SALARY` See also: ■ *Example 1 - DISPLAY Statement Using nX and nT Notation* (below) ■ *Column Spacing - SF Parameter and nX Notation* (in the *Programming Guide*)
*n*T	**Tab Setting:** The *n*T notation causes positioning (tabulation) to display position *n*. Backward positioning is not permitted.

Syntax Element	Description
	In the following example, `NAME` is displayed beginning in position 25, and `SALARY` beginning in position 50: `DISPLAY 25T NAME 50T SALARY` See also: ■ *Example 1 - DISPLAY Statement Using nX and nT Notation* (below) ■ *Tab Setting - nT Notation* (in the *Programming Guide*)
x/*y*	**x/y Positioning:** The *x*/*y* notation causes the next element to be placed *x* lines below the output of the last statement, beginning in column *y*. *y* must not be zero. Backward positioning is not permitted.
T**field-name*	**Field Related Positioning:** The `T*` notation is used to position to a specific print position of a field used in a previous `DISPLAY` statement. Backward positioning is not permitted.
P**field-name*	**Field and Line Related Positioning:** The `P*` notation is used to position to a specific print position and line of a field used in a previous `DISPLAY` statement. It is most often used in conjunction with vertical display mode. Backward positioning is not permitted. See also: ■ *Example 3 - DISPLAY Statement Using P* Notation* (below) ■ *Tab Notation P*field* (in the *Programming Guide*)

Override Column Heading Assignment

Syntax Element	Description
'*text*' '/'	**Text Assignment:** If placed immediately before a field, the text enclosed by single quotes overrides the column heading. The slash character `'/'` before a field causes the header for the field to be suppressed.

Syntax Element	Description
	`DISPLAY 'EMPLOYEE' NAME 'MARITAL/STATUS' MAR-STAT` If multiple '*text*' elements are specified before a field name, the *last* '*text*' element will be used as the column header and the other text elements will be placed before the value of the field within the column. See also: ■ *Define Your Own Column Headers* (in the *Programming Guide*) ■ *Text Notation, Defining a Text to Be Used with a Statement* (in the *Programming Guide*) ■ ***Example 4 - DISPLAY Statement Using 'text', 'c(n)' and Attribute Notation*** (below)

'*c*'(*n*)	**Character Repetition:**
	
	The character enclosed by single quotes is displayed *n* times immediately before the field value. For example:
	
	`DISPLAY '*' (5) '=' NAME`
	
	results in
	
	`***** SMITH`
	
	See also:
	
	■ *Text Notation, Defining a Character to Be Displayed n Times before a Field Value* (in the *Programming Guide*)
	■ ***Example 4 - DISPLAY Statement Using 'text', 'c(n)' and Attribute Notation*** (below)

Output Attributes

attributes indicates the output attributes to be used for text display. Attributes can be:

```
{ AD=ad-value ... }
{ CD=cd-value     }
{ PM=pm-value ... } ...

{ ad-value }
{ cd-value } ...
```

Where:

ad-value, *cd-value* and *pm-value* denote the possible values of the corresponding session parameters AD, CD and PM described in the relevant sections of the *Parameter Reference* documentation.

The compiler actually accepts more than one attribute value for an output field. For example, you can specify: `AD=BDI`. In such a case, however, only the last value applies. In the given example, only the value `I` becomes effective and the output field is displayed intensified.

For an alphanumeric/Unicode constant (Natural data format A or U), you can specify *`ad-value`* and/or *`cd-value`* without preceding `CD=` or `AD=`, respectively. The single value entered is then checked against all possible `CD` values first. For example: a value of `IRE` will be interpreted as intensified/red but not as intensified/right-justified/mandatory. You cannot combine a single *`cd-value`* or *`ad-value`* with a value preceded by `CD=` or `AD=`.

Vertical/Horizontal Display

The `VERT` clause may be used to cause multiple field values to be positioned underneath one another in the same column. In vertical mode, a new column may be initiated by specifying the keyword `VERT` or `HORIZ`.

The column heading in vertical mode is controlled using the entry or entries specified with the `AS` clause as described below.

Syntax Element	Description
`VERTICALLY`	**DISPLAY VERT without AS Clause:** Vertical column orientation. No column heading is produced if the `AS` clause is omitted. `DISPLAY VERT NAME SALARY` For an example, see *DISPLAY VERT without AS Clause* in the *Programming Guide*.
`AS '`*`text`*`'`	**DISPLAY VERT AS '*`text`*' Clause:** Vertical column orientation. If `AS '`*`text`*`'` is specified, the text enclosed by single quotes is used as the column heading. For an example, see *DISPLAY VERT AS 'text'* in the *Programming Guide*. The slash character / in the character string of `'`*`text`*`'` will cause multiple lines of column headings. `DISPLAY VERT AS 'LAST/NAME' NAME`

Syntax Element	Description
`AS 'text' CAPTIONED`	**DISPLAY VERT AS '*text*' CAPTIONED Clause:** Vertical column orientation. If `AS 'text' CAPTIONED` is specified, `'text'` is used as the column heading and the standard heading text or field name is inserted immediately before the field value in each detail display line. `DISPLAY VERT AS 'PERSONS/SELECTED' CAPTIONED NAME FIRST-NAME` For an example, see *DISPLAY VERT AS 'text' CAPTIONED* in the *Programming Guide*.
`AS CAPTIONED`	**DISPLAY VERT AS CAPTIONED Clause:** Vertical column orientation. If `AS CAPTIONED` is specified, the standard heading text for the field (either heading text or the field name) will be used as the column heading. `DISPLAY VERT AS CAPTIONED NAME FIRST-NAME`
`HORIZONTALLY`	**DISPLAY HORIZ Clause:** Horizontal column orientation. This is the default display mode.

Vertical and horizontal column orientation may be intermixed by using the respective keyword.

To suspend vertical display for a single output element, you may place a dash (-) in front of the element. For example:

```
DISPLAY VERT NAME - FIRST-NAME SALARY
```

In the above example, `FIRST-NAME` will be output horizontally next to `NAME`, while `SALARY` will be output vertically again, i.e. below `NAME`.

The standard display mode is horizontal. A column is constructed for each field to be displayed.

Column headings are obtained and used by Natural according to the following priority:

1. heading `'text'` supplied in the `DISPLAY` statement;
2. the default heading defined in the DDM (database fields), or the name of a user-defined variable;
3. the field name as defined in the DDM (if no heading text was defined for the database field).

For group names, a group heading is produced for the entire group. When specifying a group, only the heading for the entire group may be overridden by a user-specified heading.

The maximum number of column header lines is 15.

Line size overflow is not permitted for output resulting from a `DISPLAY` statement. If a line overflow occurs, an error message is issued.

For more information about vertical/horizontal display usage, see:

- *Example 5 - DISPLAY Statement Using Horizontal Display*
- *Example 6 - DISPLAY Statement Using Vertical and Horizontal Display*
- *DISPLAY VERT AS CAPTIONED and HORIZ* (in the *Programming Guide*)

Output Element

```
[ { 'text'  [(attributes)]   } ...  ]
[ { 'c'(n) [(attributes)]    }      ]
[   nX                              ]  ['='] {operand1 [(parameters)]}
[   nT                              ]
[   x/y                             ]
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
operand1		S	A	G	N	A	N	P	I	F	B	D	T	L		G	O	yes	no

Syntax Element Description

Syntax Element	Description
nX	**Column Spacing:** This is the same as under *Output Format Definitions* (**see above**).
nT	**Tab Setting:** This is the same as under *Output Format Definitions* (**see above**).
x/y	**x/y Positioning:** This is the same as under *Output Format Definitions* (**see above**).
'*text*'	**Text Assignment:** This is the same as under *Output Format Definitions* (**see above**).
'*c*'(*n*)	**Character Repetition:** This is the same as under *Output Format Definitions* (**see above**).

Syntax Element	Description

`'text' '='`
 	
`'c' (n) '='`
	If `'text' '='` is placed immediately before the field, `text` is output immediately before the field value. This applies analogously with `'c' (n) '='`.
	
	`DISPLAY '*****' '=' NAME`

`attributes`	**Output Attributes:** This is the same as under *Output Attributes* (**see above**).
`operand1`	The field to be displayed.
`parameters`	**Parameter Definition at Element (Field) Level:** One or more parameters, enclosed within parentheses, may be specified at element (field) level, that is, immediately after `operand1`. Each parameter specified in this manner will override the corresponding parameter previously specified **at statement level** or in a `GLOBALS` command, `SET GLOBALS` (in Reporting Mode only) or `FORMAT` statement. If more than one parameter is specified, one or more blanks must be placed between each entry. An entry must not be split between two statement lines. See also: ■ *List of Parameters* ■ *Example of Parameter Usage at Statement and Element (Field) Level*

Defaults Applicable for a DISPLAY Statement

The following defaults are applicable for a `DISPLAY` statement:

- **Report Width**
  The width of the report defaults to the value set when Natural is installed. This default value is normally 132 in batch mode or the line length of the terminal in TP mode. It may be overridden with the session parameter `LS`. In TP mode, line size (`LS`) and page size (`PS`) parameters are set by Natural based on the physical characteristics of the terminal type in use.
- **Terminal Screen Output**
  When the `DISPLAY` output is displayed on a terminal (emulation) screen, the output begins in physical Column 2 (because Column 1 must be reserved for possible use as an attribute position on a 3270-type terminal).
- **Printout on Paper**
  When the `DISPLAY` output is printed on paper, the printout begins in the leftmost column (Column 1).

- **Spacing Factor**
  The default spacing factor between elements is one position. There is a minimum of one space between columns (reserved for terminal attributes). This default may be overridden with the session parameter `SF`.
- **Field Output**
  The length of the field or the field heading, whichever is greater, determines the column width for the report (unless the `HW` parameter is used).
  - If the field is longer than the heading, the heading will be centered over the column unless the `HC=L` or `HC=R` parameter is used to produce a left-justified or right-justified heading.
  - If the heading is longer than the field, the field will be left-justified under the heading.
  - The values contained in the field are left-justified for alphanumeric fields and right-justified for numeric fields.
  - Numeric fields may be displayed left-justified by specifying `AD=L`.
  - Alphanumeric fields may be displayed right-justified by specifying `AD=R`.
  - In a vertical display, the longest data value or heading among all fields determines the column width (unless the `HW` parameter is used).
- **Sign**
  One extra high-order print position is reserved for a sign when printing a numeric field. The session parameter `SG` may be used to suppress the sign position.
- **Page Overflow**
  Page overflow is checked before execution of a `DISPLAY` statement. No new page title or trailer information is generated during the execution of a `DISPLAY` statement.

Examples

- Example 1 - DISPLAY Statement Using nX and nT Notation
- Example 2 - DISPLAY Statement Using GIVE SYSTEM FUNCTIONS Clause
- Example 3 - DISPLAY Statement Using P* Notation
- Example 4 - DISPLAY Statement Using 'text ', 'c(n)' and Attribute Notation
- Example 5 - DISPLAY Statement Using Horizontal Display
- Example 6 - DISPLAY Statement Using Vertical and Horizontal Display
- Example 7 - DISPLAY Statement Using Parameters on Statement/Element Level

- Example 8 - Report Specification with Output File Defined to Natural as PC

Example 1 - DISPLAY Statement Using nX and nT Notation

```
** Example 'DISEX1': DISPLAY (with nX, nT notation)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
END-DEFINE
*
LIMIT 4
READ EMPL-VIEW BY NAME
  DISPLAY NOTITLE 5X NAME 50T JOB-TITLE
END-READ
*
END
```

Output of Program DISEX1:

```
        NAME                                             CURRENT
                                                         POSITION
--------------------                            -------------------------

ABELLAN                                         MAQUINISTA
ACHIESON                                        DATA BASE ADMINISTRATOR
ADAM                                            CHEF DE SERVICE
ADKINSON                                        PROGRAMMER
```

Example 2 - DISPLAY Statement Using GIVE SYSTEM FUNCTIONS Clause

```
** Example 'DISEX2': DISPLAY (with GIVE SYSTEM FUNCTIONS)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
  2 SALARY    (1)
  2 CURR-CODE (1)
END-DEFINE
*
LIMIT 15
FORMAT PS=15
*
READ EMPLOY-VIEW
  DISPLAY GIVE SYSTEM FUNCTIONS
          PERSONNEL-ID NAME FIRST-NAME SALARY (1) CURR-CODE (1)
  AT END OF PAGE
```

```
    WRITE /    'SALARY STATISTICS:'
          / 7X 'MAXIMUM:' MAX(SALARY(1))  CURR-CODE (1)
          / 7X 'MINIMUM:' MIN(SALARY(1))  CURR-CODE (1)
          / 7X 'AVERAGE:' AVER(SALARY(1)) CURR-CODE (1)
  END-ENDPAGE
END-READ
*
END
```

Output of Program DISEX2:

```
Page      1                                                  05-01-12  09:47:48

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

50005500  BLOND                ALEXANDRE                172000 FRA
50005300  MAIZIERE             ELISABETH                166900 FRA
50004900  CAOUDAL              ALBERT                   167350 FRA
50004600  VERDIE               BERNARD                  170100 FRA
50004200  VAUZELLE             BERNARD                  159790 FRA
50004100  CHAPUIS              ROBERT                   169900 FRA
50003800  JOUSSELIN            DANIEL                   171990 FRA
50006900  BAILLET              PATRICK                  188000 FRA
50007600  MARX                 JEAN-MARIE               365700 FRA

SALARY STATISTICS:
       MAXIMUM:     365700 FRA
       MINIMUM:     159790 FRA
       AVERAGE:     192414 FRA
```

Example 3 - DISPLAY Statement Using P* Notation

```
** Example 'DISEX3': DISPLAY (with P* notation)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 SALARY (1)
  2 BIRTH
  2 CITY
END-DEFINE
*
LIMIT 2
READ EMPL-VIEW BY CITY FROM 'N'
 DISPLAY NOTITLE NAME CITY
         VERT AS 'BIRTH/SALARY' BIRTH (EM=YYYY-MM-DD) SALARY (1)
  SKIP 1
  AT BREAK OF CITY
    DISPLAY P*SALARY (1) AVER(SALARY (1))
```

```
    SKIP 1
  END-BREAK
END-READ
END
```

Output of Program DISEX3:

```
        NAME                 CITY           BIRTH
                                            SALARY
-------------------- -------------------- ----------

WILCOX               NASHVILLE            1970-01-01
                                               38000

MORRISON             NASHVILLE            1949-07-10
                                               36000


                                               37000
```

Example 4 - DISPLAY Statement Using 'text ', 'c(n)' and Attribute Notation

```
** Example 'DISEX4': DISPLAY (with 'c(n)' notation and attribute)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 DEPT
  2 LEAVE-DUE
  2 NAME
END-DEFINE
*
LIMIT 4
READ EMPL-VIEW BY DEPT FROM 'T'
  IF LEAVE-DUE GT 40
   DISPLAY NOTITLE
           'EMPLOYEE' NAME                /* OVERRIDE STANDARD HEADER
           'LEAVE ACCUMULATED' LEAVE-DUE  /* OVERRIDE STANDARD HEADER
           '*'(10)(I)                     /* DISPLAY 10 '*' INTENSIFIED

  ELSE
    DISPLAY NAME LEAVE-DUE
  END-IF
END-READ
*
END
```

Output of Program DISEX4:

```
      EMPLOYEE       LEAVE ACCUMULATED
-------------------- -----------------


LAVENDA               33
BOYER                 33
CORREARD              45               **********
BOUVIER               19
```

Example 5 - DISPLAY Statement Using Horizontal Display

```
** Example 'DISEX5': DISPLAY (horizontal display)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 SALARY    (1:2)
  2 CURR-CODE (1:2)
END-DEFINE
*
LIMIT 4
READ EMPL-VIEW BY NAME
 DISPLAY NOTITLE NAME JOB-TITLE SALARY (1:2) CURR-CODE (1:2)
  SKIP 1
END-READ
*
END
```

Output of Program DISEX5:

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


ABELLAN              MAQUINISTA                   1450000 PTA
                                                  1392000 PTA

ACHIESON             DATA BASE ADMINISTRATOR        11300 UKL
                                                    10500 UKL

ADAM                 CHEF DE SERVICE               159980 FRA
                                                        0

ADKINSON             PROGRAMMER                     34500 USD
                                                    31700 USD
```

Example 6 - DISPLAY Statement Using Vertical and Horizontal Display

```
** Example 'DISEX6': DISPLAY (vertical and horizontal display)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 JOB-TITLE
  2 SALARY    (1:2)
  2 CURR-CODE (1:2)
END-DEFINE
*
LIMIT 1
READ EMPL-VIEW BY NAME
 DISPLAY NOTITLE VERT AS CAPTIONED
          NAME CITY 'POSITION' JOB-TITLE
          HORIZ 'SALARY' SALARY (1:2) 'CURRENCY' CURR-CODE (1:2)
  /*
  SKIP 1
END-READ
END
```

Output of Program DISEX6:

```
          NAME              SALARY   CURRENCY
          CITY
        POSITION
------------------------- ---------- --------

ABELLAN                      1450000 PTA
MADRID                       1392000 PTA
MAQUINISTA
```

Example 7 - DISPLAY Statement Using Parameters on Statement/Element Level

```
** Example 'DISEX7': DISPLAY (with parameters for statement/element)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 PERSONNEL-ID
  2 TELEPHONE
    3 AREA-CODE
    3 PHONE
END-DEFINE
*
LIMIT 3
READ EMPL-VIEW BY NAME
 DISPLAY NOTITLE (AL=16 GC=+ NL=8 SF=3 UC==)
```

```
          PERSONNEL-ID NAME TELEPHONE (LC=< TC=>)
END-READ
END
```

Output of Program DISEX7:

```
   PERSONNEL             NAME         +++++++++++++++TELEPHONE+++++++++++++++
       ID
                                             AREA             TELEPHONE
                                             CODE
================   ================   ==================   ==================

60008339           ABELLAN            <1      >            <4356726         >
30000231           ACHIESON           <0332   >            <523341          >
50005800           ADAM               <1033   >            <44864858        >
```

Example 8 - Report Specification with Output File Defined to Natural as PC

```
** Example 'PCDIEX1': DISPLAY and WRITE to PC
**
** NOTE: Example requires that Natural Connection is installed.
************************************************************************
DEFINE DATA LOCAL
01 PERS VIEW OF EMPLOYEES
  02 PERSONNEL-ID
  02 NAME
  02 CITY
END-DEFINE
*
FIND PERS WITH CITY = 'NEW YORK'               /* Data selection
  WRITE (7) TITLE LEFT 'List of employees in New York' /
  DISPLAY (7)          /* (7) designates the output file (here the PC).
    'Location'  CITY
    'Surname'   NAME
    'ID'        PERSONNEL-ID
END-FIND
END
```

62 DIVIDE

Related Statements: ADD | COMPRESS | COMPUTE | EXAMINE | MOVE | MOVE ALL | MULTIPLY | RESET | SEPARATE | SUBTRACT

Belongs to Function Group: ***Arithmetic and Data Movement Operations***

Function

The `DIVIDE` statement is used to divide an arithmetic expression or operand into two operands.

Note: Concerning Division by Zero: If an attempt is made to use a divisor (*operand1*) which is zero, either an error message or a result equal to zero will be returned; this depends on the setting of the session parameter `ZD` (described in the *Parameter Reference* documentation).

Syntax 1 - DIVIDE Statement without GIVING Clause

```
DIVIDE [ROUNDED] { (arithmetic-expression) } INTO operand2
                 { operand1                }
```

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

Operand Definition Table:

Operand	Possible Structure				Possible Formats	Referencing Permitted	Dynamic Definition
operand1	C	S	A	N	N P I F	yes	no
operand2		S	A	M	N P I F	yes	no

Syntax Element Description:

Syntax Element	Description
arithmetic-expression	See ***Arithmetic Expression*** in the `COMPUTE` statement.
operand1 INTO *operand2*	**Operands:** *operand1* is the divisor, *operand2* is the dividend. The result is stored in *operand2* (result field), hence the statement is equivalent to:

Syntax Element	Description
	`operand2 := operand2 / operand1` If an *arithmetic-expression* is used, *operand2* must not be an array range. The number of decimal positions for the result of the division is evaluated from the result field (that is, *operand2*). For the precision of the result, see *Rules for Arithmetic Assignments*, *Precision of Results of Arithmetic Operations* in the *Programming Guide*.
ROUNDED	**ROUNDED Option:** If you specify the keyword ROUNDED, the result will be rounded. For information on rounding, see *Rules for Arithmetic Assignment*, *Field Truncation and Field Rounding* in the *Programming Guide*.

Syntax 2 - DIVIDE Statement with GIVING Clause

```
DIVIDE     { (arithmetic-expression) } INTO { (arithmetic-expression) } GIVING
[ROUNDED]  { operand1                }      { operand2                } operand3
```

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

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A		N			N	P	I	F								yes	no
operand2	C	S	A		N			N	P	I	F								yes	no
operand3		S	A			A	U	N	P	I	F	B*							yes	yes

* Format B of *operand3* may be used only with a length of less than or equal to 4.

Syntax Element Description:

Syntax Element	Description
arithmetic-expression	See ***Arithmetic Expression*** in the COMPUTE statement.
operand1 INTO *operand2* GIVING *operand3*	**Operands:** *operand1* is the divisor, *operand2* is the dividend. The result of the division is stored in *operand3*, hence the statement is equivalent to:

Syntax Element	Description
	`operand3 := operand2 / operand1` The number of decimal positions for the result of the division is evaluated from the result field (that is, *operand3*). For the precision of the result, see *Rules for Arithmetic Assignments*, *Precision of Results of Arithmetic Operations* in the *Programming Guide*.
ROUNDED	**ROUNDED Option:** If you specify the keyword ROUNDED, the result will be rounded. For information on rounding, see *Rules for Arithmetic Assignment*, *Field Truncation and Field Rounding* in the *Programming Guide*.

Syntax 3 - DIVIDE Statement with REMAINDER Clause

```
DIVIDE { (arithmetic-expression) } INTO { (arithmetic-expression) } [GIVING operand3]
       { operand1               }      { operand2               } REMAINDER operand4
```

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

Operand Definition Table:

Operand	Possible Structure				Possible Formats	Referencing Permitted	Dynamic Definition
operand1	C	S	A	N	N P I	yes	no
operand2	C	S	A	N	N P I	yes	no
operand3		S	A		A U N P I F B* T	yes	yes
operand4		S	A		A U N P I F B* T	yes	yes

* Format B of *operand3* and *operand4* may be used only with a length of less than or equal to 4.

Syntax Element Description:

Syntax Element	Description
arithmetic-expression	See ***Arithmetic Expression*** in the COMPUTE statement.
operand1 *operand2*	**Operands:** *operand1* is the divisor, *operand2* is the dividend. If the GIVING clause is not used, the result is stored in *operand2*.

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td></td><td>If <code>operand2</code> is a constant or a non-modifiable Natural system variable, the <code>GIVING</code> clause is required.</td></tr>
<tr><td><code>GIVING</code> operand3</td><td>GIVING Clause:

If this clause is used, operand2 will not be modified and the result will be stored in operand3.

The number of decimal positions for the result of the division is evaluated from the result field (that is, operand2 if no <code>GIVING</code> clause is used, or operand3 if the <code>GIVING</code> clause is used).

For the precision of the result, see Rules for Arithmetic Assignments, Precision of Results of Arithmetic Operations (in the Programming Guide).</td></tr>
<tr><td><code>REMAINDER</code> operand4</td><td>REMAINDER Clause:

The remainder of the division is placed into the field specified in operand4.

■ If the <code>GIVING</code> clause is used, the statement is equivalent to:

<code>operand3 := operand2 / operand1</code>
<code>operand4 := operand2 - (operand3 * operand1)</code>

None of the four operands may be an array range.

■ If the <code>GIVING</code> clause is not used, the statement is equivalent to:

<code>temporary := operand2</code>
<code>operand2 := operand2 / operand1</code>
<code>operand4 := temporary - (operand2 * operand1)</code>

where temporary is a temporary field with the same format/length as operand2.

For each of these steps, the rules described in Precision of Results of Arithmetic Operations in the Programming Guide apply.</td></tr>
</table>

Example

```
** Example 'DIVEX1': DIVIDE
************************************************************************
DEFINE DATA LOCAL
1 #A (N7) INIT <20>
1 #B (N7)
1 #C (N3.2)
1 #D (N1)
1 #E (N1) INIT <3>
1 #F (N1)
```

```
END-DEFINE
*
DIVIDE 5 INTO #A
WRITE NOTITLE 'DIVIDE 5 INTO #A' 20X '=' #A
*
RESET INITIAL #A
DIVIDE 5 INTO #A GIVING #B
WRITE 'DIVIDE 5 INTO #A GIVING #B' 10X '=' #B
*
DIVIDE 3 INTO 3.10 GIVING #C
WRITE 'DIVIDE 3 INTO 3.10 GIVING #C' 8X '=' #C
*
DIVIDE 3 INTO 3.1 GIVING #D
WRITE 'DIVIDE 3 INTO 3.1 GIVING #D' 9X '=' #D
*
DIVIDE 2 INTO #E REMAINDER #F
WRITE 'DIVIDE 2 INTO #E REMAINDER #F' 7X '=' #E '=' #F
*
END
```

Output of Program DIVEX1:

```

DIVIDE 5 INTO #A                    #A:        4
DIVIDE 5 INTO #A GIVING #B          #B:        4
DIVIDE 3 INTO 3.10 GIVING #C        #C:    1.03
DIVIDE 3 INTO 3.1 GIVING #D         #D:  1
DIVIDE 2 INTO #E REMAINDER #F       #E:  1 #F:  1
```

63 DO/DOEND

```
DO statement ... DOEND
```

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

Belongs to Function Group: **Reporting Mode Statements**

Function

The `DO` and `DOEND` statements are used in reporting mode to specify a group of statements to be executed based on a logical condition as specified in any of the statements listed below.

- `AT BREAK`
- `AT END OF DATA`
- `AT END OF PAGE`
- `AT START OF DATA`
- `AT TOP OF PAGE`
- `BEFORE BREAK PROCESSING`
- `FIND ... IF NO RECORDS FOUND`
- `IF`
- `IF SELECTION`
- `ON ERROR`
- `READ WORK FILE ... AT END OF FILE`

Note: If you specify a only single statement to be executed based on a logical condition, you can omit the `DO` and `DOEND` statements. But with respect to good coding practice, you are not recommended to do so.

Restrictions

- The `DO` and `DOEND` statements are only valid in reporting mode.
- `WRITE TITLE`, `WRITE TRAILER`, and the `AT` condition statements `AT BREAK`, `AT END OF DATA`, `AT END OF PAGE`, `AT START OF DATA`, `AT TOP OF PAGE` are not permitted *within* a `DO`/`DOEND` statement group.
- A loop-initiating statement may be used within a `DO`/`DOEND` statement group provided that the loop is closed prior to the `DOEND` statement.

Example

```
** Example 'DOEEX1': DO/DOEND
************************************************************************
*
EMP. FIND EMPLOYEES WITH CITY = 'MILWAUKEE'
  VEH. FIND VEHICLES WITH PERSONNEL-ID = PERSONNEL-ID
    IF NO RECORDS FOUND DO
      ESCAPE
    DOEND
    DISPLAY PERSONNEL-ID (EMP.) NAME (EMP.)
            SALARY (EMP.,1)
            MAKE (VEH.) MAINT-COST (VEH.,1)
    AT END OF DATA DO
      WRITE NOTITLE
        / 10X 'AVG SALARY:'
               T*SALARY (1) AVER(SALARY (1))
        / 10X 'AVG MAINTENANCE (ZERO VALUES EXCLUDED):'
               T*MAINT-COST (1) NAVER(MAINT-COST (1))
    DOEND
    /*
  LOOP
LOOP
END
```

Output of Program DOEEX1:

```

PERSONNEL         NAME           ANNUAL           MAKE         MAINT-COST
   ID                            SALARY
--------- -------------------- ---------- -------------------- ----------


20021100  JONES                     31000 GENERAL MOTORS              140
20027800  LAWLER                    29000 GENERAL MOTORS                0
20027800  LAWLER                    29000 TOYOTA                       86
20030600  NORDYKE                   47000 FORD                        194

          AVG SALARY:               35666
          AVG MAINTENANCE (ZERO VALUES EXCLUDED):                     140
```

64 DOWNLOAD PC FILE

```
                                      ┌ [VARIABLE] operand1                 ┐
{ DOWNLOAD } { PC   } [FILE]          │                                     │
{ WRITE    } { WORK } work-file-number┤ COMMAND operand2 {   SYNC   }       │
                                      └                  {   ASYNC  }       ┘
```

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

Related Statements: `CLOSE PC FILE` | `UPLOAD PC FILE` | `WRITE WORK FILE`

Belongs to Function Group: ***Control of Work Files / PC Files***

Function

This statement is used to transfer data from a Linux platform to the PC.

See also the Natural Connection and Entire Connection documentation

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
operand1	C	S	A	G		A	U	N	P	I	F	B	D	T	L	C	yes	no
operand2	C	S				A											yes	yes

When using the work file types `ENTIRECONNECTION` or `TRANSFER`, *operand1* may neither be of Format C, nor G.

Syntax Element Description:

Syntax Element	Description
work-file-number	**Work File Number:** The work file number to be used. This number must correspond to one of the work file numbers for the PC as defined to Natural.
`VARIABLE`	**Variable Format:** The records in the PC file will be written in variable format. Note that variable records cannot be converted to PC spreadsheet formats.
operand1	**Field Specification:** With *operand1* you specify the fields to be downloaded to the PC.

Syntax Element	Description
`COMMAND`	**COMMAND Clause:** With the `COMMAND` clause, you can send PC commands (that is, any command that can be entered in the command line of Entire Connection on the PC) from the mainframe to the PC. Entire Connection checks whether the command sent is valid or not. A command that cannot be recognized by the PC is rejected. In this case, Natural issues the error message that the downloaded command was rejected by the PC. You can use the `COMMAND` clause, for example, to execute Entire Connection tasks from the mainframe. If you have the task `DIR` which lists PC directory information, you can initiate this directly out of your Natural program on the mainframe with the following statement: `DOWNLOAD PC FILE 7 COMMAND 'DIR'` In ***Example 2*** below, the `COMMAND` clause is used to define the name of the PC file that is to receive the downloaded data. In this way, you can avoid prompting for the name of the file.
operand2	**COMMAND Specification:** With *operand2*, you specify the DOS command or Entire Connection task that is to be executed on the PC. *operand2* must be an alphanumeric constant or variable.
`SYNC`	**SYNC Option:** With `SYNC`, you specify that the PC returns control to Natural after executing and terminating `COMMAND`. `SYNC` can be used, for example, to ensure that the command `SET PCFILE` has been executed before a file transfer starts.
`ASYNC`	**ASYNC Option:** With `ASYNC`, you specify that the PC immediately returns control to Natural, regardless of whether the execution of `COMMAND` has terminated or not.

Examples

- Example 2 - Use of COMMAND Clause

Example 1 - Use of DOWNLOAD PC FILE Statement

The following program demonstrates the use of the `DOWNLOAD PC FILE` statement. The data is first selected and then downloaded to the PC by using Work File 7.

```
** Example 'PCDOEX1': DOWNLOAD PC FILE
**
** NOTE: Example requires that Natural Connection is installed.
************************************************************************
DEFINE DATA LOCAL
01 PERS VIEW OF EMPLOYEES
  02 PERSONNEL-ID
  02 NAME
  02 CITY
END-DEFINE
*
FIND PERS WITH CITY = 'NEW YORK'                     /* Data selection
  DOWNLOAD PC FILE 7 CITY NAME PERSONNEL-ID    /* Data download
END-FIND
END
```

Output of Program PCDOEX1:

When you run the program, a window appears in which you specify the name of the PC file into which the data is to be downloaded. The data is then downloaded to the PC.

Example 2 - Use of COMMAND Clause

The following program demonstrates the use of the `COMMAND` clause in the `DOWNLOAD PC FILE` statement. The name of the receiving PC file is first defined. Then the data is selected and downloaded to this file.

```
** Example 'PCDOEX2': DOWNLOAD PC FILE
**
** NOTE: Example requires that Natural Connection is installed.
************************************************************************
DEFINE DATA LOCAL
01 PERS VIEW OF EMPLOYEES
  02 PERSONNEL-ID
  02 NAME
  02 CITY
01 CMD (A80)                               /* Variable for transfer
END-DEFINE                                 /* of the PC command
*
MOVE 'SET PCFILE 7 DOWN DATA PERS.NCD' TO CMD  /* PC command to define
*
DOWNLOAD PC FILE 6 COMMAND CMD             /* Command download
```

```
*
FIND PERS WITH CITY = 'NEW YORK'              /* Data selection
  DOWNLOAD PC FILE 7 CITY NAME PERSONNEL-ID    /* Data download
END-FIND
END
```

Note: The PC file number in two successive `DOWNLOAD PC FILE` statements must be different.

Output of Program PCDOEX2:

When you run the program, the data is downloaded to the PC file that was specified in the program. A window does not appear.

65 EJECT

Related Statements: AT END OF PAGE | AT TOP OF PAGE | CLOSE PRINTER | DEFINE PRINTER | DISPLAY | FORMAT | NEWPAGE | PRINT | SKIP | SUSPEND IDENTICAL SUPPRESS | WRITE | WRITE TITLE | WRITE TRAILER

Belongs to Function Group: *Creation of Output Reports*

Function

The `EJECT` statement may be used to control page advance/page ejection.

Syntax Description

Two different structures are possible for this statement.

- EJECT - Syntax 1
- EJECT - Syntax 2

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

EJECT - Syntax 1

```
EJECT { ON  } [(rep)]
      { OFF }
```

Syntax Element Description:

<table>
<tr><th>Syntax Element</th><th colspan="2">Description</th></tr>
<tr><td rowspan="3">EJECT ON/OFF (rep)</td><td colspan="2">With Report Specification - Online and Batch Modes:</td></tr>
<tr><td>EJECT OFF (rep)</td><td>Causes no page advance (except as specified with Syntax 2 of the EJECT statement) for the specified report to be executed.</td></tr>
<tr><td>EJECT ON (rep)</td><td>Causes page advances for the specified report to be executed.</td></tr>
<tr><td>EJECT ON/OFF</td><td colspan="2">Without Report Specification - Batch Mode only:

Without report notation (rep), EJECT ON/OFF may be used in batch mode to control page ejection between the output listings created during the execution of a program.</td></tr>
</table>

Syntax Element	Description	
	`EJECT ON`	Causes Natural to generate a page eject between the source program listing, the output report and the message `EXECUTION COMPLETED` . This is the default setting.
	`EJECT OFF`	Causes Natural to suppress page breaks between the above output. `EJECT OFF` remains in effect until revoked with a subsequent `EJECT ON` statement.
`(rep)`	**Report Specification:** The notation `(rep)` may be used to specify the identification of the report for which the `EJECT` statement is applicable. A value in the range 0 - 31 or a logical name which has been assigned using the `DEFINE PRINTER` statement may be specified. If `(rep)` is not specified, the `EJECT` statement will be applicable to the first report (Report 0). For information on how to control the format of an output report created with Natural, see *Report Format and Control* in the *Programming Guide*.	

EJECT - Syntax 2

This form of the `EJECT` statement may be used to cause a page advance without a title or heading line being generated on the next page and without `TOP/END PAGE` processing.

```
EJECT  [(rep)] [ [ IF   ] LESS [THAN] operand1  [LINES] [LEFT] ]
                 [ WHEN ]
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
operand1	C	S					N	P	I									yes	no

Syntax Element Description:

Syntax Element	Description
(*rep*)	**Report Specification:** The notation (*rep*) may be used to specify the identification of the report for which the EJECT statement is applicable. A value in the range 0 - 31 or a logical name which has been assigned using the DEFINE PRINTER statement may be specified. If (*rep*) is not specified, the EJECT statement will be applicable to the first report (Report 0). For information on how to control the format of an output report created with Natural, see *Report Format and Control* in the *Programming Guide*.
IF LESS THAN *operand1* LINES LEFT	**IF LESS THAN ... LINES LEFT Clause:** A page advance will be performed only when the current line for the page is greater than the page size minus *operand1*. The value for *operand1* may be specified as a numeric constant or as a variable.

Processing

The execution of an EJECT statement does not cause any statements used with an AT TOP OF PAGE, AT END OF PAGE, WRITE TITLE or WRITE TRAILER statement to be executed. It does not affect system functions evaluated by DISPLAY GIVE SYSTEM FUNCTIONS.

EJECT causes a new physical page only. It causes the Natural system variable *LINE-COUNT to be set to 1 but has no effect on the setting of the Natural system variable *PAGE-NUMBER.

Example

```
** Example 'EJTEX1': EJECT
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
END-DEFINE
*
FORMAT PS=15
LIMIT 9
READ EMPLOY-VIEW BY CITY
  /*
  AT START OF DATA
```

```
    EJECT
    WRITE /// 20T '%' (29) /
              20T '%%'                         47T '%%' /
              20T '%%' 3X 'REPORT OF EMPLOYEES' 47T '%%' /
              20T '%%' 3X '  SORTED BY CITY   ' 47T '%%' /
              20T '%%'                         47T '%%' /
              20T '%' (29) /
    EJECT
  END-START
  EJECT WHEN LESS THAN 3 LINES LEFT
  /*
  WRITE '*' (64)
  DISPLAY NOTITLE NOHDR CITY NAME JOB-TITLE 5X *LINE-COUNT
  WRITE '*' (64)
END-READ
END
```

Output of Program EJTEX1:

```



                     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                     %%                         %%
                     %%   REPORT OF EMPLOYEES   %%
                     %%     SORTED BY CITY      %%
                     %%                         %%
                     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

After pressing ENTER:

```
****************************************************************
AIKEN                SENKO                PROGRAMMER                         2
****************************************************************
****************************************************************
AIX EN OTHE          GODEFROY             COMPTABLE                          5
****************************************************************
****************************************************************
AJACCIO              CANALE               CONSULTANT                         8
****************************************************************
****************************************************************
ALBERTSLUND          PLOUG                KONTORASSISTENT                   11
****************************************************************
****************************************************************
ALBUQUERQUE          HAMMOND              SECRETARY                         14
****************************************************************
```

After pressing ENTER:

```
***************************************************************
ALBUQUERQUE          ROLLING              MANAGER                            2
***************************************************************
***************************************************************
ALBUQUERQUE          FREEMAN              MANAGER                            5
***************************************************************
***************************************************************
ALBUQUERQUE          LINCOLN              ANALYST                            8
***************************************************************
***************************************************************
ALFRETON             GOLDBERG             JUNIOR                            11
***************************************************************
```

66 END

```
{ END }
{  .  }
```

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

Function

The END statement is used to mark the physical end of a Natural program. No symbols may follow the END statement.

In reporting mode, any processing loop which is currently active (that is, which has not been closed with a LOOP statement) is closed by the END statement.

Considerations for Program Execution

When an END statement is executed in a main program (that is, a program executing on Level 1), final end-page processing is performed as well as final break processing for user-initiated breaks (PERFORM BREAK PROCESSING) which have not been associated with a processing loop by specifying a reference notation (*r*).

When an END statement is executed in a subprogram, or in a program invoked with FETCH RETURN, control will be returned to the invoking program without any final processing.

Syntax Description

Syntax Element	Description
END	**Keyword:** The Natural reserved keyword END is normally used to mark the physical end of a Natural program.
.	**Period:** Instead of the Natural reserved keyword END, a period (.) may be used. It must be preceded by at least one blank if other statements are contained in the same line.

Examples

For some typical examples, see ***Examples of DEFINE DATA Statement Usage***.

67 END TRANSACTION

```
END [OF] TRANSACTION [operand1 ...]
```

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

Related Statements: ACCEPT/REJECT | AT BREAK | AT START OF DATA | AT END OF DATA | BACKOUT TRANSACTION | BEFORE BREAK PROCESSING | DELETE | FIND | GET | GET SAME | GET TRANSACTION DATA | FIND HISTOGRAM | LIMIT | PASSW | PERFORM BREAK PROCESSING | READ | RETRY | STORE | UPDATE

Belongs to Function Group: ***Database Access and Update***

Function

The `END TRANSACTION` statement is used to indicate the end of a logical transaction. A logical transaction is the smallest logical unit of work (as defined by the user) which must be performed in its entirety to ensure that the information contained in the database is logically consistent.

Successful execution of an `END TRANSACTION` statement ensures that all updates performed during the transaction have been or will be physically applied to the database regardless of subsequent user, Natural, database or operating system interruption. Updates performed within a transaction for which the `END TRANSACTION` statement has not been successfully completed will be backed out automatically.

The `END TRANSACTION` statement also results in the release of all records placed in hold status during the transaction.

The `END TRANSACTION` statement can be executed based upon a logical condition.

For further information, see the section *Database Update - Transaction Processing* (in the *Programming Guide*).

Restriction

This statement cannot be used with Entire System Server.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S			N	A	U	N	P	I	F	B	D	T					yes	no

Syntax Element Description:

Syntax Element	Description
operand1	**Storage of Transaction Data:** For a transaction applied to an Adabas database, you may also use this statement to store transaction-related information. These transaction data must not exceed 2000 bytes. They may be read with a `GET TRANSACTION DATA` statement. The transaction data are written to the database specified with the profile parameter `ETDB`. If the `ETDB` parameter is not specified, the transaction data are written to the database specified with the profile parameter `UDB`, except on mainframe computers: here, they are written to the database where the Natural Security system file (`FSEC`) is located (if `FSEC` is not specified, it is considered to be identical to the Natural system file, `FNAT`; if Natural Security is not installed, the transaction data are written to the database where `FNAT` is located). **Note:** `END TRANSACTION` cannot be used if *operand1* is a dynamic variable.

Databases Affected

An `END TRANSACTION` statement *without* transaction data (that is, without *operand1*) will only be executed if a database transaction under control of Natural has taken place. Depending on the setting of the Natural profile parameter `ET`, the statement will be executed only for the database affected by the transaction (`ET=OFF`), or for all databases that have been referenced since the last execution of a `BACKOUT TRANSACTION` or `END TRANSACTION` statement (`ET=ON`).

An `END TRANSACTION` statement *with* transaction data (that is, with specifying *operand1*) will always be executed and the transaction data be stored in a database as described in the following section. It depends on the setting of the `ET` parameter (see above) for which other databases the `END TRANSACTION` statement will be executed.

Database-Specific Considerations

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

Examples

- Example 1 - END TRANSACTION
- Example 2 - END TRANSACTION with ET Data

Example 1 - END TRANSACTION

```
** Example 'ETREX1': END TRANSACTION
**
** CAUTION: Executing this example will modify the database records!
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 COUNTRY
END-DEFINE
*
FIND EMPLOY-VIEW WITH CITY = 'BOSTON'
  ASSIGN COUNTRY = 'USA'
  UPDATE
  END TRANSACTION
  /*
  AT END OF DATA
    WRITE NOTITLE *NUMBER 'RECORDS UPDATED'
  END-ENDDATA
  /*
END-FIND
END
```

Output of Program ETREX1:

```
7 RECORDS UPDATED
```

Example 2 - END TRANSACTION with ET Data

```
** Example 'ETREX2': END TRANSACTION (with ET data)
**
** CAUTION: Executing this example will modify the database records!
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
  2 CITY
*
1 #PERS-NR (A8) INIT <' '>
END-DEFINE
*
REPEAT
  INPUT 'ENTER PERSONNEL NUMBER TO BE UPDATED:' #PERS-NR
  IF #PERS-NR = ' '
    ESCAPE BOTTOM
  END-IF
  /*
  FIND EMPLOY-VIEW PERSONNEL-ID = #PERS-NR
    INPUT (AD=M)    NAME / FIRST-NAME / CITY
    UPDATE
    END TRANSACTION #PERS-NR
  END-FIND
  /*
END-REPEAT
END
```

Output of Program ETREX2:

```
ENTER PERSONNEL NUMBER TO BE UPDATED: 20027800
```

After entering and confirming the personnel number:

```

NAME LAWLER
FIRST-NAME SUNNY
CITY MILWAUKEE
```

68 ESCAPE

Structured Mode Syntax

```
ESCAPE  { TOP [REPOSITION]                }
        { BOTTOM [(r)] [IMMEDIATE]        }
        { ROUTINE [IMMEDIATE]             }
        { MODULE [IMMEDIATE]              }
```

Reporting Mode Syntax

```
ESCAPE  [ TOP [REPOSITION]                ]
        [ BOTTOM [(r)] [IMMEDIATE]        ]
        [ ROUTINE [IMMEDIATE]             ]
        [ MODULE [IMMEDIATE]              ]
```

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

Related Statements:

- FIND | FOR | HISTOGRAM | PARSE XML | READ | READ RESULT SET (SQL) | READ WORK FILE | READLOB | REPEAT | SORT
- CALL | CALL FILE | CALL LOOP | CALLNAT | DEFINE SUBROUTINE | FETCH | PERFORM

Belongs to Function Group:

- *Loop Execution*
- *Invoking Programs and Routines*

Function

The ESCAPE statement is used to interrupt the linear flow of execution of a processing loop or a routine.

With the keywords TOP, BOTTOM and ROUTINE you indicate where processing is to continue when the ESCAPE statement is encountered.

An ESCAPE TOP/BOTTOM statement, when encountered for processing, will internally refer to the innermost active processing loop. The ESCAPE statement need not be physically placed within the processing loop.

If an ESCAPE TOP/BOTTOM statement is placed in a routine (subroutine, subprogram, function, or a program invoked with FETCH RETURN), the routine(s) entered during execution of the processing loop will be terminated automatically.

Additional Considerations

More than one ESCAPE statement may be contained within the same processing loop.

The execution of an ESCAPE statement may be based on a logical condition. If an ESCAPE statement is encountered during processing of an AT END OF DATA, AT BREAK or AT END OF PAGE block, the execution of the special condition block will be terminated and ESCAPE processing will continue as required.

If an ESCAPE statement is encountered during processing of an if-no-records-found condition, no loop-end processing will be performed (equivalent to ESCAPE IMMEDIATE).

Syntax Description

Syntax Element	Description
ESCAPE TOP	**Top Option:** TOP indicates that processing is to continue at the top of the processing loop. This starts the next repetition of the processing loop.
REPOSITION	**Top Reposition Option:** When an ESCAPE TOP REPOSITION statement is executed, Natural immediately continues processing at the top of the active READ loop, using the current value of the search variable as new start value. At the same time, ESCAPE TOP REPOSITION resets the system variable *COUNTER to zero. ESCAPE TOP REPOSITION can be specified within a READ statement loop accessing an Adabas database. The READ statement concerned must contain the option WITH REPOSITION.
ESCAPE BOTTOM	**Bottom Option:** BOTTOM indicates that processing is to continue with the first statement following the processing loop. The loop is terminated and loop-end processing (final BREAK and END DATA) is executed for all loops being terminated. In reporting mode, ESCAPE BOTTOM is the default.
(*r*)	**Statement Reference:** Notation (*r*): If BOTTOM is followed by a label or reference number, processing will continue with the first statement following the processing loop identified by the label or reference number. A label or a reference number can only be specified if the ESCAPE BOTTOM statement is physically placed within the referenced processing loop.

Syntax Element	Description
IMMEDIATE	**Immediate Option:** If you specify the keyword IMMEDIATE, no loop-end processing will be performed.
ESCAPE ROUTINE	**Routine Option:** This option indicates that the current Natural routine, which may have been invoked via a PERFORM, CALLNAT, FETCH RETURN, or as a main program, is to relinquish control. In the case of a subroutine, processing will continue with the first statement after the statement used to invoke the subroutine. In the case of a main program, Natural command mode will be entered. All loops currently active within the routine will be terminated and loop-end processing performed as well as final processing for user-initiated (PERFORM BREAK) processing. If the program containing the ESCAPE ROUTINE is executed as a main program (Level 1), final end-page processing is performed.
ESCAPE MODULE	**Module Option:** This option indicates that the entire current program level, with all internal subroutines, is to relinquish control. The control is then returned to the object of the former program level. If ESCAPE MODULE is used in a hierarchy of internal subroutines, it allows to escape all routines working at this level at once. If no internal subroutine is active, ESCAPE MODULE has the same result as ESCAPE ROUTINE. ESCAPE MODULE is only relevant in inline subroutines. In external subroutines, subprograms and invoked programs, it has the same effect as ESCAPE ROUTINE. As with ESCAPE ROUTINE, loop-end processing will be performed. However, if you specify the keyword IMMEDIATE, no loop-end processing will be performed.

Example

```
** Example 'ESCEX1': ESCAPE
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 FIRST-NAME
  2 NAME
  2 AREA-CODE
  2 PHONE
*
1 #CITY (A20) INIT <' '>
1 #CNTL (A1)  INIT <' '>
END-DEFINE
*
REPEAT
```

```
  INPUT 'ENTER VALUE FOR CITY:  ' #CITY
      / 'OR ''.'' TO TERMINATE  '
  IF #CITY = '.'
    ESCAPE BOTTOM
  END-IF
  /*
  FND. FIND EMPLOY-VIEW WITH CITY = #CITY
    /*
    IF NO RECORDS FOUND
      WRITE 'NO RECORDS FOUND'
      ESCAPE BOTTOM (FND.)
    END-NOREC
    AT START OF DATA
      INPUT (AD=O) 'RECORDS FOUND:' *NUMBER //
                   'ENTER ''D'' TO DISPLAY RECORDS' #CNTL (AD=M)
      IF #CNTL NE 'D'
        ESCAPE BOTTOM (FND.)
      END-IF
    END-START
    /*
    DISPLAY NOTITLE NAME FIRST-NAME PHONE
  END-FIND
END-REPEAT
```

Output of Program ESCEX1:

```

ENTER VALUE FOR CITY:  PARIS
(OR '.' TO TERMINATE)
```

After entering and confirming city name:

```

RECORDS FOUND:        26
ENTER 'D' TO DISPLAY RECORDS D
```

Result after entering and confirming D:

```

        NAME              FIRST-NAME         TELEPHONE
-------------------- -------------------- ---------------


MAIZIERE             ELISABETH            46758304
MARX                 JEAN-MARIE           40738871
REIGNARD             JACQUELINE           48472153
RENAUD               MICHEL               46055008
REMOUE               GERMAINE             36929371
LAVENDA              SALOMON              40155905
BROUSSE              GUY                  37502323
GIORDA               LOUIS                37497316
SIECA                FRANCOIS             40487413
```

```
CENSIER             BERNARD             38070268
DUC                 JEAN-PAUL           38065261
CAHN                RAYMOND             43723961
MAZUY               ROBERT              44286899
FAURIE              HENRI               44341159
VALLY               ALAIN               47326249
BRETON              JEAN-MARIE          48467146
GIGLEUX             JACQUES             40477399
KORAB-BRZOZOWSKI    BOGDAN              45288048
XOLIN               CHRISTIAN           46060015
LEGRIS              ROGER               39341509
VVVV
```

69 EXAMINE

Related Statements: ADD | COMPRESS | COMPUTE | DIVIDE | MOVE | MOVE ALL | MULTIPLY | RESET | SEPARATE | SUBTRACT

Belongs to Function Group: *Arithmetic and Data Movement Operations*

Syntax 1 - EXAMINE

```
EXAMINE [DIRECTION-clause]
                                         operand1
        [FULL [VALUE [OF]]]  {           SUBSTRING                                  }
                                         (operand1,operand2,operand3)
        [POSITION-clause]
        [FOR] [FULL [VALUE [OF]]] [PATTERN] operand4
        [DELIMITERS-option]
        DELETE-REPLACE-clause
{       GIVING-clause                                                               }
        DELETE-REPLACE-clause GIVING-clause
```

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

Syntax Description - Syntax 1

The EXAMINE statement is used to inspect the content of an alphanumeric or binary field, or a range of fields within an array, and to

- return the number of how many times a search-pattern was found;
- return the byte position where a search-pattern appears first;
- return the significant content length of a field; that is, the field length without trailing blanks;
- return the occurrence number (indices) of an array field, where a pattern was found first;
- replace a pattern by another pattern;
- delete a pattern.

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C*	S	A			A	U					B							yes	no
operand2	C	S						N	P	I		B*							yes	no
operand3	C	S						N	P	I		B*							yes	no
operand4	C	S	A*			A	U					B							yes	no

* *operand1* can only be a constant if the GIVING clause is used, but not if the DELETE/REPLACE clause is used.

* *operand4* can also be used as an array, see ***Search and Replace with Multiple Values***.

* Format B of *operand2* and *operand3* may be used only with a length of less than or equal to 4.

Syntax Element Description:

Syntax Element	Description
DIRECTION-clause	**DIRECTION Clause:** This clause determines the search direction. For details, see ***DIRECTION Clause*** below.
operand1	**Field to be Examined:** *operand1* is the field whose content is to be examined. If *operand1* is a DYNAMIC variable, a REPLACE operation may cause its length to be increased or decreased; a DELETE operation may cause its length to be set to zero. The current length of a DYNAMIC variable can be ascertained by using the system variable *LENGTH.
POSITION-clause	**POSITION Clause:** This clause may be used to specify a starting and ending position within *operand1* (or the substring of *operand1*) for the examination. For details, see ***POSITION Clause*** below.
operand4	**Value to be Used for EXAMINE Operation:** *operand4* is the value which is searched for in the examined field(s). You may search for a single value or for multiple values. For more information on *operand4* and *operand6*, see *operand6*, which is used in the ***DELETE REPLACE Clause*** described below.
FULL	**FULL Option:** If FULL is specified for an operand, the entire value, including trailing blanks, will be processed. If FULL is not specified, trailing blanks in the operand will be ignored.
SUBSTRING	**SUBSTRING Option:**

Syntax Element	Description
	Normally, the content of a field is examined from the beginning of the field to the end or to the last non-blank character. With the `SUBSTRING` option, you examine only a certain part of the field. After the field name (*operand1*) in the `SUBSTRING` clause, you specify first the starting position (*operand2*) and then the length (*operand3*) of the field portion to be examined. For example, to examine the 5th to 12th position inclusive of a field #A, you would specify: `EXAMINE SUBSTRING(#A,5,8).` **Note:** 1. If you omit *operand2*, the starting position is assumed to be `1`. 2. If you omit *operand3*, the length is assumed to be from the starting position to the end of the field. 3. If `SUBSTRING` is used in conjunction with a `DYNAMIC` variable, the field behaves like a fixed length variable; that is, the length (`*LENGTH`) does not change as a result of the `EXAMINE` operation, regardless of whether a `DELETE` or `REPLACE` operation was executed or not.
`PATTERN`	**PATTERN Option:** If you wish to examine the field for a value which contains "wild characters", that is symbols for positions not to be examined, you use the `PATTERN` option. *operand4* may then include the following symbols for positions to be ignored: - A period (.), question mark (?) or underscore (_) indicates a single position that is not to be examined. - An asterisk (*) or a percent sign (%) indicates any number of positions not to be examined. Example: With `PATTERN 'NAT*AL'` you could examine the field for any value which contains `NAT` and `AL` no matter which and how many other characters are between `NAT` and `AL` (this would include the values `NATURAL` and `NATIONAL` as well as `NATAL`). **Note:** If you use a pattern that starts with an asterisk (*) or percent sign (%), the following rule applies: - All positions from the previous delimiter are not examined. If there is no delimiter, all positions from the beginning of the string are not examined. If you use a pattern that ends with an asterisk (*) or percent sign (%), the following rule applies:

Syntax Element	Description
	■ All positions to the next delimiter are not examined. If there is no delimiter, all positions to the end of the string are not examined.
DELIMITERS-option	**DELIMITERS Option:** This option is used to scan for a value which exhibits delimiters. For details, see ***DELIMITERS Option*** below.
DELETE-REPLACE-clause	**DELETE REPLACE Clause:** The `DELETE` option of this clause is used to delete each search-value (*operand4*) found in *operand1*, whereas the `REPLACE` option is used to replace each search-value (*operand4*) found in *operand1* by the value specified in *operand6*. For details, see ***DELETE REPLACE Clause*** below.
GIVING-clause	For details, see ***GIVING Clause*** below.

DIRECTION Clause

The direction clause determines the search direction.

```
DIRECTION { FORWARD  }
          { BACKWARD }
          { operand8 }
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
operand8	C	S				A1												yes	no

Syntax Element Description:

Syntax Element	Description
`FORWARD`	**Examine in Left-to-Right Direction:** If you specify `FORWARD`, the contents of the field are examined from left to right.
`BACKWARD`	**Examine in Right-to-Left Direction:** If you specify `BACKWARD`, the contents of the field are examined from right to left.
operand8	**Alternative Specification:** If you specify *operand8*, the search direction is determined by the contents of *operand8*. *operand8* must be defined with format/length A1. If *operand8* contains an `F`, then the search direction is `FORWARD`, if *operand8* contains a `B`, the search direction is `BACKWARD`. All other values are invalid and are rejected at compile time if *operand8* is a constant, or at run time if *operand8* is a variable.

Note: If the DIRECTION clause is not specified, the default direction is FORWARD.

POSITION Clause

The POSITION clause may be used to specify a starting and ending position within *operand1* (or the substring of *operand1*) for the examination.

```
[[STARTING] FROM [POSITION] operand9] [ { ENDING AT } [POSITION] operand10 ]
                                        { THRU      }
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
operand9	C	S					N	P	I									yes	no
operand10	C	S					N	P	I									yes	no

Syntax Element Description:

Syntax Element	Description
FROM *operand9*	**Starting Position:** *operand9* is used to define the starting position for the examination.
ENDING AT / THRU *operand10*	**Ending Position:** *operand10* is used to define the ending position for the examination.

The starting position (*operand9*) and the ending position (*operand10*) are relative to *operand1* or the substring of *operand1*, and both are processed.

The search is performed starting from the starting position and ending at the ending position.

If the starting and/or ending position are not specified, the default position value applies. This value is determined by the search direction:

Direction	Default Starting Position	Default Ending Position
FORWARD	1 (first character)	length of *operand1* (last character)
BACKWARD	length of *operand1* (last character)	1 (first character)

Note: If the search direction is FORWARD and the start position is greater than the end position, or if the search direction is BACKWARD and the start position is less than the end position, no search is performed.

DELIMITERS Option

```
{ ABSOLUTE                               }
{ WITH [DELIMITERS] [operand5]           }
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand5	C	S				A	U					B							yes	no

Syntax Element Description:

Syntax Element	Description
`ABSOLUTE`	**Absolute Scan Option:** This is the default option. It results in an absolute scan of the field for the specified value regardless of what other characters may surround the value.
`WITH DELIMITERS`	**WITH DELIMITERS Option:** This option is used to scan for a value which is delimited by blanks or by any character that is neither a letter nor a numeric character.
`WITH DELIMITERS` *operand5*	**Specific Delimiter Option:** This option is used to scan for a value which is delimited by the character or any of the characters specified in *operand5*. If the search value was found at the beginning or end of the examined field, only the right or left side has to be delimited by one of the *operand5* characters.

DELETE/REPLACE Clause

```
      { DELETE [FIRST]                                         }
[AND] { REPLACE [FIRST] [WITH] [FULL [VALUE [OF]]] operand6    }
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand6	C	S	A*			A	U					B							yes	no

* *operand6* can also be used as an array, see ***Search and Replace with Multiple Values***.

Syntax Element Description:

Syntax Element	Description
`DELETE`	**DELETE Option:** This option is used to delete the first (or all) occurrence(s) of the search-value (*operand4*) in the content of *operand1*.
`REPLACE`	**REPLACE Option:** This option is used to replace the first (or all) occurrence(s) of the search-value (*operand4*) in *operand1* by the replace value specified in *operand6*.
`FIRST`	**FIRST Option:** If you specify the keyword `FIRST`, only the first identical value will be deleted/replaced.

Notes:

1. If the `REPLACE` operation results in more characters being generated than will fit into *operand1*, you will receive an error message.
2. If *operand1* is a dynamic variable, a `REPLACE` operation may cause its length to be increased or decreased; a `DELETE` operation may cause its length to be set to zero. The current length of a dynamic variable can be ascertained by using the system variable `*LENGTH`. For general information on dynamic variables, see *Using Dynamic Variables*.

Search and Replace with Multiple Values

The search (*operand4*) and replace value (*operand6*) may also be defined as array fields. This allows to substitute multiple different patterns in the examined field (*operand1*), all with an unique `EXAMINE` statement. It is not necessary to have the same number of occurrences for the search and replace operand. All what is required is the transfer compatibility between these fields; that is, `operand4:=operand6` must be a valid operation; see *Assignment Operations with Arrays* in the *Programming Guide*.

The operation logic for the multi-value search is as follows:

- The field to be examined (*operand1*) is passed through only a single time, either from left to right for direction `FORWARD` or from right to left for direction `BACKWARD`.
- Beginning with the first position, the values in the search array (*operand4*) are tested for a match, one after the other, starting with the array occurrence with the lowest index.
- If no search value was found, the comparison repeats on the next field position.
- If one of the searched patterns is detected in the examined field (*operand1*), it is substituted with the value of the replace array (*operand6*), which overlays the matching pattern in *operand4*, if a `operand4:=operand6` would be executed.
- After a pattern replacement was performed, the compare process continues with the first occurrence for the search array, immediately after the inserted value. This means, a replaced pattern is skipped and may not be replaced a second time.

Example:

This example shows an HTML translation for the characters less than (<), greater than (>), and ampersand (&).

```
DEFINE DATA LOCAL
1 #HTML  (A/1:3) DYNAMIC INIT <'&lt;','&gt;','&amp;'>
1 #TAB   (A/1:3) DYNAMIC INIT <'<','>','&'>
1 #DOC(A) DYNAMIC  /* document to be replaced
END-DEFINE
#DOC := 'a&lt;&lt;b&amp;b&gt;c&gt;'
WRITE #DOC (AL=30) 'before'
/* Replace #DOC using #HTML to #TAB (n:1 replacement)
EXAMINE   #DOC FOR  #HTML(*)  REPLACE  #TAB(*)
/* '&lt;'  is replaced by '<'  (4:1 replacement)
/* '&gt;'  is replaced by '>'  (4:1 replacement)
/* '&amp;' is replaced by '&'  (5:1 replacement)
WRITE #DOC (AL=30) 'after'
END
```

See also *Example 3 - EXAMINE AND REPLACE WITH MULTIPLE VALUES*.

GIVING Clause

```
[ { GIVING [IN] operand7          } ]
  { [GIVING] NUMBER [IN] operand7 }
[[GIVING] POSITION [IN] operand7]
[[GIVING] LENGTH [IN] operand7]
[[GIVING] INDEX [IN] operand7 ...3]
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand7		S					N	P	I										yes	yes

Syntax Element Description:

Syntax Element	Description
`GIVING`	**GIVING Clause:** If only the keyword `GIVING` is specified, this corresponds to `GIVING NUMBER` (default).
`NUMBER`	**GIVING NUMBER Clause:** Is used to obtain information on how many times the search value (*operand4*) was found in the field (*operand1*) whose content is to be examined.

Syntax Element	Description
`POSITION`	**GIVING POSITION Clause:** Is used to obtain the byte position within *operand1* (or the substring of *operand1*) where the first value identical to *operand4* was found.
`LENGTH`	**GIVING LENGTH Clause:** Is used to obtain the remaining content length of *operand1* (or the substring of *operand1*) after all delete/replace operations have been performed. Trailing blanks are ignored.
operand7	**Number of Occurrences:** The number of occurrences of the search-value. If the `REPLACE FIRST` or `DELETE FIRST` option is also used, the number will not exceed 1.
`INDEX` *operand7* `...3`	**GIVING INDEX Clause:** This option is only applicable if the underlying field to be examined is an array field. `GIVING INDEX` is used to obtain the array occurrence number (index) of *operand1* in which the first search-value (*operand4*) was found. *operand7* must be specified as many times as there are dimensions in *operand1* (maximum three times). *operand7* will return `0` if the search-value is found in none of the occurrences. **Note:** If the index range of *operand1* includes the occurrence `0` (for example, `0:5`), a value of `0` in *operand7* is ambiguous. In this case, an additional `GIVING NUMBER` clause should be used to ascertain whether the search-value was actually found or not.

Syntax 2 - EXAMINE TRANSLATE

```
EXAMINE { operand1                                               } [AND]
        { SUBSTRING   (operand1,operand2,operand3)               }

          TRANSLATE  {  INTO { UPPER }            [CASE] }
                     {       { LOWER }                   }
                     {  USING [INVERTED] operand4        }
```

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

Syntax Description - Syntax 2

The `EXAMINE TRANSLATE` statement is used to translate the characters contained in a field into upper-case or lower-case, or into other characters.

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1		S	A			A	U					B							yes	no
operand2	C	S						N	P	I		B*							yes	no
operand3	C	S						N	P	I		B*							yes	no
operand4		S	A			A	U					B							yes	no

*Format B of *operand2* and *operand3* may be used only with a length of less than or equal to 4.

Syntax Element Description:

Syntax Element	Description
`EXAMINE` *operand1*	**Complete Field Content Translation:** *operand1* is the field whose content is to be translated.
`EXAMINE SUBSTRING` *operand1* *operand2* *operand3*	**Partial Field Content Translation:** Normally, the entire content of a field is translated. With the `SUBSTRING` option, you translate only a certain part of the field. After the field name (*operand1*) in the `SUBSTRING` clause, you specify first the starting position (*operand2*) and then the length (*operand3*) of the field portion to be examined. For example, to translate the 5th to 12th position inclusive of a field `#A`, you would specify: `EXAMINE SUBSTRING(#A,5,8) AND TRANSLATE ...` **Note:** If you omit *operand2*, the starting position is assumed to be `1`. If you omit *operand3*, the length is assumed to be from the starting position to the end of the field.
`TRANSLATE INTO UPPER CASE`	**Upper Case Translation:** The content of *operand1* will be translated into upper case.
`TRANSLATE INTO LOWER CASE`	**Lower Case Translation:** The content of *operand1* will be translated into lower case.
`TRANSLATE USING` *operand4*	**Translation Table:**

Syntax Element	Description
	`operand4` is the translation table to be used for character translation. The table must be of format/length A2, U2 or B2. **Note:** If for a character to be translated more than one translation is defined in the translation table, the last of these translations applies.
`INVERTED`	**INVERTED Option:** If you specify the keyword `INVERTED`, the translation table (`operand4`) will be used inverted; that is, the translation direction will be reversed.

Syntax 3 - EXAMINE for Unicode Graphemes

```
EXAMINE [FULL     {  operand1                                      }
[VALUE [OF]]]     {  SUBSTRING (operand1,operand2,operand3)        }
[POSITION-clause]
                  {  CHARPOSITION operand4 CHARLENGTH operand5     }
[FOR]             {  CHARPOSITION operand4                         }
                  {  CHARLENGTH operand5                           }
{  [GIVING] POSITION [IN] operand6 [GIVING] LENGTH [IN] operand7  }
{  [GIVING] POSITION [IN] operand6                                }
{  [GIVING] LENGTH [IN] operand7                                  }
```

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

Syntax Description - Syntax 3

A “grapheme” is what a user normally thinks of as a character. In most cases, a UTF-16 code unit (= U format character) is a grapheme, however, a grapheme can also consist of several code units. Examples are: a sequence of a base character followed by combining characters or a surrogate pair. For more information on graphemes and other Unicode terms, see *The Unicode Standard* at ***http://www.unicode.org/***.

The `EXAMINE` statement for U format operands in general operates on code units. However, with the `CHARPOSITION` and `CHARLENGTH` clauses it is possible to obtain the starting position and length (in terms of code units) of a graphemes sequence. The returned code unit values can then be used in other statements/clauses which require code unit operands (for example, in a `SUBSTRING` clause).

For further information on this syntax of the `EXAMINE` statement, see also *Unicode and Code Page Support* in the *Natural Programming Language*, section *Statements*, *EXAMINE*.

Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
operand1	C	S	A			U												yes	no
operand2	C	S					N	P	I		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		S					N	P	I									yes	no
operand7		S					N	P	I									yes	no

* Format B of *operand2* and *operand3* may be used only with a length of less than or equal to 4.

Syntax Element Description:

Syntax Element	Description
`FULL`	**FULL Option:** If `FULL` is specified for an operand, the entire value, including trailing blanks, will be processed. If `FULL` is not specified, trailing blanks in the operand will be ignored.
`SUBSTRING` *operand1 operand2 operand3*	**SUBSTRING Clause:** Normally, the content of a field is examined from the beginning of the field to the end or to the last non-blank character. With the `SUBSTRING` option, you examine only a certain part of the field. After the field name (*operand1*) in the `SUBSTRING` clause, you specify first the starting position (*operand2*) and then the length (*operand3*) of the field portion to be examined. *operand2* and *operand3* are specified in terms of code units. For example, to examine the 5th to 12th position inclusive of a field `#A`, you would specify: `EXAMINE SUBSTRING (#A,5,8)` **Note:** 1. If you omit *operand2*, the starting position is assumed to be `1`. 2. If you omit *operand3*, the length is assumed to be from the starting position to the end of the field. 3. If `SUBSTRING` is used in conjunction with a `DYNAMIC` variable, the field behaves like a fixed length variable; that is, the length (`*LENGTH`) does not change as a result of the `EXAMINE` operation, regardless of whether a `DELETE` or `REPLACE` operation was executed or not.
POSITION-clause	**POSITION Clause:**

Syntax Element	Description
	FROM and THRU positions are given in terms of code units. For further information, see *POSITION Clause* under *Syntax 1*.
CHARPOSITION *operand4*	**CHARPOSITION Clause:** *operand4* defines the starting position (in terms of Unicode graphemes) of the grapheme sequence. The according position in terms of code units is returned in *operand6*. This clause can be omitted if the CHARLENGTH clause is specified; in this case the starting position 1 is assumed.
CHARLENGTH *operand5*	**CHARLENGTH Clause:** *operand5* defines the length (in terms of Unicode graphemes) of the grapheme sequence. The length of the grapheme sequence in terms of code units is returned in *operand7*. This clause can be omitted if the CHARPOSITION clause is specified; in this case the length from the starting position up to the end of the string is returned.
GIVING POSITION IN *operand6*	**GIVING POSITION Clause:** *operand6* receives the starting position (in terms of code units) of the grapheme sequence defined by *operand4* and *operand5*. If *operand1* has less than *operand4* graphemes, 0 is returned. This clause can be omitted if the GIVING LENGTH clause is specified.
GIVING LENGTH IN *operand7*	**GIVING LENGTH Clause:** *operand7* receives the length (in terms of code units) of the grapheme sequence defined by *operand4* and *operand5*. If *operand1* has less than *operand4*+*operand5* graphemes, 0 is returned. This clause can be omitted if the GIVING POSITION clause is specified.

Notes:

1. Either the CHARPOSITION or the CHARLENGTH clause or both must be specified.
2. Either the GIVING POSITION or GIVING LENGTH clause or both must be specified.

Examples

- Example 1 - EXAMINE
- Example 2 - EXAMINE TRANSLATE
- Example 3 - EXAMINE AND REPLACE WITH MULTIPLE VALUES

- Example 4 - EXAMINE for Unicode Graphemes

Example 1 - EXAMINE

```
** Example 'EXMEX1': EXAMINE
************************************************************************
DEFINE DATA LOCAL
1 #TEXT   (A45)
1 #ARRAY  (A5/1:3)
1 #A      (A3)
1 #START  (N2)
1 #NUM    (N2)
1 #NUM1   (N2)
1 #NUM2   (N2)
1 #NUM3   (N2)
1 #POS    (N2)
1 #POS1   (N2)
1 #LENG   (N2)
1 #INDEX  (N2)
END-DEFINE
*
MOVE 'ABC   A B C   .A.  .B.  .C.    -A-  -B-  -C- ' TO #TEXT
*
WRITE / 'EXAMPLE 1  (DELIMITER, GIVING NUMBER)'
WRITE NOTITLE '#TEXT: ' #TEXT
EXAMINE #TEXT FOR 'A' GIVING NUMBER #NUM1
EXAMINE #TEXT FOR 'A' WITH DELIMITER GIVING NUMBER #NUM2
EXAMINE #TEXT FOR 'A' WITH DELIMITER '.' GIVING NUMBER #NUM3
WRITE 'EXAMINE #TEXT FOR "A" ' 57T 'Number found:' #NUM1
WRITE 'EXAMINE #TEXT FOR "A" WITH DELIMITER' 57T 'Number found:' #NUM2
WRITE 'EXAMINE #TEXT FOR "A" WITH DELIMITER "."'
  57T 'Number found:' #NUM3
*
WRITE / 'EXAMPLE 2  (DELIMITER, REPLACE, GIVING NUMBER)'
WRITE 'EXAMINE #TEXT FOR "A" WITH DELIMITER "-" REPLACE WITH "*"'
WRITE 'Before:' #TEXT
EXAMINE #TEXT FOR 'A' WITH DELIMITER '-' REPLACE WITH '*'
        GIVING NUMBER #NUM
WRITE 'After: ' #TEXT 57T 'Number found:' #NUM
*
*
NEWPAGE
*
WRITE / 'EXAMPLE 3  (REPLACE, GIVING NUMBER)'
WRITE 'EXAMINE      #TEXT FOR " " REPLACE WITH "+"'
WRITE 'Before:' #TEXT
EXAMINE #TEXT FOR ' ' REPLACE WITH '+' GIVING NUMBER #NUM
WRITE 'After: ' #TEXT 57T 'Number found:' #NUM
*
WRITE / 'EXAMPLE 4  (FULL, REPLACE, GIVING NUMBER)'
WRITE 'EXAMINE FULL #TEXT FOR " " REPLACE WITH "+"'
```

```
WRITE 'Before:' #TEXT
EXAMINE FULL #TEXT FOR ' ' REPLACE WITH '+' GIVING NUMBER #NUM
WRITE 'After: ' #TEXT 57T 'Number found:' #NUM
*
WRITE / 'EXAMPLE 5  (DELETE, GIVING POSITION)'
WRITE 'EXAMINE #TEXT FOR "+" DELETE GIVING POSITION #POS'
WRITE 'Before:' #TEXT
EXAMINE #TEXT FOR '+' DELETE GIVING POSITION #POS
WRITE 'After: ' #TEXT 57T 'Position found:' #POS
*
WRITE / 'EXAMPLE 6  (DELETE, GIVING LENGTH)'
WRITE 'EXAMINE #TEXT FOR "A" DELETE GIVING LENGTH #LENG'
WRITE 'Before:' #TEXT
EXAMINE #TEXT FOR 'A' DELETE GIVING LENGTH #LENG
WRITE 'After: ' #TEXT 57T 'Length found:' #LENG
*
*
NEWPAGE
*
MOVE 'ABC   A B C   .A.  .B.  .C.    -A-  -B-  -C- ' TO #TEXT
*
WRITE / 'EXAMPLE 7  (PATTERN, REPLACE, GIVING NUMBER)'
WRITE 'EXAMINE #TEXT FOR         ".A." AND REPLACE "***"'
WRITE 'Before:' #TEXT
EXAMINE #TEXT FOR         '.A.' AND REPLACE '***' GIVING NUMBER #NUM
WRITE 'After: ' #TEXT 57T 'Number found:' #NUM
*
MOVE 'ABC   A B C   .A.  .B.  .C.    -A-  -B-  -C- ' TO #TEXT
*
WRITE 'EXAMINE #TEXT FOR PATTERN ".A." AND REPLACE "***"'
WRITE 'Before:' #TEXT
EXAMINE #TEXT FOR PATTERN '.A.' AND REPLACE '***' GIVING NUMBER #NUM
WRITE 'After: ' #TEXT 57T 'Number found:' #NUM
*
MOVE 'ABC   A B C   .A.  .B.  .C.    -A-  -B-  -C- ' TO #TEXT
*
#A   := 'B C'
#POS := 6
#LENG:= 25
*
WRITE / 'EXAMPLE 8  (SUBSTRING, REPLACE, GIVING POSITION)'
WRITE '#A := "B C" ; #POS := 6 ; #LENG:= 25 '
WRITE 'EXAMINE SUBSTRING(#TEXT,#POS,#LENG) FOR #A AND REPLACE "***"'
WRITE 'Before:' #TEXT
EXAMINE SUBSTRING(#TEXT,#POS,#LENG) FOR #A AND REPLACE '***'
        GIVING POSITION #POS1
WRITE 'After: ' #TEXT 57T 'Position found:' #POS1
*
*
NEWPAGE
*
MOVE 'ABC   A B C   .A.  .B.  .C.    -A-  -B-  -C- ' TO #TEXT
```

```
*
WRITE / 'EXAMPLE 9  (DELETE, GIVING NUMBER, GIVING POSITION, '-
        'GIVING LENGTH)'
WRITE 'EXAMINE #TEXT FOR "." DELETE GIVING NUMBER   #NUM'
WRITE 30T 'GIVING POSITION #POS'
WRITE 30T 'GIVING LENGTH   #LENG'
WRITE 'Before:' #TEXT
EXAMINE #TEXT FOR '.' DELETE GIVING NUMBER #NUM
                             GIVING POSITION #POS
                             GIVING LENGTH   #LENG
WRITE 'After: ' #TEXT
WRITE 'Number found:  ' #NUM
WRITE 'Position found:' #POS
WRITE 'Length found:  ' #LENG
*
*
*
MOVE 'ABC  ' TO #ARRAY (1)
MOVE '.A.B.' TO #ARRAY (2)
MOVE '-A-B-' TO #ARRAY (3)
*
WRITE / 'EXAMPLE 10 (GIVING NUMBER, GIVING POSITION, GIVING INDEX)'
WRITE '#ARRAY(1):' #ARRAY(1)
WRITE '#ARRAY(2):' #ARRAY(2)
WRITE '#ARRAY(3):' #ARRAY(3)
WRITE 'EXAMINE #ARRAY(*) FOR "B" GIVING NUMBER   #NUM'
WRITE 27T 'GIVING POSITION #POS'
WRITE 27T 'GIVING INDEX    #INDEX'
EXAMINE #ARRAY(*) FOR 'B' GIVING NUMBER   #NUM
                          GIVING POSITION #POS
                          GIVING INDEX    #INDEX
WRITE 'Number found:  ' #NUM
WRITE 'Position found:' #POS
WRITE 'Index found:   ' #INDEX
END
```

Output of Program EXMEX1:

```
EXAMPLE 1  (DELIMITER, GIVING NUMBER)
#TEXT:  ABC   A B C   .A.  .B.  .C.    -A-  -B-  -C-
EXAMINE #TEXT FOR 'A'                                  Number found:   4
EXAMINE #TEXT FOR 'A' WITH DELIMITER                   Number found:   3
EXAMINE #TEXT FOR 'A' WITH DELIMITER '.'               Number found:   1

EXAMPLE 2  (DELIMITER, REPLACE, GIVING NUMBER)
EXAMINE #TEXT FOR 'A' WITH DELIMITER '-' REPLACE WITH '*'
Before: ABC   A B C   .A.  .B.  .C.    -A-  -B-  -C-
After:  ABC   A B C   .A.  .B.  .C.    -*-  -B-  -C-    Number found:   1

EXAMPLE 3  (REPLACE, GIVING NUMBER)
EXAMINE      #TEXT FOR ' ' REPLACE WITH '+'
Before: ABC   A B C   .A.  .B.  .C.    -*-  -B-  -C-
```

```
After:  ABC+++A+B+C+++.A.++.B.++.C.++++-*-++-B-++-C-    Number found:  20

EXAMPLE 4  (FULL, REPLACE, GIVING NUMBER)
EXAMINE FULL #TEXT FOR ' ' REPLACE WITH '+'
Before: ABC+++A+B+C+++.A.++.B.++.C.++++-*-++-B-++-C-
After:  ABC+++A+B+C+++.A.++.B.++.C.++++-*-++-B-++-C-+   Number found:   1

EXAMPLE 5  (DELETE, GIVING POSITION)
EXAMINE #TEXT FOR '+' DELETE GIVING POSITION #POS
Before: ABC+++A+B+C+++.A.++.B.++.C.++++-*-++-B-++-C-+
After:  ABCABC.A..B..C.-*--B--C-                        Position found:   4

EXAMPLE 6  (DELETE, GIVING LENGTH)
EXAMINE #TEXT FOR 'A' DELETE GIVING LENGTH #LENG
Before: ABCABC.A..B..C.-*--B--C-
After:  BCBC...B..C.-*--B--C-                           Length found:  21

EXAMPLE 7  (PATTERN, REPLACE, GIVING NUMBER)
EXAMINE #TEXT FOR         '.A.' AND REPLACE '***'
Before: ABC   A B C   .A.  .B.  .C.    -A-  -B-  -C-
After:  ABC   A B C   ***  .B.  .C.    -A-  -B-  -C-    Number found:   1
EXAMINE #TEXT FOR PATTERN '.A.' AND REPLACE '***'
Before: ABC   A B C   .A.  .B.  .C.    -A-  -B-  -C-
After:  ABC  ***B C   ***  .B.  .C.    ***  -B-  -C-    Number found:   3

EXAMPLE 8  (SUBSTRING, REPLACE, GIVING POSITION)
#A := 'B C' ; #POS := 6 ; #LENG:= 25
EXAMINE SUBSTRING(#TEXT,#POS,#LENG) FOR #A AND REPLACE '***'
Before: ABC   A B C   .A.  .B.  .C.    -A-  -B-  -C-
After:  ABC   A ***   .A.  .B.  .C.    -A-  -B-  -C-    Position found:   4

EXAMPLE 9  (DELETE, GIVING NUMBER, GIVING POSITION, GIVING LENGTH)
EXAMINE #TEXT FOR '.' DELETE GIVING NUMBER   #NUM
                             GIVING POSITION #POS
                             GIVING LENGTH   #LENG
Before: ABC   A B C   .A.  .B.  .C.    -A-  -B-  -C-
After:  ABC   A B C   A  B  C    -A-  -B-  -C-
Number found:     6
Position found:  15
Length found:    38

EXAMPLE 10 (GIVING NUMBER, GIVING POSITION, GIVING INDEX)
#ARRAY(1): ABC
#ARRAY(2): .A.B.
#ARRAY(3): -A-B-
EXAMINE #ARRAY(*) FOR 'B' GIVING NUMBER   #NUM
                          GIVING POSITION #POS
                          GIVING INDEX    #INDEX
Number found:     3
Position found:   2
Index found:      1
```

Example 2 - EXAMINE TRANSLATE

```
** Example 'EXMEX2': EXAMINE TRANSLATE
************************************************************************
DEFINE DATA LOCAL
1 #TEXT  (A50)
1 #TAB   (A2/1:10)
1 #POS   (N2)
1 #LENG  (N2)
END-DEFINE
*
MOVE 'ABC   A B C   .A.  .B.  .C.    -A-  -B-  -C- ' TO #TEXT
*
MOVE 'AX' TO #TAB(1)
MOVE 'BY' TO #TAB(2)
MOVE 'CZ' TO #TAB(3)
*
*
WRITE NOTITLE / 'EXAMPLE 1  (WITH TRANSLATION TABLE)'
WRITE 'EXAMINE #TEXT TRANSLATE USING #TAB(*)'
WRITE 'Before:' #TEXT
EXAMINE #TEXT TRANSLATE USING #TAB(*)
WRITE 'After: ' #TEXT
*
WRITE / 'EXAMPLE 2  (WITH INVERTED TRANSLATION TABLE)'
WRITE 'EXAMINE #TEXT TRANSLATE USING INVERTED #TAB(*)'
WRITE 'Before:' #TEXT
EXAMINE #TEXT TRANSLATE USING INVERTED #TAB(*)
WRITE 'After: ' #TEXT
*
#POS := 13
#LENG:= 15
*
WRITE / 'EXAMPLE 3  (WITH LOWER CASE TRANSLATION)'
WRITE '#POS := 13 ; #LENG:= 15 '
WRITE 'EXAMINE SUBSTRING(#TEXT,#POS,#LENG) TRANSLATE INTO LOWER CASE'
WRITE 'Before:' #TEXT
EXAMINE SUBSTRING(#TEXT,#POS,#LENG) TRANSLATE INTO LOWER CASE
WRITE 'After: ' #TEXT
*
END
```

Output of Program EXMEX2:

```
EXAMPLE 1  (WITH TRANSLATION TABLE)
EXAMINE #TEXT TRANSLATE USING #TAB(*)
Before: ABC   A B C   .A.  .B.  .C.    -A-  -B-  -C-
After:  XYZ   X Y Z   .X.  .Y.  .Z.    -X-  -Y-  -Z-

EXAMPLE 2  (WITH INVERTED TRANSLATION TABLE)
EXAMINE #TEXT TRANSLATE USING INVERTED #TAB(*)
Before: XYZ   X Y Z   .X.  .Y.  .Z.    -X-  -Y-  -Z-
After:  ABC   A B C   .A.  .B.  .C.    -A-  -B-  -C-

EXAMPLE 3  (WITH LOWER CASE TRANSLATION)
#POS := 13 ; #LENG:= 15
EXAMINE SUBSTRING(#TEXT,#POS,#LENG) TRANSLATE INTO LOWER CASE
Before: ABC   A B C   .A.  .B.  .C.    -A-  -B-  -C-
After:  ABC   A B C   .a.  .b.  .c.    -A-  -B-  -C-
```

Example 3 - EXAMINE AND REPLACE WITH MULTIPLE VALUES

```
*  EXAMPLE 'EXMEX3': EXAMINE AND REPLACE WITH MULTIPLE VALUES
************************************************************************
* This example shows a translation of the pattern
* 'AA', 'Aa' and 'aA' into '++',
* 'BB', 'Bb' and 'bB' into '--' and
* 'CC', 'Cc' and 'cC' into '**'.
************************************************************************
DEFINE DATA LOCAL
1 #SV     (A2/1:3,1:3) INIT (1,V) <'AA','BB','CC'>
                           (2,V) <'Aa','Bb','Cc'>
                           (3,V) <'aA','bB','cC'>
1 #RV     (A2/1:3)     INIT       <'++','--','**'>
1 #STRING (A20)        INIT <'AAABbbbbBCCCcccCaaaA'>
1 #NUM    (N2)
END-DEFINE
*
*
WRITE NOTITLE / 'EXAMINE #STRING FOR #SV(*,*) AND REPLACE WITH #RV(*)' /
*
WRITE 'Before:' #STRING   /* shows   'AAABbbbbBCCCcccCaaaA'
*
EXAMINE #STRING FOR #SV(*,*) AND REPLACE WITH #RV(*)
        GIVING NUMBER #NUM
*
WRITE 'After: ' #STRING   /* shows   '++A--bb--****c**aa++'
  40T 'Number found:' #NUM
*
```

Output of Program EXMEX3:

```
EXAMINE #STRING FOR #SV(*,*) AND REPLACE WITH #RV(*)

Before: AAABbbbbBCCCccccCaaaA
After:  ++A--bb--****c**aa++          Number found:   7
```

Example 4 - EXAMINE for Unicode Graphemes

This example demonstrates the analysis of a Unicode string containing the characters `ä` und `ü`. Both characters are defined as base character followed by a combining character: `ä` is coded with `U+0061` followed by `U+0308`, and `ü` is coded with `U+0075` followed by `U+0308`.

```
DEFINE DATA LOCAL
1 #U (U20)
1 #START (I2)
1 #POS (I2)
1 #LEN (I2)
END-DEFINE
#U := U'AB'-UH'00610308'-U'CD'-UH'00750308'-U'EF'
*
REPEAT
  #START := #START + 1
  EXAMINE #U FOR CHARPOSITION #START
                 CHARLENGTH      1
             GIVING POSITION IN #POS
                    LENGTH IN #LEN
*
  INPUT (AD=O) MARK POSITION #POS IN FIELD *#U
    '          UNICODE-STRING:' #U     (AD=MI)
 // '           CHARACTER NO.:' #START (EM=9)
  / 'STARTS AT BYTE POSITION:' #POS   (EM=9)
  / '       AND THE LENGTH IS:' #LEN   (EM=9)
WHILE #POS NE 0
END-REPEAT
END
```

Output:

<table>
<tr><th>Mainframe Environments:</th><th>Windows and Linux Environments (with Natural Web I/O Interface):</th></tr>
<tr><td><pre>UNICODE-STRING: ABa?CDu?EF

         CHARACTER NO.: 1
STARTS AT BYTE POSITION: 1
      AND THE LENGTH IS: 1</pre></td><td><pre>UNICODE-STRING: ABäCDüEF

         CHARACTER NO.: 1
STARTS AT BYTE POSITION: 1
      AND THE LENGTH IS: 1</pre></td></tr>
<tr><td>Press ENTER to continue.</td><td>Press ENTER to continue.</td></tr>
<tr><td><pre>UNICODE-STRING: ABa?CDu?EF

         CHARACTER NO.: 2
STARTS AT BYTE POSITION: 2
      AND THE LENGTH IS: 1</pre></td><td><pre>UNICODE-STRING: ABäCDüEF

         CHARACTER NO.: 2
STARTS AT BYTE POSITION: 2
      AND THE LENGTH IS: 1</pre></td></tr>
<tr><td>Press ENTER to continue.</td><td>Press ENTER to continue.</td></tr>
<tr><td colspan="2">Note that the character in position 3 is a combining character sequence and is two code units long.</td></tr>
<tr><td><pre>UNICODE-STRING: ABa?CDu?EF

         CHARACTER NO.: 3
STARTS AT BYTE POSITION: 3
      AND THE LENGTH IS: 2</pre></td><td><pre>UNICODE-STRING: ABäCDüEF

         CHARACTER NO.: 3
STARTS AT BYTE POSITION: 3
      AND THE LENGTH IS: 2</pre></td></tr>
<tr><td>And so on.</td><td>And so on.</td></tr>
</table>

70 EXPAND

```
EXPAND { dynamic-clause } [GIVING operand5]
       { array-clause   }
```

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

Related statements: `REDUCE` | `RESIZE`

Belongs to Function Group: *Memory Management Control for Dynamic Variables or X-Arrays*

Function

The `EXPAND` statement is used to expand:

- the allocated length of a dynamic variable (*`dynamic-clause`*), or
- the number of occurrences of X-arrays (*`array-clause`*).

For further information, see the following sections in the *Programming Guide*:

- *Using Dynamic Variables*
- *Allocating/Freeing Memory Space for a Dynamic Variable*
- *X-Arrays*
- *Storage Management of X-Group Arrays*

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1		S	A			A	U					B							no	no
operand2	C	S								I									no	no
operand3			A	G		A	U	N	P	I	F	B	D	T	L	C	G	O	yes	no
operand4	C	S						N	P	I									no	no
operand5		S								I4									no	yes

Syntax Element Description:

Syntax Element	Description
dynamic-clause	**Dynamic Clause:** The `EXPAND DYNAMIC VARIABLE` statement expands the allocated length of a dynamic variable (*operand1*) to the value specified with *operand2*. For more information, see ***Dynamic Clause*** below.
operand1	**Dynamic Variable:** *operand1* is the dynamic variable for which the size is to be expanded.
operand2	**Target Length of Dynamic Variable:** *operand2* is used to specify the length to which the dynamic variable is to be expanded. The value specified must be a non-negative integer constant or a variable of type integer.
array-clause	**Array Clause:** The `EXPAND ARRAY` statement increases the number of occurrences of the X-array (*operand3*) to the upper and lower bound specified with (*dim*[,*dim*[,*dim*]]). For more information, see ***Array Clause*** below.
operand3	**X-Array:** *operand3* is the X-array for which the number of occurrences may be increased. The index notation of the array is optional. As index notation only the complete range notation * is allowed for each dimension.
dim *operand4*	**Dimension:** The lower and upper bound notation (*operand4* or asterisk) to which the X-array should be expanded is specified here. If the current value of the upper or lower bound should be used, an asterisk (*) may be specified in place of *operand4*. For more information, see ***Dimension*** below.
GIVING *operand5*	**GIVING Clause:** If the `GIVING` clause is not specified, Natural runtime error processing is triggered if an error occurs. If the `GIVING` clause is specified, *operand5* contains the Natural message number if an error occurred, or zero upon success.

Dynamic Clause

`[SIZE OF] DYNAMIC [VARIABLE] operand1 TO operand2`

The `EXPAND DYNAMIC VARIABLE` statement expands the allocated size of a dynamic variable (*operand1*) to the value specified with *operand2*.

If *operand2* is less than the currently allocated length of *operand1*, the statement will be ignored for this dynamic variable. The currently allocated length (`*LENGTH`) of the dynamic variable is not modified.

Array Clause

`[AND RESET] [OCCURRENCES OF] ARRAY operand3 TO (dim[,dim [,dim]])`

The `EXPAND ARRAY` statement increases the number of occurrences of the X-array (*operand3*) to the upper and lower bound specified with `TO (dim [,dim[,dim]])`.

The `RESET` option resets all occurrences of the expanded X-array to its default zero value. By default (no `RESET` option), the actual values are kept and the expanded (new) occurrences are reset.

When using the `EXPAND` statement, it is only possible to increase the number of occurrences. If the requested number is smaller than the currently allocated number of occurrences, it will simply be ignored.

An upper or lower bound used in an `EXPAND` statement must be exactly the same as the corresponding upper or lower bound defined for the array.

Example:

```
DEFINE DATA LOCAL
1 #a(I4/1:*)
1 #g(1:*)
  2 #ga(I4/1:*)

1 #i(i4)
END-DEFINE
...
/* allocating #a(1:10)
EXPAND ARRAY #a TO (1:10)       /* #a is allocated 10
EXPAND ARRAY #a TO (*:10)       /* occurrences.


/* allocating #ga(1:10,1:20)
EXPAND ARRAY #g TO (1:10)       /* 1st dimension is set to (1:10)
EXPAND ARRAY #ga TO (*:*,1:20)  /* 1st dimension is dependent and
                                /* therefore kept with (*:*)
                                /* 2nd dimension is set to (1:20)
```

```
EXPAND ARRAY #a TO (5:10)       /* This is rejected because the lower index
                                /* must be 1 or *
EXPAND ARRAY #a TO (#i:10)      /* This is rejected because the lower index
                                /* must be 1 or *

EXPAND ARRAY #ga TO (1:10,1:20) /* (1:10) for the 1st dimension is rejected
                                /* because the dimension is dependent and
                                /* must be specified with (*:*).
```

For further information, see the following topics in the *Programming Guide*:

- *Storage Management of X-Arrays*
- *Storage Management of X-Group Arrays*

Dimension

Each of the dimensions (`dim`) specified in the ***Array Clause*** is defined using the following syntax:

$$\left\{ \begin{array}{l} * \\ \left\{ \begin{array}{l} * \\ \mathit{operand4} \end{array} \right\} : \left\{ \begin{array}{l} * \\ \mathit{operand4} \end{array} \right\} \end{array} \right\}$$

The lower and upper bound notation (`operand4` or asterisk) to which the X-array should be expanded is specified here. If the current value of the upper or lower bound should be used, an asterisk (*) may be specified in place of `operand4`. Instead of `*:*`, you may also specify a single asterisk.

The number of dimensions (`dim`) must exactly match the defined number of dimensions of the X-array (1, 2 or 3).

If the number of occurrences for a specified dimension is less than the number of the currently allocated occurrences, the number of occurrences is not changed for the corresponding dimension.

IX

71 FETCH

```
FETCH [ { REPEAT } ] operand1 [operand2 [(parameter)]] ...
        { RETURN }
```

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

Related Statements: `CALL` | `CALL FILE` | `CALL LOOP` | `CALLNAT` | `DEFINE SUBROUTINE` | `ESCAPE` | `FETCH` | `PERFORM`

Belongs to Function Group: ***Invoking Programs and Routines***

Function

The `FETCH` statement is used to execute a Natural object program written as a main program. The program to be loaded must have been previously stored in the Natural system file (cataloged or stowed). Execution of the `FETCH` statement does not overwrite any source program in the Natural source work area.

For Natural RPC: See *Notes on Natural Statements on the Server* (in the *Natural RPC (Remote Procedure Call)* documentation).

Additional Considerations

In addition to the parameters passed explicitly with `FETCH`, the fetched program also has access to the established global data area.

The `FETCH` statement may cause the internal execution of an `END TRANSACTION` statement based on the setting of the Natural profile parameter `OPRB` (Database Open/Close Processing) as set by the Natural administrator. If a logical transaction is to span multiple Natural programs, the Natural administrator should be consulted to ensure that the `OPRB` parameter is set correctly.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S				A													yes	no
operand2	C	S	A	G		A	U	N	P	I	F	B	D	T	L		G		yes	yes

Syntax Element Description:

Syntax Element	Description
REPEAT	**REPEAT Option:** The REPEAT option causes Natural to suppress the prompt for user input for each INPUT statement issued during the execution of the FETCHed program. It may be used to send information about the execution of the program to the terminal without the user having to reply with ENTER.
RETURN	**RETURN Option:** Without the specification of RETURN, the execution of the program issuing the FETCH statement will be terminated immediately and the fetched program will be activated as a "main program" (Level 1). If a program is invoked with FETCH RETURN, the execution of the invoking program will be suspended - not terminated - and the FETCHed program will be activated as a "subordinate program" on a higher level. Control is returned to the invoking program when an END or ESCAPE ROUTINE statement is encountered in the FETCHed program. Processing is continued with the statement following the FETCH RETURN statement.
operand1	**Program Name:** The name of the program module (maximum 8 characters) can be specified as an alphanumeric constant or the content of an alphanumeric variable of length 1 to 8. Natural will attempt to locate the program in the library currently active at the time the FETCH statement is issued. If the program is not found, Natural will attempt to locate the program in the steplibs. If the program is still not found, an error message will be issued. The program name may contain an ampersand (&); at execution time, this character will be replaced by the one-character code corresponding to the current value of the system variable *LANGUAGE. This makes it possible, for example, to invoke different programs for the processing of input, depending on the language in which input is provided.
operand2	**Passing Parameter Fields:** The FETCH statement may also be used to pass parameter fields to the invoked program. A parameter field may be defined with any format. The parameters are converted to a format suitable for a corresponding INPUT field. All parameters are placed on the top of the Natural stack. The parameter fields can be read by the FETCHed program using an INPUT statement. The first INPUT statement will result in the insertion of all parameter field values into the fields specified in the INPUT statement. The INPUT statement must have the sign position specification (session parameter SG=ON) for parameter fields defined with numeric format, because each parameter field defined with numeric format in the FETCH statement will receive a sign position if its value is negative. If more parameters are passed than are read by the next INPUT statement, the extra parameters are ignored. The number of parameters may be obtained with the Natural system variable *DATA.

Syntax Element	Description
	Note: If *operand2* is a time variable (format T), only the time component of the variable content is passed, but not the date component.
parameter	**Date Format:** If *operand2* is a date variable, you can specify the session parameter DF (Date Format) as *parameter* for this variable.

Example

Invoking Program:

```
** Example 'FETEX1': FETCH  (with parameter)
************************************************************************
DEFINE DATA LOCAL
1 #PNUM (N8)
1 #FNC  (A1)
END-DEFINE
*
INPUT 10X 'SELECTION MENU FOR EMPLOYEES SYSTEM' /
      10X '-' (35) //
      10X 'ADD      (A)' /
      10X 'UPDATE   (U)' /
      10X 'DELETE   (D)' /
      10X 'STOP     (.)' //
      10X 'PLEASE ENTER FUNCTION: ' #FNC ///
      10X 'PERSONNEL NUMBER:' #PNUM
*
DECIDE ON EVERY VALUE OF #FNC
  VALUE 'A', 'U', 'D'
    IF #PNUM = 0
      REINPUT 'PLEASE ENTER A VALID NUMBER' MARK *#PNUM
    END-IF
  VALUE 'A'
    FETCH 'FETEXAD' #PNUM
  VALUE 'U'
    FETCH 'FETEXUP' #PNUM
  VALUE 'D'
    FETCH 'FETEXDE' #PNUM
  VALUE '.'
    STOP
  NONE
    REINPUT 'PLEASE ENTER A VALID FUNCTION' MARK *#FNC
END-DECIDE
*
END
```

Invoked Program FETEXAD:

```
** Example 'FETEXAD': FETCH (called by FETEX1)
************************************************************************
DEFINE DATA LOCAL
1 #PERS-NR (N8)
END-DEFINE
*
INPUT #PERS-NR
*
WRITE *PROGRAM 'Record added with personnel number:' #PERS-NR
*
END
```

Invoked Program FETEXUP:

```
** Example 'FETEXUP': FETCH (called by FETEX1)
************************************************************************
DEFINE DATA LOCAL
1 #PERS-NR (N8)
END-DEFINE
*
INPUT #PERS-NR
*
WRITE *PROGRAM 'Record updated with personnel number:' #PERS-NR
*
END
```

Invoked Program FETEXDE:

```
** Example 'FETEXDE': FETCH (called by FETEX1)
************************************************************************
DEFINE DATA LOCAL
1 #PERS-NR (N8)
END-DEFINE
*
INPUT #PERS-NR
*
WRITE *PROGRAM 'Record deleted with personnel number:' #PERS-NR
*
END
```

Output of Program FETEX1:

```
SELECTION MENU FOR EMPLOYEES SYSTEM
-----------------------------------

ADD      (A)
UPDATE   (U)
DELETE   (D)
STOP     (.)

PLEASE ENTER FUNCTION: D


PERSONNEL NUMBER: 1150304
```

After entering and confirming function and personnel number:

```
Page      1                                                   05-01-13  11:58:46

FETEXDE  Record deleted with personnel number:   1150304
```

72 FIND

Related Statements: ACCEPT/REJECT | AT BREAK | AT START OF DATA | AT END OF DATA | BACKOUT TRANSACTION | BEFORE BREAK PROCESSING | DELETE | END TRANSACTION | GET | GET SAME | GET TRANSACTION | HISTOGRAM | LIMIT | PASSW | PERFORM BREAK PROCESSING | READ | RETRY | STORE | UPDATE

Belongs to Function Group: ***Database Access and Update***

Function

The `FIND` statement is used to select a set of records from the database based on search criteria consisting of fields defined as descriptors (keys).

This statement causes a processing loop to be initiated and then executed for each record selected. Each field in each record may be referenced within the processing loop. It is not necessary to issue a `READ` statement following the `FIND` in order to reference the fields within each record selected.

See also the following sections in the *Programming Guide*:

- *FIND Statement*
- *Loop Processing*
- *Referencing of Database Fields Using (r) Notation*

Database-Specific Considerations

Database	Explanation
SQL	`FIND FIRST` as well as the `PASSWORD`, `CIPHER`, `COUPLED` and `RETAIN` clauses are not permitted. `FIND UNIQUE` is not permitted. The `SORTED BY` clause corresponds with the SQL clause `ORDER BY`. The basic search criteria for an SQL-database table may be specified in the same manner as for an Adabas file. The term record used in this context corresponds with the SQL term “row”.
XML	`FIND FIRST`, as well as the `PASSWORD`, `CIPHER`, `COUPLED` and `RETAIN` clauses are not permitted. `FIND UNIQUE` is not permitted. The basic search criteria for an XML-database may be specified in the same manner as for an Adabas file. The term record used in this context corresponds with the XML term “XML object”.

System Variables Available with the FIND Statement

The Natural system variables `*ISN`, `*NUMBER`, and `*COUNTER` are automatically created for each `FIND` statement issued. A reference number must be supplied if the system variable was referenced outside the current processing loop or through a `FIND UNIQUE`, `FIND FIRST`, or `FIND NUMBER` statement. The format/length of each of these system variables is P10; this format/length cannot be changed.

System Variable	Availability/Usage
`*ISN`	■ **Adabas** `*ISN` contains the Adabas internal sequence number (ISN) of the record currently being processed. `*ISN` is not available for the `FIND NUMBER` statement. ■ **Tamino** `*ISN` contains the XML object ID. ■ **SQL** `*ISN` is not available. ■ **Entire System Server** `*ISN` is not available.
`*NUMBER`	See system variable **NUMBER* in the *System Variables* documentation. With Entire System Server, `*NUMBER` is not available.
`*COUNTER`	The system variable `*COUNTER` contains the number of times the processing loop has been entered.

See also ***Example 13 - Using System Variables with the FIND Statement***.

Issuing Multiple FIND Statements

Multiple `FIND` statements may be issued to create nested loops whereby an inner loop is entered for each record selected in the outer loop.

See also ***Example 14 - Multiple FIND Statements***.

Restrictions

With Entire System Server, `FIND NUMBER` and `FIND UNIQUE` as well as the `PASSWORD`, `CIPHER`, `COUPLED` and `RETAIN` clauses are not permitted.

Syntax 1 - FIND Statement with Processing Loop

```
FIND  [ { ALL        } ]  [MULTI-FETCH-clause] [RECORDS] [IN] [FILE] view-name
        { (operand1) }
      [PASSWORD=operand2]
      [CIPHER=operand3]
      [WITH] [[LIMIT] (operand4)]  basic-search-criteria
      [COUPLED-clause] ...  4/42
      [STARTING WITH ISN=operand5]
      [SORTED-BY-clause]
      [RETAIN-clause]
      [[IN] SHARED HOLD [MODE=option]]
      [SKIP [RECORDS] IN HOLD]
      [WHERE-clause]
      [IF-NO-RECORDS-FOUND-clause]
      statement ...
END-FIND                                  (structured mode only)
LOOP                                      (reporting mode only)
```

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

Syntax 2 - FIND Statement without Processing Loop

```
FIND [ { FIRST  } ]  [RECORDS] [IN] [FILE]  view-name
       { NUMBER }
       { UNIQUE }
     [PASSWORD=operand2]
     [CIPHER=operand3]
     [WITH] [[LIMIT] (operand4)]  basic-search-criteria
     [COUPLED-clause] ...  4/42
```

```
[SORTED-BY-clause]    (only for FIND FIRST)
[RETAIN-clause]
[WHERE-clause]
```

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

Syntax Description

Operand Definition Table:

Operand	Possible Structure				Possible Formats	Referencing Permitted	Dynamic Definition
operand1	C	S			N P I B*	yes	no
operand2	C	S			A	yes	no
operand3	C	S			N	yes	no
operand4	C	S			N P I B*	yes	no
operand5	C	S			N P I B*	yes	no

* Format B of *operand1*, *operand4* and *operand5* may be used only with a length of less than or equal to 4.

Syntax Element Description:

Syntax Element	Description
ALL/*operand1*	**Processing Limit:** The number of records to be processed from the selected set may be limited by specifying *operand1* (enclosed in parentheses, immediately after the keyword FIND) - either as a numeric constant (in the range from 0 to 4294967295) or as the name of a numeric variable. ALL may be optionally specified. It emphasizes that all selected records are to be processed. If you specify a limit with *operand1*, this limit applies to the FIND loop being initiated. Records rejected for processing by the WHERE clause are not counted against this limit.

Syntax Element	Description

	`FIND (5) IN EMPLOYEES WITH ...`
	
	`MOVE 10 TO #CNT(N2)`
	`FIND (#CNT) EMPLOYEES WITH ...`
	
	For this statement, the specified limit has priority over a limit set with a `LIMIT` statement.
	
	If a smaller limit is set with the `LT` parameter, the `LT` limit applies.
	
	Note:
	
	1. If you wish to process a 4-digit number of records, specify it with a leading zero: (`0`*nnnn*); because Natural interprets every 4-digit number enclosed in parentheses as a line-number reference to a statement.
	2. *operand1* has no influence on the size of an ISN set that is to be retained by a `RETAIN` clause. *operand1* is evaluated when the `FIND` loop is entered. If the value of *operand1* is modified within the `FIND` loop, this does not affect the number of records processed.

`FIND FIRST` | `FIND NUMBER` | `FIND UNIQUE`	**FIND FIRST, FIND NUMBER, FIND UNIQUE Option:** These options are used ■ to select the first record of a selected set (see ***FIND FIRST***), ■ to determine the number of records in a selected set (see ***FIND NUMBER***), or ■ to ensure that only one record satisfies a selection criterion (see ***FIND UNIQUE***). For a detailed description of these options, see below.
MULTI-FETCH-clause	**MULTI-FETCH Clause:** For Adabas databases, Natural offers a `MULTI-FETCH` clause that allows you to read more than one record per database access. For further information, see ***MULTI-FETCH Clause***.
view-name	**View Name:** The name of a view as defined either within a `DEFINE DATA` block or in a separate global or local data. In reporting mode, *view-name* is the name of a DDM if no `DEFINE DATA LOCAL` statement is used.
`PASSWORD=`*operand2*	**PASSWORD Clause:** The `PASSWORD` clause applies only for Adabas databases. This clause is not permitted with Entire System Server.

Syntax Element	Description
	The PASSWORD clause is used to provide a password (*operand2*) when reading/writing data from an Adabas file which is password protected. If you require access to a password-protected file, contact the person responsible for database security concerning password usage/assignment. If the PASSWORD clause is omitted, the default password specified with the PASSW statement applies. The password value must not be changed during the execution of a processing loop. See also ***Example 1 - PASSWORD Clause***.
CIPHER=*operand3*	**CIPHER Clause:** The CIPHER clause only applies to Adabas databases. This clause is not permitted with Entire System Server. The CIPHER clause is used to provide a cipher key (*operand3*) when retrieving data from Adabas files which are enciphered. If you require access to an enciphered file, contact the person responsible for database security concerning cipher key usage/assignment. The cipher key may be specified as a numeric constant with 8 digits or as a user-defined variable with format/length N8. The value of the cipher key must not be changed during the processing of a loop initiated by a FIND statement. See also ***Example 2 - CIPHER Clause***.
WITH LIMIT *operand4* *basic-search-criteria*	**WITH Clause:** The WITH clause is required. It is used to specify the *basic-search-criteria* (see ***Search Criteria for Adabas Files***) consisting of key fields (descriptors) defined in the database. The following database-specific consideration applies. You may use Adabas descriptors, subdescriptors, superdescriptors, hyperdescriptors, and phonetic descriptors within a WITH clause. A non-descriptor (that is, a field marked in the DDM with N) can also be specified. The number of records to be selected as a result of a WITH clause may be limited by specifying the keyword LIMIT together with a numeric constant or a user-defined variable, enclosed within parentheses, which contains the limit value (*operand4*, range from 1 to 4294967295). If the number of records selected exceeds the limit, the program will be terminated with an error message.

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td></td><td>Note: If the limit is to be a 4-digit number, specify it with a leading zero (<code>0nnnn</code>); because Natural interprets every 4-digit number enclosed in parentheses as a line-number reference to a statement.</td></tr>
<tr><td>COUPLED-clause</td><td>COUPLED Clause:

This clause may be used to specify a search which involves the use of the Adabas coupling facility. See COUPLED Clause.</td></tr>
<tr><td>STARTING WITH ISN=operand5</td><td>STARTING WITH Clause:

This clause may be used for repositioning within a <code>FIND</code> loop whose processing has been interrupted. See STARTING WITH Clause.</td></tr>
<tr><td>SORTED-BY-clause</td><td>SORTED BY Clause:

This clause may be used to cause Adabas to sort the selected records based on the sequence of one to three descriptors. See SORTED BY Clause.</td></tr>
<tr><td>RETAIN-clause</td><td>RETAIN Clause:

This clause may be used to retain the result of an extensive search in large files for further processing. See RETAIN Clause.</td></tr>
<tr><td>[[IN] SHARED HOLD [MODE=option]]</td><td>SHARED HOLD Clause

Note: This clause can be used only for access to Adabas.

This clause can be used to place records being read in a “shared hold” state. A record can be put in shared hold by many users at the same time. As long as a record is in a shared hold state, it is protected from being updated, because it cannot be set into an exclusive hold by parallel users. This ensures data consistency for the record data, as no one can update the record while it is being processed.

Especially if the same record is fetched with multiple statements to read different MU/PE occurrences (<code>GET SAME</code> statement) or to browse over a LOB field in a piecemeal technique (<code>READLOB</code> statement), the shared hold state can guarantee data stability over this transaction without blocking the record for other users.

Although such a hold state is an efficient way to protect read sequences, it is a basic and important matter when to release the record again from this “soft lock”. Since this question depends on individual application aspects, different options can be selected with the <code>MODE</code> subclause.
<table>
<tr><th>MODE Option</th><th>Hold Period</th><th>Explanation</th></tr>
<tr><td>C</td><td>Only at the moment of reading the record.</td><td>Ensures only that the record version being read has been committed by the last user who updated the record. This</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><td></td><td></td><td>option does not really set a lock in hold state, but checks only that the record is not in exclusive hold by another user at time of read.</td></tr>
<tr><td>Q</td><td>Until the next record in a sequence is read.</td><td>Releases the record from shared hold when
- the next record is read in the loop sequence or
- the loop is terminated or
- an END TRANSACTION or BACKOUT TRANSACTION is executed.</td></tr>
<tr><td>S</td><td>Until the logical transaction is terminated.</td><td>Releases the record from shared hold when a logical transaction is terminated with an END TRANSACTION or BACKOUT TRANSACTION statement.</td></tr>
</table>
MODE=Q and MODE=S ensure that the record being read cannot be updated concurrently by other users until it has been released from hold again.

If the MODE subclause is not specified, MODE=C is the default.

See also Example 15 - SHARED HOLD Clause below.
</td></tr>
<tr><td>SKIP RECORDS IN HOLD</td><td>SKIP RECORDS Clause:

Note: This clause can be used only for access to Adabas.

Whenever a record is going to be read with hold, a Natural error NAT3145 (Adabas response code 145) might happen if the record is in hold by another user at this time. This occurs if a shared hold is requested and the record is in exclusive hold or if an exclusive hold is requested and the record is in either exclusive or shared hold.

Although error NAT3145 is surely the right reaction to assure a “clean data processing”, sometimes it might be useful if a record in hold could be skipped. If it is alright that such a record will not be processed and the loop processing should continue, the SKIP RECORDS clause should be used.
</td></tr>
</table>

Syntax Element	Description
	If the SKIP RECORDS clause is applied, Natural first tries to read the record with hold. If the record is already in hold and a Natural error NAT3145 would occur, ■ no error processing is initiated; ■ the record (currently in hold by another user) is instantly re-fetched without hold, but not processed in terms of the program logic; ■ the record which comes next after the skipped record is read with hold and the processing continues. See also ***Example 16 - SKIP RECORDS Clause***.
WHERE-clause	**WHERE Clause:** This clause may be used to specify an additional selection criterion (*logical-condition*). See ***WHERE Clause***.
IF-NO-RECORDS-FOUND-clause	**IF NO RECORDS FOUND Clause:** This clause may be used to cause a processing loop initiated with a FIND statement to be entered in the event that no records meet the selection criteria specified in the WITH clause and the WHERE clause. See ***IF NO RECORDS FOUND Clause***.
END-FIND	**End of FIND Statement:** In structured mode with processing loop, the Natural reserved keyword END-FIND must be used to end the FIND statement. In reporting mode with processing loop, the Natural statement LOOP is used to end the FIND statement.
LOOP	

FIND FIRST

The FIND FIRST statement may be used to select and process the first record which meets the WITH and WHERE criteria.

For Adabas databases, the record processed will be the record with the lowest Adabas ISN from the set of qualifying records.

This statement does *not* initiate a processing loop.

Restrictions with FIND FIRST

- FIND FIRST can only be used in reporting mode.
- FIND FIRST is not available for SQL databases.

System Variables Available with FIND FIRST

The following Natural system variables are available with the FIND FIRST statement:

System Variable	Explanation
*ISN	The system variable *ISN contains the Adabas ISN of the selected record. *ISN will be zero if no record is found after the evaluation of the WITH and WHERE criteria. *ISN is not available with Entire System Server.
*NUMBER	The system variable *NUMBER contains the number of records found after the evaluation of the WITH criterion and before evaluation of any WHERE criteria. *NUMBER will be zero if no record meets the WITH criterion. *NUMBER is not available with Entire System Server.
*COUNTER	The system variable *COUNTER contains 1 if a record was found; contains 0 if no record was found.

Example of FIND FIRST Statement: See the program FNDFIR (reporting mode)

FIND NUMBER

The FIND NUMBER statement is used to determine the number of records which satisfy the WITH/WHERE criteria specified. It does *not* result in the initiation of a processing loop and *no data fields from the database are made available*.

Note: Use of the WHERE clause may result in significant overhead.

Restrictions with FIND NUMBER

- The WHERE clause can only be used in reporting mode.
- FIND NUMBER is not available with Entire System Server.

System Variables Available with FIND NUMBER

The following Natural system variables are available with the FIND NUMBER statement:

System Variable	Explanation
*NUMBER	The system variable *NUMBER contains the number of records found after the evaluation of the WITH criterion.
*COUNTER	The system variable *COUNTER contains the number of records found after the evaluation of the WHERE criterion. *COUNTER is only available if the FIND NUMBER statement contains a WHERE clause.

Example for FIND NUMBER: See the program FNDNUM (reporting mode).

FIND UNIQUE

The FIND UNIQUE statement may be used to ensure that only one record is selected for processing. It does *not* result in the initiation of a processing loop. If a WHERE clause is specified, an automatic internal processing loop is created to evaluate the WHERE clause.

If no records or more than one record satisfy the criteria, an error message will be issued. This condition can be tested with the ON ERROR statement.

Restrictions with FIND UNIQUE

- FIND UNIQUE can only be used in reporting mode.
- FIND UNIQUE is not available with Entire System Server.
- For SQL databases, FIND UNIQUE cannot be used. (Exception: On mainframe computers, FIND UNIQUE can be used for primary keys; however, this is only permitted for compatibility reasons and should not be used.)

System Variables Available with FIND UNIQUE

System Variable	Explanation
*ISN	The system variable *ISN contains the unique ISN number of the record, which itself must be unique.
*NUMBER	The system variable *NUMBER always contains 1 for a valid FIND UNIQUE execution. *NUMBER may contain any other positive value (= 0 or >= 2) if an error has occurred. This error condition may be used by the ON ERROR statement. *NUMBER is not allowed if the WHERE clause is missing.
*COUNTER	The system variable *COUNTER contains the number of records found after the evaluation of the WHERE criterion. *COUNTER is not allowed if the WHERE clause is missing.

Example for FIND UNIQUE: See the Program FNDUNQ (reporting mode).

MULTI-FETCH Clause

Note: This clause can only be used for Adabas databases.

```
             { ON                    }
MULTI-FETCH  { OFF                   }
             { [OF] multi-fetch-factor }
```

Note: [MULTI-FETCH OF *multi-fetch-factor*] is supported for database types ADA/ADA2. The default processing mode is applied; see profile parameter MFSET. The MULTI-FETCH clause is ignored in case Adabas LA or large objects fields are used or a view size greater than 64KB is defined.

For more information, see the section *MULTI-FETCH Clause* (Adabas) in the *Programming Guide*.

Search Criteria for Adabas Files

```
1 descriptor [(i)]  { EQ       }  value { [ OR { EQ       } value ]...                   }
                    { =        }        {      { =        }                              }
                    { EQUAL    }        {      { EQUAL    }                              }
                    { EQUAL TO }        {      { EQUAL TO }                              }
                                        { THRU value [BUT NOT value [THRU value]]         }

2 descriptor [(i)]  { EQ            }  value
                    { =             }
                    { EQUAL         }
                    { EQUAL TO      }
                    { NE            }
                    { <>            }
                    { NOT =         }
                    { NOT EQ        }
                    { NOTEQUAL      }
                    { NOT EQUAL     }
                    { NOT EQUAL TO  }
                    { LT            }
                    { LESS THAN     }
                    { <             }
                    { GE            }
                    { GREATER EQUAL }
                    { >=            }
                    { NOT <         }
                    { NOT LT        }
                    { GT            }
                    { GREATER THAN  }
                    { >             }
                    { LE            }
                    { LESS EQUAL    }
                    { <=            }
                    { NOT >         }
                    { NOT GT        }

3 set-name
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
descriptor		S	A			A	U	N	P	I	F	B	D	T	L				no	no
value	C	S				A	U	N	P	I	F	B	D	T	L				yes	no
set-name	C	S				A													no	no

Syntax Element Description:

Syntax Element	Description
descriptor	**Descriptor:** Adabas descriptor, subdescriptor, superdescriptor, hyperdescriptor, or phonetic descriptor. A field marked as non-descriptor in the DDM can also be specified.
(*i*)	**Index Specification:** A descriptor contained within a periodic group may be specified with or without an index. If no index is specified, the record will be selected if the value specified is located in any occurrence. If an index is specified, the record is selected only if the value is located in the occurrence specified by the index. The index specified must be a constant. An index range must not be used. No index must be specified for a descriptor which is a multiple-value field. The record will be selected if the value is located in the record regardless of the position of the value.
value	**Search Value:** The formats of the descriptor and the search value must be compatible.
set-name	**Set Name:** Identifies a set of records previously selected with a `FIND` statement in which the `RETAIN` clause was specified. The set referenced in a `FIND` must have been created from the same physical Adabas file. *set-name* may be specified as a text constant (maximum 32 characters) or as the content of an alphanumeric variable. *set-name* cannot be used with Entire System Server.

See also:

- *Example 3 - Basic Search Criteria in WITH Clause*
- *Example 4 - Basic Search Criteria with Multiple-Value Field*

Search Criterion with Null Indicator

```
null-indicator { =           } value
               { EQ          }
               { EQUAL [TO]  }
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
null-indicator		S								I									no	no
value	C	S						N	P	I	F	B							yes	no

Syntax Element Description:

Syntax Element	Description	
null-indicator	The null indicator.	
value	**Possible Values**	**Meaning**
	`-1`	The corresponding field contains no value.
	`0`	The corresponding field does contain a value.

Connecting Search Criteria (for Adabas Files)

basic-search-criteria can be combined using the Boolean operators AND, OR, and NOT. Parentheses may also be used to control the order of evaluation. The order of evaluation is as follows:

1. `( )`: Parentheses
2. `NOT`: Negation (only for *basic-search-criteria* of form [2]).
3. `AND`: AND operation
4. `OR`: OR operation

basic-search-criteria may be connected by logical operators to form a complex *search-expression*. The syntax for such a complex *search-expression* is as follows:

```
[NOT] { basic-search-criteria } [ { OR  } search-expression ] ...
      { (search-expression)   }   { AND }
```

See also ***Example 5 - Various Samples of Complex Search Expression in WITH Clause***.

Descriptor-Key Usage

Adabas users may use database fields which are defined as descriptors to construct basic search criteria.

Subdescriptors, Superdescriptors, Hyperdescriptors and Phonetic Descriptors

With Adabas, subdescriptors, superdescriptors, hyperdescriptors and phonetic descriptors may be used to construct search criteria.

- A subdescriptor is a descriptor formed from a portion of a field.
- A superdescriptor is a descriptor whose value is formed from one or more fields or portions of fields.
- A hyperdescriptor is a descriptor which is formed using a user-defined algorithm.
- A phonetic descriptor is a descriptor which allows the user to perform a phonetic search on a field (for example, a person's name). A phonetic search results in the return of all values which sound similar to the search value.

Which fields may be used as descriptors, subdescriptors, superdescriptors, hyperdescriptors and phonetic descriptors with which file is defined in the corresponding DDM.

Values for Subdescriptors, Superdescriptors, Phonetic Descriptors

Values used with these types of descriptors must be compatible with the internal format of the descriptor. The internal format of a subdescriptor is the same as the format of the field from which the subdescriptor is derived. The internal format of a superdescriptor is binary if all of the fields from which it is derived are defined with numeric format; otherwise, the format is alphanumeric. Phonetic descriptors always have alphanumeric format.

Values for subdescriptors and superdescriptors may be specified in the following ways:

- Numeric or hexadecimal constants may be specified. A hexadecimal constant must be used for a value for a superdescriptor which has binary format (see above).
- Values in user-defined variable fields may be specified using the `REDEFINE` statement to select the portions that form the subdescriptor or superdescriptor value.

Using Descriptors Contained within a Database Array

A descriptor which is contained within a database array may also be used in the construction of basic search criterion. For Adabas databases, such a descriptor may be a multiple-value field or a field contained within a periodic group.

A descriptor contained within a periodic group may be specified with or without an index. If no index is specified, the record will be selected if the value specified is located in any occurrence. If an index is specified, the record is selected only if the value is located in the occurrence specified by the index. The index specified must be a constant. An index range must not be used.

No index must be specified for a descriptor which is a multiple-value field. The record will be selected if the value is located in the record regardless of the position of the value.

See also *Example 6 - Various Samples Using Database Arrays*.

COUPLED Clause

This clause only applies to Adabas databases.

This clause is not permitted with Entire System Server.

```
{ AND }  COUPLED   [TO] [FILE] view-name
{ OR  }
                   [                     { EQ         }              ]
                   [ VIA descriptor1     { =          } descriptor2  ]
                   [                     { EQUAL [TO] }              ]

                   [WITH]
                   basic-search-criteria
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
descriptor1		S	A			A	N	P			B							no	no
descriptor2		S	A			A	N	P			B							no	no

Note: Without the `VIA` clause, the `COUPLED` clause may be specified up to 4 times; with the `VIA` clause, it may be specified up to 42 times.

The `COUPLED` clause is used to specify a search which involves the use of the Adabas coupling facility. This facility permits database descriptors from different files to be specified in the search criterion of a single `FIND` statement.

The same Adabas file must not be used in two different `FIND COUPLED` clauses within the same `FIND` statement.

A *`set-name`* (see ***RETAIN Clause***) must not be specified in the *`basic-search-criteria`*.

Database fields in a file specified within the `COUPLED` clause are not available for subsequent reference in the program unless another `FIND` or `READ` statement is issued separately against the coupled file.

Note: If the `COUPLED` clause is used, the main `WITH` clause may be omitted. If the main `WITH` clause is omitted, the keywords `AND`/`OR` of the `COUPLED` clause must not be specified.

Physical Coupling without VIA Clause

The files used in a `COUPLED` clause without `VIA` must be physically coupled using the appropriate Adabas utility (as described in the Adabas documentation).

See also ***Example 7 - Using Physically Coupled Files***.

The reference to `NAME` in the `DISPLAY` statement of the above example is valid since this field is contained in the `EMPLOYEES` file, whereas a reference to `MAKE` would be invalid since `MAKE` is contained in the `VEHICLES` file, which was specified in the `COUPLED` clause.

In this example, records will be found only if `EMPLOYEES` and `VEHICLES` have been physically coupled.

Logical Coupling - VIA Clause

The option `VIA` *`descriptor1`* `=` *`descriptor2`* allows you to logically couple multiple Adabas files in a search query, where:

- *`descriptor1`* is a field from the first view.
- *`descriptor2`* is a field from the second view.

The two files need not be physically coupled in Adabas.

See also ***Example 8 - VIA Clause***.

STARTING WITH Clause

This clause applies only to Adabas databases.

You can use this clause to specify as *operand5* an Adabas ISN (internal sequence number) which is to be used as a start value for the selection of records. *operand5* must be in the range from `0` to `4294967295`.

This clause may be used for repositioning within a `FIND` loop whose processing has been interrupted, to easily determine the next record with which processing is to continue. This is particularly useful if the next record cannot be identified uniquely by any of its descriptor values. It can also be useful in a distributed client/server application where the reading of the records is performed by a server program while further processing of the records is performed by a client program, and the records are not processed all in one go, but in batches.

Note: The start value actually used will not be the value of *operand5*, but the next higher value.

Example:

See the program `FNDSISN` in the library `SYSEXSYN`.

SORTED BY Clause

This clause only applies to Adabas, Tamino and SQL databases.

This clause is not permitted with Entire System Server.

```
SORTED [BY] descriptor ... 3 [DESCENDING]
```

The `SORTED BY` clause is used to cause Adabas to sort the selected records based on the sequence of one to three descriptors. The descriptors used for controlling the sort sequence may be different from those used for selection.

By default, the records are sorted in *ascending* sequence of values; if you want them to be in descending sequence, specify the keyword `DESCENDING`. The sort is performed using the Adabas inverted lists and does not result in any records being read.

Note: The use of this clause may result in significant overhead if any descriptor used to control the sort sequence contains a large number of values. This is because the entire value list may have to be scanned until all selected records have been located in the list. When a large number of records is to be sorted, you should use the `SORT` statement.

Adabas sort limits (see the `ADARUN LS` parameter in the Adabas documentation) are in effect when the `SORTED BY` clause is used.

A descriptor which is contained in a periodic group must not be specified in the `SORTED BY` clause. A multiple-value field (without an index) may be specified.

Non-descriptors may also be specified in the `SORTED BY` clause. However, this function is not available on mainframes.

If the `SORTED BY` clause is used, the `RETAIN` clause must not be used.

See also *Example 9 - SORTED BY Clause*.

Considerations for Combined Use of STARTING WITH and SORTED BY Clauses

If both the `STARTING WITH` and the `SORTED BY` clause are used in the same `FIND` statement and the underlying database is Adabas, the following should be considered.

With Adabas for Mainframes

On Adabas for Mainframes, the `FIND` statement is executed in the following steps:

1. All records matching the search criterion are gathered and put in ISN sequence.
2. The records are sorted by the descriptor specified in the `SORTED BY` clause.
3. The record whose ISN value is specified in the `STARTING WITH` clause is positioned in the "sorted-by-descriptor" record list.
4. The records following the record found under Step 3 are returned in the `FIND` loop.

With Adabas for OpenSystems

On Adabas for OpenSystems (Linux and Cloud or Windows) the same statement is executed as follows:

1. All records matching the search criterion are gathered and put in ISN sequence.
2. The record whose ISN value is specified in the `STARTING WITH` clause is positioned in the "sorted-by-ISN" record list.
3. All records following the record found under Step 2 are sorted by the descriptor specified in the `SORTED BY` clause and returned in the `FIND` loop.

Example:

If the following program is executed with Adabas for Mainframes and Adabas on Linux and Cloud/Windows:

```
DEFINE DATA LOCAL
1 V1 VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
1 #ISN (I4)
END-DEFINE
FORMAT NL=5 SG=OFF PS=43 AL=15
*
PRINT   'FIND' (I)
FIND V1 WITH NAME = 'B' THRU 'BALBIN'
  RETAIN AS 'SET1'
  IF *COUNTER = 4 THEN
    #ISN := *ISN
  END-IF
  DISPLAY *ISN V1
END-FIND
*
PRINT / 'FIND .. SORTED BY NAME' (I)
FIND V1 WITH 'SET1'
  SORTED BY NAME
  DISPLAY *ISN V1
END-FIND
*
PRINT / 'FIND .. STARTING WITH ISN = ' (I) #ISN (AD=I)
FIND V1 WITH 'SET1'
  STARTING WITH ISN = #ISN
  DISPLAY *ISN V1
END-FIND
*
PRINT / 'FIND .. STARTING WITH ISN = ' (I) #ISN (AD=I)
        ' .. SORTED BY NAME' (I)
FIND V1 WITH 'SET1'
  STARTING WITH ISN = #ISN
  SORTED BY NAME
  DISPLAY *ISN V1
END-FIND
END
```

The result is as follows:

Results on Natural for Mainframes

```
ISN        NAME         FIRST-NAME         CITY
----- --------------- --------------- ---------------

FIND V1 WITH NAME = 'B' THRU 'BALBIN'
   12 BAILLET         PATRICK         LYS LEZ LANNOY
   58 BAGAZJA         MARJAN          MONTHERME
  351 BAECKER         JOHANNES        FRANKFURT
  355 BAECKER         KARL            SINDELFINGEN
```

```
  370 BACHMANN        HANS            MUENCHEN
  490 BALBIN          ENRIQUE         BARCELONA
  650 BAKER           SYLVIA          OAK BROOK
  913 BAKER           PAULINE         DERBY

FIND .. SORTED BY NAME
  370 BACHMANN        HANS            MUENCHEN
  351 BAECKER         JOHANNES        FRANKFURT
  355 BAECKER         KARL            SINDELFINGEN
   58 BAGAZJA         MARJAN          MONTHERME
   12 BAILLET         PATRICK         LYS LEZ LANNOY
  650 BAKER           SYLVIA          OAK BROOK
  913 BAKER           PAULINE         DERBY
  490 BALBIN          ENRIQUE         BARCELONA

FIND .. STARTING WITH ISN = 355
  370 BACHMANN        HANS            MUENCHEN
  490 BALBIN          ENRIQUE         BARCELONA
  650 BAKER           SYLVIA          OAK BROOK
  913 BAKER           PAULINE         DERBY

FIND .. STARTING WITH ISN = 355 .. SORTED BY NAME
   58 BAGAZJA         MARJAN          MONTHERME
   12 BAILLET         PATRICK         LYS LEZ LANNOY
  650 BAKER           SYLVIA          OAK BROOK
  913 BAKER           PAULINE         DERBY
  490 BALBIN          ENRIQUE         BARCELONA
```

Results on Natural for OpenSystems

```

ISN        NAME         FIRST-NAME         CITY
----- --------------- --------------- ---------------

FIND V1 WITH NAME = 'B' THRU 'BALBIN'
   12 BAILLET         PATRICK         LYS LEZ LANNOY
   58 BAGAZJA         MARJAN          MONTHERME
  351 BAECKER         JOHANNES        FRANKFURT
  355 BAECKER         KARL            SINDELFINGEN
  370 BACHMANN        HANS            MUENCHEN
  490 BALBIN          ENRIQUE         BARCELONA
  650 BAKER           SYLVIA          OAK BROOK
  913 BAKER           PAULINE         DERBY

FIND .. SORTED BY NAME
  370 BACHMANN        HANS            MUENCHEN
  351 BAECKER         JOHANNES        FRANKFURT
  355 BAECKER         KARL            SINDELFINGEN
   58 BAGAZJA         MARJAN          MONTHERME
   12 BAILLET         PATRICK         LYS LEZ LANNOY
  650 BAKER           SYLVIA          OAK BROOK
  913 BAKER           PAULINE         DERBY
```

```
  490 BALBIN          ENRIQUE         BARCELONA

FIND .. STARTING WITH ISN = 355
  370 BACHMANN        HANS            MUENCHEN
  490 BALBIN          ENRIQUE         BARCELONA
  650 BAKER           SYLVIA          OAK BROOK
  913 BAKER           PAULINE         DERBY

FIND .. STARTING WITH ISN = 355 .. SORTED BY NAME
  370 BACHMANN        HANS            MUENCHEN
  650 BAKER           SYLVIA          OAK BROOK
  913 BAKER           PAULINE         DERBY
  490 BALBIN          ENRIQUE         BARCELONA
```

A `FIND` statement with at most one of these options (`SORTED BY` or `STARTING WITH ISN`) always returns the same records in the same sequence, regardless under which system the statement is executed. If, however, both clauses are used together, the result returned depends on which Adabas platform is used to serve the database statement.

Therefore, if a Natural program is intended to be used on multiple platforms, the combination of a `SORTED BY` and `STARTING WITH ISN` clause in the same `FIND` statement should be avoided.

RETAIN Clause

This clause only applies to Adabas databases.

This clause is not permitted with Entire System Server.

```
RETAIN AS operand6
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
operand6	C	S				A											yes	no

Syntax Element Description:

Syntax Element	Description
`RETAIN AS`	**Retain Result:** By using the `RETAIN` clause, the result of an extensive search in large files can be retained for further processing. The selection is retained as an `ISN-set` in the Adabas work file. The set may be used in subsequent `FIND` statements as a basic search criterion for further refinement of the set or for further processing of the records.

Syntax Element	Description
	The set created is file-specific and may only be used in another `FIND` statement that processes the same file. The set may be referenced by any Natural program.
operand6	**Set Name:** The set name is used to identify the record set. It may be specified as an alphanumeric constant or as the content of an alphanumeric user-defined variable. Duplicate set names are not checked; consequently, if a duplicate set name is specified, the new set replaces the old set.

See also ***Example 10 - RETAIN Clause***.

Releasing Sets

There is no specific limit for the number of sets that can be retained or the number of ISNs in a set. It is recommended that the minimum number of ISN sets needed at one time be defined. Sets that are no longer needed should be released using the `RELEASE SETS` statement.

If they are not released with a `RELEASE` statement, retained sets exist until the end of the Natural session, or until a logon to another library, when they are released automatically. A set created by one program may be referenced by another program for processing or further refinement using additional search criteria.

Updates by Other Users

The records identified by the ISNs in a retained set are not locked against access and/or update by other users. Before you process records from the set, it is therefore useful to check whether the original search criteria which were used to create the set are still valid: This check is done with another `FIND` statement, using the set name in the `WITH` clause as a basic search criterion and specifying in a `WHERE` clause the original search criteria (that is, the basic search criteria as specified in the `WITH` clause of the `FIND` statement which was used to create the set).

Restriction

If the `RETAIN` clause is used, the `SORTED BY` clause must not be used.

WHERE Clause

```
WHERE logical-condition
```

The `WHERE` clause may be used to specify an additional selection criterion (*`logical-condition`*) which is evaluated *after* a value has been read and *before* any processing is performed on the value (including the `AT BREAK` evaluation).

The syntax for a *`logical-condition`* is described in the section *Logical Condition Criteria* in the *Programming Guide*.

If a processing limit is specified in a FIND statement containing a WHERE clause, records which are rejected as a result of the WHERE clause are *not* counted against the limit. These records are, however, counted against a global limit specified in the Natural session parameter LT, the GLOBALS command, or LIMIT statement.

See also *Example 11 - WHERE Clause*.

IF NO RECORDS FOUND Clause

Structured Mode Syntax

```
IF NO [RECORDS] [FOUND]
{   ENTER          }
{   statement ...  }
END-NOREC
```

Reporting Mode Syntax

```
IF NO [RECORDS] [FOUND]
{   ENTER                       }
{   statement                   }
{   DO statement ... DOEND      }
```

Syntax Element Description:

Syntax Element	Description
IF NO RECORDS FOUND	**IF NO RECORDS FOUND Clause:** The IF NO RECORDS FOUND clause may be used to cause a processing loop initiated with a FIND statement to be entered in the event that no records meet the selection criteria specified in the WITH clause and the WHERE clause. If no records meet the specified WITH and WHERE criteria, the IF NO RECORDS FOUND clause causes the FIND processing loop to be executed once with an "empty" record. If this is not desired, specify the statement ESCAPE BOTTOM within the IF NO RECORDS FOUND clause.
ENTER *statement ...*	**Statement Execution:** If one or more statements are specified with the IF NO RECORDS FOUND clause, the statements will be executed immediately before the processing loop is entered. If no statements are to be executed before entering the loop, the keyword ENTER must be used.

Syntax Element	Description
`END-NOREC`	**End of IF NO RECORDS FOUND Clause:** In structured mode, the Natural reserved word `END-NOREC` must be used to end the `IF NO RECORDS FOUND` clause. In reporting mode, use the `DO ... DOEND` statements to supply one or several suitable statements, depending on the situation, and to end the `IF NO RECORDS FOUND` clause. If you specify only a single statement or the keyword `ENTER` (see above), you can omit the `DO ... DOEND` statements. With respect to good coding practice, this is not recommended.

`ENTER`
`statement`
`DO` *`statement`* `...` `DOEND`	

See also ***Example 12 - IF NO RECORDS FOUND Clause***.

Database Values

Unless other value assignments are made in the statements accompanying an `IF NO RECORDS FOUND` clause, Natural will reset to empty all database fields which reference the file specified in the current loop.

Evaluation of System Functions

Natural system functions are evaluated once for the empty record that is created for processing as a result of the `IF NO RECORDS FOUND` clause.

Restriction

This clause cannot be used with `FIND FIRST`, `FIND NUMBER` and `FIND UNIQUE`.

Examples

- Example 15 - SHARED HOLD Clause
- Example 16 - SKIP RECORDS Clause

See also the example for `FIND NUMBER`: program `FNDNUM`.

Example 1 - PASSWORD Clause

```
** Example 'FNDPWD': FIND (with PASSWORD clause)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 PERSONNEL-ID
*
1 #PASSWORD (A8)
END-DEFINE
*
INPUT 'ENTER PASSWORD FOR EMPLOYEE FILE:' #PASSWORD (AD=N)
LIMIT 2
*
FIND EMPLOY-VIEW PASSWORD = #PASSWORD
                 WITH NAME = 'SMITH'
  DISPLAY NOTITLE NAME PERSONNEL-ID
END-FIND
*
END
```

Output of Program FNDPWD:

```
ENTER PASSWORD FOR EMPLOYEE FILE:
```

Example 2 - CIPHER Clause

```
** Example 'FNDCIP': FIND (with PASSWORD/CIPHER clause)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 PERSONNEL-ID
*
1 #PASSWORD (A8)
1 #CIPHER   (N8)
END-DEFINE
*
LIMIT 2
INPUT 'ENTER PASSWORD FOR EMPLOYEE FILE:   ' #PASSWORD (AD=N)
    / 'ENTER CIPHER KEY FOR EMPLOYEE FILE: ' #CIPHER   (AD=N)
*
FIND EMPLOY-VIEW PASSWORD = #PASSWORD
```

```
                 CIPHER   = #CIPHER
                 WITH NAME = 'SMITH'
  DISPLAY NOTITLE NAME PERSONNEL-ID
END-FIND
*
END Output of Program FNDCIP:
```

```
ENTER PASSWORD FOR EMPLOYEE FILE:
ENTER CIPHER KEY FOR EMPLOYEE FILE:
```

Example 3 - Basic Search Criteria in WITH Clause

```
FIND STAFF WITH NAME = 'SMITH'
FIND STAFF WITH CITY NE 'BOSTON'
FIND STAFF WITH BIRTH = 610803
FIND STAFF WITH BIRTH = 610803 THRU 610811
FIND STAFF WITH NAME = 'O HARA' OR = 'JONES' OR = 'JACKSON'
FIND STAFF WITH PERSONNEL-ID = 100082 THRU 100100
                                 BUT NOT 100087 THRU 100095
```

Example 4 - Basic Search Criteria with Multiple-Value Field

When the descriptor used in the basic search criteria is a multiple-value field, basically four different kinds of results can be obtained (the field `MU-FIELD` in the following examples is assumed to be a multiple-value field):

```
FIND XYZ-VIEW WITH MU-FIELD = 'A'
```

This statement returns records in which *at least one* occurrence of `MU-FIELD` has the value `A`.

```
FIND XYZ-VIEW WITH MU-FIELD NOT EQUAL 'A'
```

This statement returns records in which *at least one* occurrence of `MU-FIELD` does *not* have the value `A`.

```
FIND XYZ-VIEW WITH NOT MU-FIELD NOT EQUAL 'A'
```

This statement returns records in which *every* occurrence of `MU-FIELD` has the value `A`.

```
FIND XYZ-VIEW WITH NOT MU-FIELD = 'A'
```

This statement returns records in which *none* of the occurrences of `MU-FIELD` has the value `A`.

Example 5 - Various Samples of Complex Search Expression in WITH Clause

```
FIND STAFF WITH BIRTH LT 19770101 AND DEPT = 'DEPT06'
```

```
FIND STAFF WITH JOB-TITLE = 'CLERK TYPIST'
                    AND (BIRTH GT 19560101 OR LANG = 'SPANISH')
```

```
FIND STAFF WITH JOB-TITLE = 'CLERK TYPIST'
                    AND NOT (BIRTH GT 19560101 OR LANG = 'SPANISH')
```

```
FIND STAFF WITH DEPT = 'ABC' THRU 'DEF'
                    AND CITY = 'WASHINGTON' OR = 'LOS ANGELES'
                    AND BIRTH GT 19360101
```

```
FIND CARS WITH MAKE = 'VOLKSWAGEN'
                   AND COLOR = 'RED' OR = 'BLUE' OR = 'BLACK'
```

Example 6 - Various Samples of Using Database Arrays

The following examples assume that the field `SALARY` is a descriptor contained within a periodic group, and the field `LANG` is a multiple-value field.

```
FIND EMPLOYEES WITH SALARY LT 20000
```

Results in a search of all occurrences of `SALARY`.

```
FIND EMPLOYEES WITH SALARY (1) LT 20000
```

Results in a search of the first occurrence only.

```
FIND EMPLOYEES WITH SALARY (1:4) LT 20000    /* invalid
```

A range specification must not be specified for a field within a periodic group used as a search criterion.

```
FIND EMPLOYEES WITH LANG = 'FRENCH'
```

Results in a search of all values of `LANG`.

```
FIND EMPLOYEES WITH LANG (1) = 'FRENCH'        /* invalid
```

An index must not be specified for a multiple-value field used as a search criterion.

Example 7 - Using Physically Coupled Files

```
** Example 'FNDCPL': FIND (using coupled files)
** NOTE: Adabas files must be physically coupled when using the
**       COUPLED clause without the VIA clause.
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
1 VEHIC-VIEW VIEW OF VEHICLES
  2 MAKE
END-DEFINE
*
FIND EMPLOY-VIEW WITH CITY = 'FRANKFURT'
     AND COUPLED TO
     VEHIC-VIEW WITH MAKE = 'VW'
  DISPLAY NOTITLE NAME
END-FIND
*
END
```

Example 8 - VIA Clause

```
** Example 'FNDVIA': FIND (with VIA clause)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
1 VEHIC-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
END-DEFINE
*
FIND EMPLOY-VIEW WITH NAME = 'ADKINSON'
     AND COUPLED TO VEHIC-VIEW
     VIA PERSONNEL-ID = PERSONNEL-ID WITH MAKE = 'VOLVO'
  DISPLAY PERSONNEL-ID NAME FIRST-NAME
END-FIND
*
END
```

Output of Program FNDVIA:

```
Page      1                                                  05-01-17  13:18:22

PERSONNEL         NAME              FIRST-NAME
   ID
--------- -------------------- --------------------

20011000  ADKINSON             BOB
```

Example 9 - SORTED BY Clause

```
** Example 'FNDSOR': FIND (with SORTED BY clause)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
  2 PERSONNEL-ID
END-DEFINE
*
LIMIT 10
FIND EMPLOY-VIEW WITH CITY = 'FRANKFURT'
                 SORTED BY NAME PERSONNEL-ID

  DISPLAY NOTITLE NAME (IS=ON) FIRST-NAME PERSONNEL-ID
END-FIND
*
END
```

Output of Program FNDSOR:

```
        NAME              FIRST-NAME      PERSONNEL
                                             ID
-------------------- -------------------- ---------

BAECKER              JOHANNES             11500345
BECKER               HERMANN              11100311
BERGMANN             HANS                 11100301
BLAU                 SARAH                11100305
BLOEMER              JOHANNES             11200312
DIEDRICHS            HUBERT               11600301
DOLLINGER            MARGA                11500322
FALTER               CLAUDIA              11300311
                     HEIDE                11400311
FREI                 REINHILD             11500301
```

Example 10 - RETAIN Clause

```
** Example 'RELEX1': FIND (with RETAIN clause and RELEASE statement)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 BIRTH
  2 NAME
*
1 #BIRTH (D)
END-DEFINE
*
MOVE EDITED '19400101' TO #BIRTH (EM=YYYYMMDD)
*
FIND NUMBER EMPLOY-VIEW WITH BIRTH GT #BIRTH
     RETAIN AS 'AGESET1'
IF *NUMBER = 0
  STOP
END-IF
*
FIND EMPLOY-VIEW WITH 'AGESET1' AND CITY = 'NEW YORK'
  DISPLAY NOTITLE NAME CITY BIRTH (EM=YYYY-MM-DD)
END-FIND
*
RELEASE SET 'AGESET1'
*
END
```

Output of Example 10:

```
        NAME                 CITY             DATE
                                               OF
                                              BIRTH
-------------------- -------------------- ----------

RUBIN                NEW YORK             1945-10-27
WALLACE              NEW YORK             1945-08-04
```

Example 11 - WHERE Clause

```
** Example 'FNDWHE': FIND (with WHERE clause)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 CITY
END-DEFINE
```

```
*
FIND EMPLOY-VIEW WITH CITY = 'PARIS'
                 WHERE JOB-TITLE = 'INGENIEUR COMMERCIAL'
  DISPLAY NOTITLE
          CITY JOB-TITLE PERSONNEL-ID NAME
END-FIND
*
END
```

Output of Program FNDWHE:

```
        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
```

Example 12 - IF NO RECORDS FOUND Clause

```
** Example 'FNDIFN': FIND (using IF NO RECORDS FOUND)
************************************************************************
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 FNDIFN:

```
        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 ***
```

Example 13 - Using System Variables with the FIND Statement

```
** Example 'FNDVAR': FIND (using *ISN, *NUMBER, *COUNTER)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 CITY
END-DEFINE
*
LIMIT 3
FIND EMPLOY-VIEW WITH CITY = 'MADRID'
  DISPLAY NOTITLE PERSONNEL-ID NAME
                  *ISN *NUMBER *COUNTER
END-FIND
*
END
```

Output of Program FNDVAR

```
PERSONNEL         NAME             ISN         NMBR        CNT
   ID
--------- -------------------- ----------- ----------- -----------

60000114  DE JUAN                      400          41           1
60000136  DE LA MADRID                 401          41           2
60000209  PINERO                       405          41           3
```

Example 14 - Multiple FIND Statements

In the following example, first all people named SMITH are selected from the `EMPLOYEES` file. Then the `PERSONNEL-ID` from the `EMPLOYEES` file is used as the search key for an access to the `VEHICLES` file.

```
** Example 'FNDMUL': FIND (with multiple files)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
1 VEHIC-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 15
EMP. FIND EMPLOY-VIEW WITH NAME = 'SMITH'
  /*
  VEH. FIND VEHIC-VIEW WITH PERSONNEL-ID = EMP.PERSONNEL-ID
    IF NO RECORDS FOUND
      MOVE '*** NO CAR ***' TO MAKE
    END-NOREC
    DISPLAY NOTITLE
            EMP.NAME (IS=ON)
            EMP.FIRST-NAME (IS=ON)
            VEH.MAKE
  END-FIND
END-FIND
END
```

Output of Program FNDMUL:

The resulting report shows the `NAME` and `FIRST-NAME` (obtained from the `EMPLOYEES` file) of all people named SMITH as well as the `MAKE` of each car (obtained from the `VEHICLES` file) owned by these people.

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


SMITH                GERHARD              ROVER
                     SEYMOUR              *** NO CAR ***
                     MATILDA              FORD
                     ANN                  *** NO CAR ***
                     TONI                 TOYOTA
                     MARTIN               *** NO CAR ***
                     THOMAS               FORD
                     SUNNY                *** NO CAR ***
                     MARK                 FORD
                     LOUISE               CHRYSLER
                     MAXWELL              MERCEDES-BENZ
                                          MERCEDES-BENZ
                     ELSA                 CHRYSLER
                     CHARLY               CHRYSLER
                     LEE                  *** NO CAR ***
                     FRANK                FORD
```

Example 15 - SHARED HOLD Clause

```
FIND EMPL-VIEW WITH NAME = ...
     IN SHARED HOLD MODE=Q  /* Record in shared hold until next record is read.
   ...
   GET EMPL-VIEW  *ISN      /* The record remains unchanged!
   ...
END-FIND
```

Example 16 - SKIP RECORDS Clause

```
FIND EMPL-VIEW WITH NAME = ...  /* Records found are put in hold while reading.
     SKIP RECORDS IN HOLD       /* Records already held by other users are
  ...                           /* skipped to prevent error NAT3145.
  UPDATE
  END TRANSACTION
END-FIND
```

73 FOR

```
FOR  operand1      { [:]=  }     { operand2                }
                   { EQ    }     { (arithmetic-expression) }
                   { FROM  }

                   { TO    }     { operand3                }
                   { THRU  }     { (arithmetic-expression) }

                   [ STEP        { operand4                } ]
                                 { (arithmetic-expression) }

              statement ...
END-FOR       (structured mode only)
LOOP          (reporting mode only)
```

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

Related Statements: `REPEAT` | `ESCAPE`

Belongs to Function Group: ***Loop Execution***

Function

The `FOR` statement is used to initiate a processing loop and to control the number of times the loop is processed.

Consistency Check

Before the `FOR` loop is entered, the values of the operands are checked to ensure that they are consistent (that is, the value of *operand3* can be reached or exceeded by repeatedly adding *operand4* to *operand2*). If the values are not consistent, the `FOR` loop is not entered (however, no error message is output, except when the `STEP` value is zero).

Syntax Description

Operand Definition Table:

Operand	Possible Structure						Possible Formats												Referencing Permitted	Dynamic Definition
operand1		S						N	P	I	F								yes	yes
operand2	C	S			N			N	P	I	F								yes	no
arithmetic-expression		S						N	P	I	F								no	no
operand3	C	S			N			N	P	I	F								yes	no

Operand	Possible Structure						Possible Formats												Referencing Permitted	Dynamic Definition
operand4	C	S			N			N	P	I	F								yes	no

Syntax Element Description:

Syntax Element	Description
operand1 *operand2*	**Loop Control Variable (*operand1*) and Initial Setting (*operand2*):** *operand1* is used to control the number of times the processing loop is to be executed. It may be a database field or a user-defined variable. The value specified after the keyword `FROM` (*operand2*) is assigned to the loop control variable field before the processing loop is entered for the first time. This value is incremented (or decremented if the `STEP` value is negative) using the value specified after the `STEP` keyword (*operand4*) each additional time the loop is processed. The loop control variable value may be referenced during the execution of the processing loop and will contain the current value of the loop control variable. **Note:** The keywords `[:]=`, `EQ` or `FROM` can be omitted.
operand3	**TO Value:** The processing loop is terminated when *operand1* is greater than (or less than if the initial value of the `STEP` value was negative) the value specified for *operand3*. **Note:** The keyword `TO` or `TRU` can be omitted.
`STEP` *operand4*	**STEP Value:** The `STEP` value may be positive or negative. If a `STEP` value is not specified, an increment of `+1` is used. The compare operation will be adjusted to “less than” or “greater than” depending on the sign of the `STEP` value when the loop is entered for the first time. **Note:** 1. *operand4* must not be zero. 2. The keyword `STEP` can be omitted.
(*arithmetic-expression*)	**Arithmetic Expression:** In place of *operand2*, *operand3* or *operand4*, any arithmetic expression may be specified.

Syntax Element	Description
	Note: 1. The arithmetic expressions must be enclosed in parentheses. 2. The preceding keyword cannot be omitted. For further information on arithmetic expressions, see *arithmetic-expression* in the COMPUTE statement description.
END-FOR	**End of FOR Statement:** In structured mode, the Natural reserved word END-FOR must be used to end the FOR statement. In reporting mode, the Natural statement LOOP is used to end the FOR statement.
LOOP	

Example

```
** Example 'FOREX1S': FOR (structured mode)
************************************************************************
DEFINE DATA LOCAL
1 #INDEX (I1)
1 #ROOT  (N2.7)
END-DEFINE
*
FOR #INDEX 1 TO 5
  COMPUTE #ROOT = SQRT (#INDEX)
  WRITE NOTITLE '=' #INDEX 3X '=' #ROOT
END-FOR
*
SKIP 1
FOR #INDEX 1 TO 5 STEP 2
  COMPUTE #ROOT = SQRT (#INDEX)
  WRITE '=' #INDEX 3X '=' #ROOT
END-FOR
*
END
```

Output of Program FOREX1S:

```
#INDEX:    1   #ROOT:   1.0000000
#INDEX:    2   #ROOT:   1.4142135
#INDEX:    3   #ROOT:   1.7320508
#INDEX:    4   #ROOT:   2.0000000
#INDEX:    5   #ROOT:   2.2360679

#INDEX:    1   #ROOT:   1.0000000
#INDEX:    3   #ROOT:   1.7320508
#INDEX:    5   #ROOT:   2.2360679
```

Equivalent reporting-mode example: FOREX1R.

74 FORMAT

FORMAT [(*rep*)] *parameter* ...

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

Related Statements: AT END OF PAGE | AT TOP OF PAGE | CLOSE PRINTER | DEFINE PRINTER | DISPLAY | EJECT | NEWPAGE | PRINT | SKIP | SUSPEND IDENTICAL SUPPRESS | WRITE | WRITE TITLE | WRITE TRAILER

Belongs to Function Group: ***Creation of Output Reports***

Function

The FORMAT statement is used to specify input and output parameter settings.

Settings specified with a FORMAT statement override (at compilation time) default settings in effect for the session that have been set by a GLOBALS command, SET GLOBALS statement, or by the Natural administrator.

These settings may in turn be overridden by parameters specified in a DISPLAY, INPUT, PRINT, WRITE, WRITE TITLE, or WRITE TRAILER statement.

The settings remain in effect until the end of a program or until another FORMAT statement is encountered.

A FORMAT statement does not generate any executable code in the Natural program. It is not executed in dependence of the logical flow of a program. It is evaluated during program compilation in order to set parameters for compiling DISPLAY, WRITE, PRINT and INPUT statements. The settings defined with a FORMAT statement are applicable to all DISPLAY, WRITE, PRINT and INPUT statements which follow.

Syntax Description

Syntax Element	Description
(*rep*)	**Report Specification:** The notation (*rep*) may be used to specify the identification of the report for which the FORMAT statement is applicable. A value in the range 0 - 31 or a logical name which has been assigned using the DEFINE PRINTER statement may be specified.

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td></td><td>If <code>(rep)</code> is not specified, the <code>FORMAT</code> statement will be applicable to the first report (Report 0).

For information on how to control the format of an output report created with Natural, see Report Format and Control (in the Programming Guide).</td></tr>
<tr><td>parameter</td><td>Parameter(s):

The parameters can be specified in any order and must be separated by one or more spaces. A single entry must not be split between two statement lines.

Field sensitive parameter settings applied here will only be regarded for variable fields used in an <code>INPUT</code>, <code>WRITE</code>, <code>DISPLAY</code> or <code>PRINT</code> statement of the selected report. They do not apply for text-constants used in any of the mentioned statements.

Example:
<pre>
DEFINE DATA LOCAL
1 VARI (A4)   INIT <'1234'>   /*   Output
END-DEFINE   /*   Produced
FORMAT AD=U   /*   ---------
WRITE 'Text'   VARI   /*   Text 1234
WRITE 'Text' (AD=U)   VARI   /*   Text 1234
END
</pre>
See also Applicable Parameters below.</td></tr>
</table>

Applicable Parameters

See the *Parameter Reference* for a detailed description of the session parameters which may be used.

Parameter	Description
AD	Attribute Definition
AL	Alphanumeric Length for Output
CD	Color Definition
DF	Date Format
DL	Display Length for Output
EM	Edit Mask
ES	Empty Line Suppression
FC	Filler Character
FL	Floating Point Mantissa Length
GC	Filler Character for Group Heading

Parameter	Description
HC	Header Centering
HW	Heading Width
IC	Insertion Character
ICU	Unicode Insertion Character
IP	Input Prompting Text
IS	Identical Suppress
KD	Key Definition
LC	Leading Characters
LCU	Unicode Leading Characters
LS	Line Size
MC	Multiple-Value Field Count (Can only be used in reporting mode.)
MP	Maximum Number of Pages of a Report, see Note below.
MS	Manual Skip
NL	Numeric Length for Output
PC	Periodic Group Count (Can only be used in reporting mode.)
PM	Print Mode
PS	Page Size, see Note below.
SF	Spacing Factor
SG	Sign Position
TC	Trailing Characters
TCU	Unicode Trailing Characters
UC	Underlining Character
ZP	Zero Printing

Note: The parameters `MP` and `PS` do not take effect for a specific I/O statement, but apply to the complete output created for the report. If multiple settings for `MP` and `PS` are performed, the last definition is used.

See also *Underlining Character for Titles and Headers - UC Parameter* (in the *Programming Guide*).

Example

```
** Example 'FMTEX1': FORMAT
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 POST-CODE
  2 COUNTRY
END-DEFINE
*
FORMAT AL=7    /* Alpha-numeric field output length
       FC=+    /* Filler character for field header
       GC=*    /* Filler character for group header
       HC=L    /* Header left justified
       IC=<<   /* Insert characters
       IS=ON   /* Identical suppress on
       TC=>>   /* Trailing character
       UC==    /* Underline character
       ZP=OFF  /* Zero print off
*
LIMIT 5
READ EMPLOY-VIEW BY NAME
  DISPLAY NOTITLE
          NAME 3X CITY 3X POST-CODE 3X COUNTRY
END-READ
*
END
```

Output of Program FMTEX1:

```
NAME+++++++   CITY+++++++   POSTAL+++++   COUNTRY++++
                              ADDRESS++++
===========   ===========   ===========   ===========

<<ABELLAN>>   <<MADRID >>   <<28014  >>   <<E  >>
<<ACHIESO>>   <<DERBY  >>   <<DE3 4TR>>   <<UK >>
<<ADAM   >>   <<JOIGNY >>   <<89300  >>   <<F  >>
<<ADKINSO>>   <<BROOKLY>>   <<11201  >>   <<USA>>
              <<BEVERLE>>   <<90211  >>
```

75 GET

In structured mode and in reporting mode using a DEFINE DATA LOCAL statement, the following syntax applies:

```
GET [IN] [FILE] view-name
[PASSWORD=operand1]
[CIPHER=operand2]
[ RECORDS ] { operand3    }
            { *ISN [(r)]  }
```

In reporting mode using no DEFINE DATA LOCAL statement, the following syntax applies:

```
GET [IN] [FILE] ddm-name
[PASSWORD=operand1]
[CIPHER=operand2]
[ RECORDS ] { operand3    }  operand4 ...
            { *ISN [(r)]  }
```

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

Related Statements: ACCEPT/REJECT | AT BREAK | AT START OF DATA | AT END OF DATA | BACKOUT TRANSACTION | BEFORE BREAK PROCESSING | DELETE | END TRANSACTION | FIND | GET SAME | GET TRANSACTION | HISTOGRAM | LIMIT | PASSW | PERFORM BREAK PROCESSING | READ | RETRY | STORE | UPDATE

Belongs to Function Group: ***Database Access and Update***

Function

The GET statement is used to read a record with a given Adabas Internal Sequence Number (ISN).

For XML databases, the GET statement is used to read an XML object with a given object ID.

The GET statement does not cause a processing loop to be initiated.

Restrictions

- The `GET` statement cannot be used for SQL databases.
- The `GET` statement cannot be used with Entire System Server.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
operand1	C	S				A												yes	no
operand2	C	S					N											no	no
operand3	C	S			N		N	P	I		B *							yes	no
operand4		S	A			A	N	P	I	F	B	D	T	L				yes	yes

* Format B of *operand3* may be used only with a length of less than or equal to 4.

Syntax Element Description:

Syntax Element	Description
view-name	**View Name:** In structured mode and in reporting mode using a `DEFINE DATA LOCAL` statement, the name of a view as defined either directly within a `DEFINE DATA` statement or in a separate global or local data area.
ddm-name	**DDM Name:** In reporting mode using no `DEFINE DATA LOCAL` statement, the name of the data definition module (DDM) is referenced.
`PASSWORD=`*operand1* `CIPHER=`*operand2*	**PASSWORD Clause/CIPHER Clause:** These clauses are applicable only to Adabas databases. The `PASSWORD` clause is used to provide a password when retrieving data from an Adabas file which is password protected. The `CIPHER` clause is used to provide a cipher key when retrieving data from an Adabas file which is enciphered. See the statements `FIND` and `PASSW` for further information.
`*ISN / `*operand3*	**Internal Sequence Number:**

Syntax Element	Description
	The ISN must be provided either in the form of a numeric constant or user-defined variable (*operand3* in the range from 1 to 4294967295), or via the Natural system variable *ISN.
(r)	**Statement Reference:** The notation (r) is used to specify the statement which contains the FIND or READ statement used to initially read the record. If (r) is not specified, the GET statement will be related to the innermost active processing loop. (r) may be specified as a reference statement number or as a statement label.
operand4	**Reference to Database Fields:** In reporting mode, subsequent references to database fields that have been read with a GET statement can contain the label or line number of the GET statement.

Example

```
** Example 'GETEX1': GET
************************************************************************
DEFINE DATA LOCAL
1 PERSONS VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
1 SALARY-INFO VIEW OF EMPLOYEES
  2 NAME
  2 CURR-CODE (1:1)
  2 SALARY    (1:1)
*
1 #ISN-ARRAY  (B4/1:10)
1 #LINE-NR    (N2)
END-DEFINE
*
FORMAT PS=16
LIMIT 10
READ PERSONS BY NAME
  MOVE *COUNTER TO #LINE-NR
  MOVE *ISN     TO #ISN-ARRAY (#LINE-NR)
  DISPLAY #LINE-NR PERSONNEL-ID NAME FIRST-NAME
  /*
  AT END OF PAGE
    INPUT / 'PLEASE SELECT LINE-NR FOR SALARY INFORMATION:' #LINE-NR
    IF #LINE-NR = 1 THRU 10
      GET SALARY-INFO #ISN-ARRAY (#LINE-NR)
      WRITE / SALARY-INFO.NAME
```

```
              SALARY-INFO.SALARY    (1)
              SALARY-INFO.CURR-CODE (1)
    END-IF
  END-ENDPAGE
  /*
END-READ
END
```

Output of Program GETEX1:

```
Page      1                                                05-01-13  13:17:42

#LINE-NR PERSONNEL         NAME              FIRST-NAME
            ID
-------- --------- -------------------- --------------------

  1      60008339  ABELLAN              KEPA
  2      30000231  ACHIESON             ROBERT
  3      50005800  ADAM                 SIMONE
  4      20008800  ADKINSON             JEFF
  5      20009800  ADKINSON             PHYLLIS
  6      20012700  ADKINSON             HAZEL
  7      20013800  ADKINSON             DAVID
  8      20019600  ADKINSON             CHARLIE
  9      20008600  ADKINSON             MARTHA
 10      20005700  ADKINSON             TIMMIE

PLEASE SELECT LINE-NR FOR SALARY INFORMATION: 1

ABELLAN                 1450000 PTA
```

76 GET SAME

Structured Mode Syntax

```
GET SAME [(r)]
```

Reporting Mode Syntax

```
GET SAME [(r)] [operand1 ...]
```

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

Related Statements: ACCEPT/REJECT | AT BREAK | AT START OF DATA | AT END OF DATA | BACKOUT TRANSACTION | BEFORE BREAK PROCESSING | DELETE | END TRANSACTION | FIND | GET | GET TRANSACTION DATA | HISTOGRAM | LIMIT | PASSW | PERFORM BREAK PROCESSING | READ | RETRY | STORE | UPDATE

Belongs to Function Group: ***Database Access and Update***

Function

The GET SAME statement is used to re-read the record currently being processed. It is most frequently used to obtain database array values (periodic groups or multiple-value fields) if the number and range of existing or desired occurrences was not known when the record was initially read.

Restrictions

- GET SAME is only valid for Natural users who are using Adabas.
- GET SAME cannot be used with Entire System Server.
- An UPDATE or DELETE statement must not reference a GET SAME statement. These statements should instead make reference to the FIND, READ or GET statement used to read the record initially.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1		S	A			A	U	N	P			B							no	yes

Syntax Element Description:

Syntax Element	Description
(*r*)	**Statement Reference:** The notation (*r*) is used to specify the statement which contains the `FIND` or `READ` statement used to initially read the record. If (*r*) is not specified, the `GET SAME` statement will be related to the innermost active processing loop. (*r*) may be specified as a reference statement number or as a statement label.
operand1	**Fields to Be Made Available:** As *operand1*, you specify the field(s) to be made available as a result of the `GET SAME` statement. **Note:** *operand1* cannot be specified if the field is defined in a `DEFINE DATA` statement.

Example

```
** Example 'GSAEX1': GET SAME
************************************************************************
DEFINE DATA LOCAL
1 I                 (P3)
1 POST-ADDRESS VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 ADDRESS-LINE    (I:I)
  2 C*ADDRESS-LINE
  2 POST-CODE
  2 CITY
*
1 #NAME             (A30)
END-DEFINE
*
FORMAT PS=20
MOVE 1 TO I
*
READ (10) POST-ADDRESS BY NAME
  COMPRESS NAME FIRST-NAME INTO #NAME WITH DELIMITER ','
  WRITE // 12T #NAME
  WRITE /  12T ADDRESS-LINE (I.1)
  /*
```

```
  IF C*ADDRESS-LINE > 1
    FOR I = 2 TO C*ADDRESS-LINE
      GET SAME                      /* READ NEXT OCCURRENCE
      WRITE 12T ADDRESS-LINE (I.1)
    END-FOR
  END-IF
  WRITE / POST-CODE CITY
  SKIP 3
END-READ
END
```

Output of Program GSAEX1:

```
Page      1                                          05-01-13  13:23:36



           ABELLAN,KEPA

           CASTELAN 23-C

28014      MADRID





           ACHIESON,ROBERT

           144 ALLESTREE LANE
           DERBY
           DERBYSHIRE

DE3 4TR    DERBY
```

77 GET TRANSACTION DATA

```
GET TRANSACTION [DATA] operand1 ...
```

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

Related Statements: `ACCEPT/REJECT` | `AT BREAK` | `AT START OF DATA` | `AT END OF DATA` | `BACKOUT TRANSACTION` | `BEFORE BREAK PROCESSING` | `DELETE` | `END TRANSACTION` | `FIND` | `GET` | `GET SAME` | `HISTOGRAM` | `LIMIT` | `PASSW` | `PERFORM BREAK PROCESSING` | `READ` | `RETRY` | `STORE` | `UPDATE`

Belongs to Function Group: ***Database Access and Update***

Function

The `GET TRANSACTION DATA` statement is used to read the data saved with a previous `END TRANSACTION` statement.

`GET TRANSACTION DATA` does not create a processing loop.

System Variable *ETID

The content of the Natural system variable `*ETID` identifies the transaction data to be retrieved from the database.

No Transaction Data Stored

If the `GET TRANSACTION DATA` statement is issued and no transaction data are found, all fields specified in the `GET TRANSACTION DATA` statement will be filled with blanks regardless of format definition.

Caution: Make sure that arithmetic operations are not performed on "empty" transaction data, because this would result in an abnormal termination of the program.

Restriction

The `GET TRANSACTION DATA` statement is only valid for transactions applied to Adabas databases.

Syntax Description

Operand Definition Table:

Operand	Possible Structure				Possible Formats													Referencing Permitted	Dynamic Definition
operand1	S				A	U	N	P	I	F	B	D	T					yes	yes

Syntax Element Description:

Syntax Element	Description
operand1	**Field Specification:** The sequence, lengths, and formats of the fields used in the `GET TRANSACTION DATA` statement must be identical to the sequence, lengths, and formats of the fields specified with the corresponding `END TRANSACTION` statement. **Note:** `GET TRANSACTION DATA` cannot be used if *operand1* is a dynamic variable.

Example

```
** Example 'GTREX1': GET TRANSACTION
**
** CAUTION: Executing this example will modify the database records!
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
  2 MIDDLE-I
  2 CITY
*
1 #PERS-NR (A8) INIT <' '>
END-DEFINE
*
GET TRANSACTION DATA #PERS-NR
IF #PERS-NR NE ' '
  WRITE 'LAST TRANSACTION PROCESSED FROM PREVIOUS SESSION' #PERS-NR
END-IF
*
REPEAT
  /*
  INPUT  10X 'ENTER PERSONNEL NUMBER TO BE UPDATED:' #PERS-NR
  IF #PERS-NR = ' '
```

```
    STOP
  END-IF
  /*
  FIND EMPLOY-VIEW WITH PERSONNEL-ID = #PERS-NR
    IF NO RECORDS FOUND
      REINPUT 'NO RECORD FOUND'
    END-NOREC
    INPUT (AD=M) PERSONNEL-ID (AD=O)
               / NAME
               / FIRST-NAME
               / CITY
    UPDATE
    END TRANSACTION #PERS-NR
  END-FIND
  /*
END-REPEAT
END
```

78 HISTOGRAM

```
HISTOGRAM  [ ALL        ]   [MULTI-FETCH-clause] [multi-fetch-factor]
           [ (operand1) ]   [IN] [FILE] view-name
 [PASSWORD=operand2]
        [       { ASCENDING          }                  ]
        [ [IN]  { DESCENDING         }     [SEQUENCE]   ]
        [       { VARIABLE operand3  }                  ]
        [       { DYNAMIC operand3   }                  ]
 [VALUE] [FOR] [FIELD] operand4
 [STARTING/ENDING-clause]
 [WHERE logical-condition]
  statement ...
END-HISTOGRAM                (structured mode only)
LOOP                         (reporting mode only)
```

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

Related Statements: ACCEPT/REJECT | AT BREAK | AT START OF DATA | AT END OF DATA | BACKOUT TRANSACTION | BEFORE BREAK PROCESSING | DELETE | END TRANSACTION | FIND | GET | GET SAME | GET TRANSACTION DATA | LIMIT | PASSW | PERFORM BREAK PROCESSING | READ | RETRY | STORE | UPDATE

Belongs to Function Group: ***Database Access and Update***

Function

The HISTOGRAM statement is used to read the values of a database field which is defined as a descriptor, subdescriptor, or a superdescriptor. The values are read directly from the Adabas inverted lists. The HISTOGRAM statement causes a processing loop to be initiated but does not provide access to any database fields other than the field specified in the HISTOGRAM statement.

See also the following sections in the *Programming Guide*:

- *HISTOGRAM Statement*
- *Loop Processing*
- *Referencing of Database Fields Using (r) Notation*

Note: For SQL databases: HISTOGRAM returns the number of rows which have the same value in a specific column.

Restrictions

- This statement cannot be used with XML databases.
- This statement cannot be used with Entire System Server.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S						N	P	I		B *							yes	no
operand2	C	S				A													yes	no
operand3		S				A													yes	no
operand4		S				A		N	P	I	F	B	D	T	L				no	no

* Format B of *operand1* may be used only with a length of less than or equal to 4.

Syntax Element Description:

Syntax Element	Description
operand1 / ALL	**Number of Descriptor Values:** You can limit the number of descriptor values to be processed with the HISTOGRAM statement by specifying *operand1* - either as a numeric constant (0 - 4294967295) or as a user-defined variable (containing an integer value). ALL may optionally be specified to emphasize that all descriptor values are to be processed. For this statement, the specified limit has priority over a limit set with a LIMIT statement. If a smaller limit is set with the LT parameter (Limit for Processing Loops), the LT limit applies. **Note:** If you wish to process a 4-digit number of descriptor values, specify it with a leading zero (0*nnnn*); because Natural interprets every 4-digit number enclosed in parentheses as a line-number reference to a statement. *operand1* is evaluated when the HISTOGRAM loop is entered. If the value of *operand1* is modified within the HISTOGRAM loop, this does not affect the number of values read.

Syntax Element	Description
MULTI-FETCH-clause	**MULTI-FETCH Clause:** See ***MULTI-FETCH Clause*** below.
view-name	**View Name:** As *view-name*, you specify the name of a view, which is defined either within a `DEFINE DATA` statement or in a separate global or local data area. The view must not contain any other fields apart from the field used in the `HISTOGRAM` statement (*operand4*). If the field in the view is a periodic-group field or multiple-value field that is defined with an index range, only the first occurrence of that range is filled by the `HISTOGRAM` statement; all other occurrences are not affected by the execution of the `HISTOGRAM` statement. In reporting mode, *view-name* is the name of a DDM if no `DEFINE DATA LOCAL` statement is used.
`PASSWORD=`*operand2*	**PASSWORD Clause:** The `PASSWORD` clause is used to provide a password (*operand2*) when retrieving data from an Adabas file which is password-protected. See the statements `FIND` and `PASSW` for further information.

`SEQUENCE`	**SEQUENCE Clause:**
	
	This clause can only be used for Adabas and SQL databases.
	
	With this clause, you can determine whether the records are to be read in ascending sequence or in descending sequence.
	
	■ The default sequence is ascending (which may, but need not, be explicitly specified by using the keyword `ASCENDING`).
	■ If the records are to be read in descending sequence, you specify the keyword `DESCENDING`.
	■ If, instead of determining it in advance, you want to have the option of determining at runtime whether the records are to be read in ascending or descending sequence, you either specify the keyword `VARIABLE` or `DYNAMIC`, followed by a variable (*operand3*). *operand3* has to be of format/length A1 and can contain the value `A` (for "ascending") or `D` (for "descending").
	  ■ If keyword `VARIABLE` is used, the reading direction (value of *operand3*) is evaluated at start of the `HISTOGRAM` processing loop and remains same until the loop is terminated, regardless if the *operand3* field is altered in the `HISTOGRAM` loop or not.
	  ■ If keyword `DYNAMIC` is used, the reading direction (value of *operand3*) is evaluated before every record fetch in the `HISTOGRAM` processing loop and may be changed from record to record. This allows to change the

Syntax Element	Description
	scroll sequence from ascending to descending (and vice versa) at any place in the HISTOGRAM loop. Examples of SEQUENCE clause: - *Example 2 - HISTOGRAM Statement with Records Read in Descending Sequence* - *Example 3 - HISTOGRAM Statement Using Variable Sequence*
operand4	**Descriptor:** As *operand4*, a descriptor, subdescriptor, superdescriptor or hyperdescriptor may be specified. A descriptor contained within a periodic group may be specified with or without an index. If no index is specified, the descriptor will be selected if the value specified is located in any occurrence. If an index is specified, the descriptor will be selected only if the value is located in the occurrence specified by the index. The index specified must be a constant. An index range must not be used. For a descriptor which is a multiple-value field an index must not be specified; the descriptor will be selected if the value is located in the record regardless of the position of the value.
STARTING-ENDING-clause	**STARTING/ENDING Clause:** Starting and ending values may be specified using the keywords STARTING and ENDING (or THRU) followed by a constant or a user-defined variable representing the value with which processing is to begin/end. For further information, see *Specifying Starting/Ending Values* below.
WHERE *logical-condition*	**WHERE Clause:** The WHERE clause may be used to specify an additional selection criteria (*logical-condition*) which is evaluated *after* a value has been read and *before* any processing is performed on the value (including the AT BREAK evaluation). The descriptor specified in the WHERE clause must be the same descriptor referenced in the HISTOGRAM statement. No other fields from the selected file are available for processing with a HISTOGRAM statement. The syntax for a *logical-condition* is described in the section *Logical Condition Criteria* (in the *Programming Guide*).
END-HISTOGRAM	**End of HISTOGRAM Statement:** In structured mode, the Natural reserved word END-HISTOGRAM must be used to end the HISTOGRAM statement.
LOOP	

Syntax Element	Description
	In reporting mode, the Natural statement LOOP must be used to end the HISTOGRAM statement.

MULTI-FETCH Clause

Note: This clause can only be used for Adabas databases.

```
MULTI-FETCH { ON
            { OFF
            { [OF] multi-fetch-factor }
```

Note: [MULTI-FETCH OF *multi-fetch-factor*] is supported for database types ADA/ADA2. The default processing mode is applied; see profile parameter MFSET. The MULTI-FETCH clause is ignored in case Adabas LA or large objects fields are used or a view size greater than 64KB is defined.

For more information, see the section *MULTI-FETCH Clause* (Adabas) in the *Programming Guide*.

Specifying Starting/Ending Values

Starting and ending values may be specified using the keywords STARTING and ENDING (or THRU) followed by a constant or a user-defined variable representing the value with which processing is to begin/end.

If a starting value is specified and the value is not present, the next higher value is used as the starting value. If no higher value is present, the HISTOGRAM loop will not be entered.

If an ending value is specified, values will be read up to and including the ending value.

Hexadecimal constants may be specified as a starting or ending value for descriptors of format A or B.

Syntax Option 1:

```
[ [STARTING] [ WITH ] [VALUES] operand5 ] [ [ THRU      ] operand6 ]
             [ FROM ]                        [ ENDING AT ]
```

Syntax Option 2:

```
[STARTING] [ WITH ] [VALUES] operand5  TO  operand6
           [ FROM ]
```

Syntax Option 3:

```
{ <             }
{ LT            }
{ LESS THAN     }
{ >             }
{ GT            }
{ GREATER THAN  }
{ <=            }  operand5
{ LE            }
{ LESS EQUAL    }
{ >=            }
{ GE            }
{ GREATER EQUAL }
```

Note: If the comparators of Diagram 3 are used, the options `ENDING AT`, `THRU` and `TO` may not be used. These comparators are also valid for the `READ` statement.

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand5	C	S				A	U	N	P	I	F	B	D	T	L				yes	no
operand6	C	S				A	U	N	P	I	F	B	D	T	L				yes	no

Syntax Element Description:

Syntax Element	Description
`STARTING FROM ... ENDING AT |TO`	**STARTING FROM / ENDING AT Clauses:** The `STARTING FROM` and `ENDING AT` clauses are used to limit reading to a user-specified range of values. The `STARTING FROM` clause (= or `EQ` or `EQUAL TO` or `[STARTING] FROM`) determines the starting value for the read operation. If a starting value is specified, reading will begin with the value specified. If the starting value does not exist, the next higher (or lower for a `DESCENDING` read) value will be returned. If no higher (or lower for `DESCENDING`) value exists, the `HISTOGRAM` loop will not be entered.

Syntax Element	Description
	In order to limit the values to an end-value, you may specify an ENDING AT clause with the terms THRU, ENDING AT or TO, that imply an inclusive range. Whenever the descriptor field exceeds the end-value specified, an automatic loop termination is performed. Although the basic functionality of the TO, THRU and ENDING AT keywords looks quite similar, internally they differ in how they work.
THRU | ENDING AT	**THRU / ENDING AT Option:** If THRU or ENDING AT is used, only the start-value is supplied to the database, but the end-value check is performed by the Natural runtime system, after the value is returned by the database. The THRU and ENDING AT options can be used for all databases which support the HISTOGRAM statements.
TO	**Range:** If the keyword TO is used, both the start-value and the end-value are sent to the database and Natural does not perform checks for value ranges. If the end-value is exceeded, the database reacts in the same way as when "end-of-file" is reached and the database loop is exited. Since the complete range checking is done by the database, the lower-value (of the range) is always supplied in the start-value and the higher-value filled into the end-value, regardless whether you are browsing in ASCENDING or in DESCENDING order.

Note: The result of READ/HISTOGRAM THRU/ENDING AT might differ from the result of READ/HISTOGRAM TO if Natural and the accessed database reside on different platforms with different collating sequences.

System Variables Available with HISTOGRAM

The Natural system variables *ISN, *NUMBER, and *COUNTER are available with the HISTOGRAM statement.

*NUMBER and *ISN are only set after the evaluation of the WHERE clause. They must not be used in the logical condition of the WHERE clause.

System Variable	Explanation
*NUMBER	The system variable *NUMBER contains the number of database records that contain the last value read. For SQL databases, see **NUMBER for SQL Databases* in the *System Variables* documentation.
*ISN	The system variable *ISN contains the number of the occurrence in which the descriptor value last read is contained. *ISN will contain 0 if the descriptor is not contained within a periodic group. *ISN is not available for SQL databases.

System Variable	Explanation
*COUNTER	The system variable *COUNTER contains a count of the total number of values which have been read (after evaluation of the WHERE clause).

Examples

- Example 1 - HISTOGRAM Statement
- Example 2 - HISTOGRAM Statement with Records Read in Descending Sequence
- Example 3 - HISTOGRAM Statement Using Variable Sequence

Example 1 - HISTOGRAM Statement

```
** Example 'HSTEX1S': HISTOGRAM (structured mode)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
END-DEFINE
*
LIMIT 8
HISTOGRAM EMPLOY-VIEW CITY STARTING FROM 'M'
  DISPLAY NOTITLE
          CITY 'NUMBER OF/PERSONS' *NUMBER *COUNTER
END-HISTOGRAM
*
END
```

Output of Program HSTEX1S:

```
        CITY          NUMBER OF      CNT
                       PERSONS
-------------------- ----------- -----------


MADISON                        3           1
MADRID                        41           2
MAILLY LE CAMP                 1           3
MAMERS                         1           4
MANSFIELD                      4           5
MARSEILLE                      2           6
MATLOCK                        1           7
MELBOURNE                      2           8
```

Equivalent reporting-mode example: HSTEX1R.

Example 2 - HISTOGRAM Statement with Records Read in Descending Sequence

```
** Example 'HSTDSCND': HISTOGRAM (with DESCENDING)
************************************************************************
DEFINE DATA LOCAL
1 EMPL VIEW OF EMPLOYEES
  2 NAME
END-DEFINE
*
HISTOGRAM (10) EMPL IN DESCENDING SEQUENCE FOR NAME FROM 'ZZZ'
  DISPLAY NAME *NUMBER
END-HISTOGRAM
END
```

Output of Program HSTDSCND:

```
Page      1                                                  05-01-13  13:41:03

        NAME             NMBR
-------------------- -----------

ZINN                           1
YOT                            1
YNCLAN                         1
YATES                          1
YALCIN                         1
YACKX-COLTEAU                  1
XOLIN                          1
WYLLIS                         2
WULFRING                       1
WRIGHT                         1
```

Example 3 - HISTOGRAM Statement Using Variable Sequence

```
** Example 'HSTVSEQ': HISTOGRAM (with VARIABLE SEQUENCE)
************************************************************************
DEFINE DATA LOCAL
1 EMPL VIEW OF EMPLOYEES
  2 NAME
*
1 #DIR      (A1)
1 #STARTVAL (A20)
END-DEFINE
*
SET KEY PF3 PF7 PF8
*
MOVE 'ADKINSON' TO #STARTVAL
*
HISTOGRAM (9) EMPL FOR NAME FROM #STARTVAL
  WRITE NAME *NUMBER
```

```
  IF *COUNTER = 5
    MOVE NAME TO #STARTVAL
  END-IF
END-HISTOGRAM
*
#DIR := 'A'
*
REPEAT
  HISTOGRAM EMPL IN VARIABLE #DIR SEQUENCE
            FOR NAME FROM #STARTVAL
    MOVE NAME TO #STARTVAL
    INPUT NO ERASE  (IP=OFF AD=O)
          15/01 NAME *NUMBER
          //   'Direction:' #DIR
          //   'Press PF3 to stop'
          /    '      PF7 to go step back'
          /    '      PF8 to go step forward'
          /    '      ENTER to continue in that direction'
    /*
    IF *PF-KEY = 'PF7' AND #DIR = 'A'
      MOVE 'D' TO #DIR
      ESCAPE BOTTOM
    END-IF
    IF *PF-KEY = 'PF8' AND #DIR = 'D'
      MOVE 'A' TO #DIR
      ESCAPE BOTTOM
    END-IF
    IF *PF-KEY = 'PF3'
      STOP
    END-IF
  END-HISTOGRAM
  /*
  IF *COUNTER(0250) = 0
    STOP
  END-IF
END-REPEAT
END
```

Output of Program HSTVSEQ:

```
Page      1                                                     05-01-13  13:50:31

ADKINSON                       8
AECKERLE                       1
AFANASSIEV                     2
AHL                            1
AKROYD                         1
ALEMAN                         1
ALESTIA                        1
ALEXANDER                      5
ALLEGRE                        1
```

```
MORE
```

After pressing ENTER:

```
Page      1                                                        05-01-13  13:50:31

ADKINSON                       8
AECKERLE                       1
AFANASSIEV                     2
AHL                            1
AKROYD                         1
ALEMAN                         1
ALESTIA                        1
ALEXANDER                      5
ALLEGRE                        1



AKROYD                         1

Direction: A

Press PF3 to stop
      PF7 to go step back
      PF8 to go step forward
      ENTER to continue in that direction
```

79 IF

Structured Mode Syntax

```
IF logical-condition
 [THEN] statement ...
 [ELSE statement ... ]
END-IF
```

Reporting Mode Syntax

```
IF logical-condition
 [THEN]   {  statement                   }
          {  DO statement ... DOEND      }
[   ELSE  {  statement                   }  ]
[         {  DO statement ... DOEND      }  ]
```

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

Related Statements: DECIDE FOR | DECIDE ON | IF SELECTION | ON ERROR

Belongs to Function Group: ***Processing of Logical Conditions***

Function

The IF statement is used to control execution of a statement or group of statements based on a logical condition.

Note: If no action is to be performed in case the condition is met, you must specify the statement IGNORE in the THEN clause.

Syntax Description

Syntax Element	Description
IF *logical-condition*	**Logical Condition Criterion:** The logical condition which is used to determine whether the statement or statements specified with the IF statement are to be executed. Examples:

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td></td><td><pre>IF #A = #B
IF LEAVE-TAKEN GT 30
IF #SALARY(1) * 1.15 GT 5000
IF SALARY (4) = 5000 THRU 6000
IF DEPT = 'A10' OR = 'A20' OR = 'A30'</pre>For further information, see the section Logical Condition Criteria (in the Programming Guide).</td></tr>
<tr><td><code>THEN statement</code></td><td>THEN Clause:

In the <code>THEN</code> clause, you specify the <code>statement</code>(s) to be executed if the logical condition is true.</td></tr>
<tr><td><code>ELSE statement</code></td><td>ELSE Clause:

In the <code>ELSE</code> clause, you specify the <code>statement</code>(s) to be executed if the logical condition is not true.</td></tr>
<tr><td><code>END-IF</code></td><td rowspan="2">END of IF Statement:

In structured mode, the Natural reserved word <code>END-IF</code> must be used to end the <code>IF</code> statement.

In reporting mode, use the <code>DO ... DOEND</code> statements to supply one or several suitable statements, depending on the situation, and to end the clauses and the <code>IF</code> statement. If you specify only a single statement, you can omit the <code>DO ... DOEND</code> statements. With respect to good coding practice, this is not recommended.</td></tr>
<tr><td><code>statement
DO statement ... DOEND</code></td></tr>
</table>

Example

```
** Example 'IFEX1S': IF (structured mode)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
  2 SALARY (1)
  2 BIRTH
1 VEHIC-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
*
1 #BIRTH (D)
END-DEFINE
*
MOVE EDITED '19450101' TO #BIRTH (EM=YYYYMMDD)
```

```
SUSPEND IDENTICAL SUPPRESS
LIMIT 20
*
FND. FIND EMPLOY-VIEW WITH CITY = 'FRANKFURT'
          SORTED BY NAME BIRTH
  IF SALARY (1) LT 40000
    WRITE NOTITLE '*****' NAME 30X 'SALARY LT 40000'
  ELSE
    IF BIRTH GT #BIRTH
      FIND VEHIC-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (FND.)
        DISPLAY (IS=ON)
                NAME BIRTH (EM=YYYY-MM-DD)
                SALARY (1) MAKE (AL=8)
      END-FIND
    END-IF
  END-IF
END-FIND
END
```

Output of Program IFEX1S:

```
        NAME            DATE       ANNUAL     MAKE
                         OF        SALARY
                        BIRTH
-------------------- ---------- ---------- --------

BAECKER              1956-01-05      74400 BMW
***** BECKER                                             SALARY LT 40000
BLOEMER              1979-11-07      45200 FIAT
FALTER               1954-05-23      70800 FORD
***** FALTER                                             SALARY LT 40000
***** GROTHE                                             SALARY LT 40000
***** HEILBROCK                                          SALARY LT 40000
***** HESCHMANN                                          SALARY LT 40000
HUCH                 1952-09-12      67200 MERCEDES
***** KICKSTEIN                                          SALARY LT 40000
***** KLEENE                                             SALARY LT 40000
***** KRAMER                                             SALARY LT 40000
```

Equivalent reporting-mode example: **IFEX1R**.

80 IF SELECTION

Structured Mode Syntax

```
IF SELECTION [NOT UNIQUE [IN [FIELDS]]] operand1 ...
  [THEN] statement...
  [ELSE statement...]
END-IF
```

Reporting Mode Syntax

```
IF SELECTION [NOT UNIQUE [IN [FIELDS]]] operand1...
          [THEN] { statement                    }
                 { DO statement... DOEND        }
        [ ELSE   { statement                    }   ]
        [        { DO statement... DOEND        }   ]
```

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

Related Statements: DECIDE FOR | DECIDE ON | IF

Belongs to Function Group: ***Processing of Logical Conditions***

Function

The IF SELECTION statement is used to verify that in a sequence of alphanumeric fields one and only one contains a value.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1		S	A			A	U							L	C				yes	no

Syntax Element Description:

Syntax Element	Description
operand1	**Selection Field(s):** As *operand1* you specify the fields which are to be checked. If you specify an attribute control variable (Format C), it is considered to contain a value if its status has been changed to `MODIFIED`. **Note:** To check if a specific attribute control variable has been assigned the status `MODIFIED`, use the `MODIFIED` option of, for example, an `IF` statement. This enables you to check that exactly one field was *modified*.
`THEN` *statement* ...	**THEN Clause:** The statement(s) specified in the `THEN` clause will be executed if one of the following conditions is true: ■ None of the fields specified in *operand1* contains a value. ■ More than one of the fields specified in *operand1* contains a value. This statement is generally used to verify that a terminal user has entered only one function in response to a map displayed via an `INPUT` statement. **Note:** If *no* action is to be performed if one of the conditions is met, you specify the statement `IGNORE` in the `THEN` clause.
`ELSE` *statement* ...	**ELSE Clause:** In the `ELSE` clause, you specify the statement(s) to be executed if exactly one field contains a value.
`END-IF`	**End of IF SELECTION Statement:** In structured mode, the Natural reserved word `END-IF` must be used to end the `IF SELECTION` statement. In reporting mode, use the `DO ... DOEND` statements to supply one or several suitable statements, depending on the situation, and to end the clauses and the `IF SELECTION` statement. If you specify only a single statement, you can omit the `DO ... DOEND` statements. With respect to good coding practice, this is not recommended.
statement ... `DO` *statement* ... `DOEND`	

Example

```
** Example 'IFSEL': IF SELECTION
************************************************************************
DEFINE DATA LOCAL
1 #A (A1)
1 #B (A1)
END-DEFINE
*
INPUT 'Select one function:' //
   9X 'Function A:' #A
   9X 'Function B:' #B
*
IF SELECTION NOT UNIQUE #A #B
  REINPUT 'Please enter one function only.'
END-IF
*
IF #A NE ' '
  WRITE 'Function A selected.'
END-IF
IF #B NE ' '
  WRITE 'Function B selected.'
END-IF
*
END
```

Output of Program IFSEL:

```
Select one function:

         Function A:           Function B:
```

After selecting and confirming function A:

```
Page      1                                               05-01-17  11:04:07

Function A selected.
```

81 IGNORE

```
IGNORE
```

Function

The `IGNORE` statement is an “empty” statement which itself does not perform any function.

During the development phase of an application, you can insert `IGNORE` temporarily within statement blocks in which one or more statements are required, but which you intend to code later (for example, within `AT BREAK` or `AT START OF DATA` / `AT END OF DATA`). This allows you to continue programming in another part of the application without the as yet incomplete statement block leading to an error.

The `IGNORE` statement must also be used in condition statements, such as `IF` or `DECIDE FOR`, if no function is to be performed in the case of a condition being met.

Example

```
...
...
AT TOP OF PAGE
   IGNORE         /* top-of-page processing still to be coded
END-TOPPAGE
...
...
```

82 INCLUDE

```
INCLUDE  copycode-name  [operand1] ... 99
```

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

Function

The `INCLUDE` statement is used to include source lines from an external object of type copycode into another object at compilation.

The `INCLUDE` statement is evaluated at *compilation* time. The source lines of the copycode will not be physically included in the source of the program that contains the `INCLUDE` statement, but they will be included during the program compilation and thus in the resulting object module.

Caution: A source code line which contains an `INCLUDE` statement must not contain any other statement.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C					A	U												no	no

Syntax Element Description:

Syntax Element	Description
copycode-name	**Copycode Name:** As *copycode-name* you specify the name of the copycode whose source is to be included. *copycode-name* may contain an ampersand (&); at compile time, this character will be replaced by the one-character code corresponding to the current value of the Natural system variable `*LANGUAGE`. This feature allows the use of multilingual copycode names. The object you specify must be of the type copycode. The copycode must be contained either in the same library as the program which contains the `INCLUDE` statement or in the respective steplib (the default steplib is `SYSTEM`). When the source of a copycode is modified, all programs using that copycode must be compiled again to reflect the changed source in their object codes. The source code of the copycode must consist of syntactically complete statements.
operand1	**Insert Values for Dynamic Insertion:**

Syntax Element	Description
	You can dynamically insert values in the copycode which is included. These values are specified with *operand1*. In the copycode, the values are referenced with the following notation: `&n&` That is, you mark the position where a value is to be inserted with `&n&`. *n* is the sequential number of each value passed with the `INCLUDE` statement. For example, `&3&` would refer to the third value specified with the statement. For every `&n&` notation in the copycode you must specify a value in the `INCLUDE` statement. For example, if the copycode contains `&5&`, *operand1* must be specified at least five times. You may write one copy code parameter (`&n&`) after another without blanks (that is, `&1&&2&&3&`). This method is used to concatenate multiple copy code parameters to a source. A string may follow one or several copy code parameters without a blank (that is, `&1&abc` or `&1&&2&abc`). This method is used to concatenate a string to multiple copy code parameters. **Note:** Because `&n&` is a valid part of an identifier, this notation may not be used as a copy code parameter substitution in other positions described above (i.e. `abc&1&` or `&1&abc&2&`). In other words, a string may only come after copy code parameters, not before or between. Values that are specified in the `INCLUDE` statement but not referenced in the copycode will be ignored.

Examples

- Example 1 - INCLUDE Statement Including Copycode
- Example 2 - INCLUDE Statement Including Copycode with Parameters
- Example 3 - INCLUDE Statement Using Nested Copycodes

- Example 4 - INCLUDE Statement with Concatenated Parameters in Copycode

Example 1 - INCLUDE Statement Including Copycode

Program containing the INCLUDE statement:

```
** Example 'INCEX1': INCLUDE  (include copycode)
************************************************************************
*
WRITE 'Before copycode'
*
INCLUDE INCEX1C
*
WRITE 'After copycode'
*
END
```

Copycode INCEX1C to be included:

```
** Example 'INCEX1C': INCLUDE (copycode used by INCEX1)
************************************************************************
*
WRITE 'Inside copycode'
```

Output of Program INCEX1:

```
Page      1                                                  05-01-25  16:26:36

Before copycode
Inside copycode
After copycode
```

Example 2 - INCLUDE Statement Including Copycode with Parameters

Program INCEX2 containing the INCLUDE statement:

```
** Example 'INCEX2': INCLUDE  (include copycode with parameters)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
END-DEFINE
*
*
INCLUDE INCEX2C 'EMPL-VIEW' 'NAME' '''ARCHER''' '20' '''BAILLET'''
*
END
```

Copycode INCEX2C to be included:

```
** Example 'INCEX2C': INCLUDE (copycode used by INCEX2)
************************************************************************
* Transferred parameters from INCEX2:
*
* &1& : EMPL-VIEW
* &2& : NAME
* &3& : 'ARCHER'
* &4& : 20
* &5& : 'BAILLET'
*
*
READ (&4&) &1&  BY &2&  = &3&
  DISPLAY &2&
  IF &2& = &5&
    WRITE 5X 'LAST RECORD FOUND' &2&
    STOP
  END-IF
END-READ
*
* Statements above will be completed to:
*
* READ (20) EMPL-VIEW  BY NAME  = 'ARCHER'
*   DISPLAY NAME
*   IF NAME = 'BAILLET'
*     WRITE 5X 'LAST RECORD FOUND' NAME
*     STOP
*   END-IF
* END-READ
```

Output of Program INCEX2:

```
Page      1                                                 05-01-25  16:30:43

        NAME
--------------------

ARCHER
ARCONADA
ARCONADA
ARNOLD
ASTIER
ATHERTON
ATHERTON
ATHERTON
AUBERT
BACHMANN
BAECKER
BAECKER
BAGAZJA
```

```
BAILLET
     LAST RECORD FOUND BAILLET
```

Example 3 - INCLUDE Statement Using Nested Copycodes

Program containing INCLUDE statement:

```
** Example 'INCEX3': INCLUDE  (using nested copycodes)
************************************************************************
DEFINE DATA LOCAL
1 #A (I4)
END-DEFINE
*
MOVE 123 TO #A
WRITE 'Program  INCEX3  ' '=' #A
*
INCLUDE INCEX31C '#A' '5'            /* source line is #A := 5
*
*
MOVE 300 TO #A
WRITE 'Program  INCEX3  ' '=' #A
*
INCLUDE INCEX32C '''#A'''  '''20'''  /* source line is #A := 20
*
WRITE 'Program  INCEX3  ' '=' #A
END
```

Copycode INCEX31C to be included:

```
** Example 'INCEX31C': INCLUDE (copycode used by INCEX3)
************************************************************************
* Transferred parameters from INCEX3:
*
* &1& : #A
* &2& : 5
*
*
&1& := &2&
*
WRITE 'Copycode INCEX31C' '=' &1&
```

Copycode INCEX32C to be included:

```
** Example 'INCEX32C': INCLUDE (copycode used by INCEX3)
************************************************************************
* Transferred parameters from INCEX3:
*
* &1& : '#A'
* &2& : '20'
*
*
WRITE 'Copycode INCEX32C' &1& &2&
*
INCLUDE INCEX31C &1& &2&
```

Output of Program INCEX3:

```
Page      1                                                  05-01-25  16:35:36

Program  INCEX3   #A:          123
Copycode INCEX31C #A:            5
Program  INCEX3   #A:          300
Copycode INCEX32C #A 20
Copycode INCEX31C #A:           20
Program  INCEX3   #A:           20
```

Example 4 - INCLUDE Statement with Concatenated Parameters in Copycode

Program containing INCLUDE statement:

```
** Example 'INCEX4': INCLUDE  (with concatenated parameters in copycode)
************************************************************************
DEFINE DATA LOCAL
1 #GROUP
  2  ABC(A10) INIT <'1234567890'>
END-DEFINE
*
INCLUDE INCEX4C  '#GROUP.'  'ABC'  'AB'
*
END
```

Copycode INCEX4C to be included:

```
** Example 'INCEX4C': INCLUDE (copycode used by INCEX4)
************************************************************************
* Transferred parameters from INCEX4:
*
* &1& : #GROUP.
* &2& : ABC
* &3& : AB
*
*
WRITE  '=' &2&          /* 'ABC'             results into ABC
WRITE  '=' &1&ABC       /* '#GROUP.' ABC     results into #GROUP.ABC
WRITE  '=' &1&&2&       /* '#GROUP.' 'ABC'   results into #GROUP.ABC
WRITE  '=' &1&&3&C      /* '#GROUP.' 'AB' C  results into #GROUP.ABC
```

Output of Program INCEX4:

```
Page      1                                                  05-01-25  16:37:59

ABC: 1234567890
ABC: 1234567890
ABC: 1234567890
ABC: 1234567890
```

X INPUT

The syntax is described separately. See:

- *INPUT Syntax 1 - Dynamic Screen Layout Specification*
- *INPUT Syntax 2 - Using Predefined Map Layout*

Related Statements: `DEFINE WINDOW` | `REINPUT` | `SET WINDOW`

Belongs to Function Group: *Screen Generation for Interactive Processing*

Function

The `INPUT` statement is used in interactive mode to create a formatted screen or map for data entry.

It may also be used in conjunction with the Natural stack (see the `STACK` statement) and to provide user data for programs being executed in batch mode.

For Natural RPC: See *Notes on Natural Statements on the Server* in the *Natural RPC (Remote Procedure Call)* documentation.

Input Modes

The `INPUT` statement may be used in screen, forms, or keyword/delimiter mode. Screen mode is generally used with video terminals/screens. Forms mode may be used with TTY terminals. Delimiter mode is used with TTY terminals, and also in batch mode. The default mode is screen mode.

You can change the input mode with the session parameter `IM`.

Screen Mode

In screen mode, execution of the `INPUT` statement results in the display of a screen according to the fields and positioning notation specified. The message line of the screen is used by Natural for error messages. The position of the message line (top or bottom of screen) may be controlled by the terminal command `%M`. The terminal user may position to specific fields using the various tabulation keys.

As Natural allows for screen window processing, the layout of the logical screen map may be larger (theoretically 250 characters per line and 250 lines, but limited by the internal screen buffer) than the physical screen size.

The windowing terminal command `%W` may be used to modify logical and physical window position and size (see the terminal command `%W` for details of window handling).

For input fields (`AD=A` or `AD=M`) that are not fully displayed on the physical screen, the following rules apply:

- Input fields whose beginning is not inside the window are always made protected.
- Input fields which begin inside and end outside the window are only made protected if the values they contain cannot be displayed completely in the window. Please note that in this case it is decisive whether the value length, not the field length, exceeds the window size. Filler characters (as specified with the profile parameter `FC` or session parameter `AD`) do not count as part of the value.
- Before an input field thus protected can be accessed and processed, the window size must be adjusted so as to fully display the field or value respectively (see the terminal command `%W`).

Non-Screen Modes

The `INPUT` statement may be used for an operation on line-oriented devices or for the processing of batch input from sequential files.

The same map layouts as defined for screen mode operation can also be processed in non-screen mode.

Forms mode and keyword/delimiter mode are also available to process the input either by simulating the screen layout in line mode or by just processing the data without any map layout.

See also:

- *Using the INPUT Statement in Non-Screen Modes*
- *Using the INPUT Statement in Batch Mode*
- *Processing Data from the Natural Stack*

Entering Data in Response to an INPUT Statement

Data for an alphanumeric field must be entered left-justified. Any character, including a blank, is meaningful. The data are assigned one character per byte to the internal field. Data entered for an alphanumeric field are not validated.

Lower and upper case translation are controlled by the terminal commands `%L` and `%U` as well as the attributes `AD=T` and `AD=W`.

Data for a numeric field may be placed anywhere in the input field. Leading and/or trailing blanks, leading zeros, a leading sign and one decimal point are permitted. Natural adjusts the value according to the internal definition of the field. If `SG=OFF` is specified, Natural does not assume or allocate a position for a sign position. Data for a field defined with format P must be entered in decimal form. Natural will convert decimal to packed wherever necessary. A field containing all blanks is interpreted as a zero value. Data for a numeric field are validated by Natural to ensure that the value consists only of leading and/or trailing blanks, an optional leading sign, an optional decimal point, and numeric characters. If no decimal point is entered, it is assumed to be to the right of the value entered.

Data for a binary field must be entered for all positions (two characters per byte). Only valid hexadecimal characters (0 - 9, A - F) may be used. A blank (H'20' in ASCII or H'40' in EBCDIC respectively) is valid and is converted to binary zeros. Data for a binary field are validated by Natural for hexadecimal characters.

Data for format L fields may be entered as blank (false) or non-blank (true).

Data for format F, D, and T are entered according to the rules stated for F, D, and T constants.

Numeric Edit Mask Free Mode

Within a field element, you may format the representation of the field content with an edit mask. The edit mask is used for two purposes:

- to build the layout for displaying the field on the screen;
- when a string has been modified and ENTER has been pressed, to extract the field data from the string entered.

The advantage of improving the format of the field data displayed with additional insert characters may actually be a disadvantage, because a new data value entered has to perfectly match the format of the edit mask.

Example:

```
SET GLOBALS ID=; DC=,
RESET N (N7,3)
INPUT N (AD=M EM=Z'.'ZZZ'.'ZZZ,999EUR)
END
```

Output value	is displayed as:	Input value	must be entered as:	leads to an input error if entered as:
0	,000EUR	1	1,000EUR	1 1EUR 01,000EUR
1234	1.234,000EUR	1234567	1.234.567,000EUR	1234567 1.234.567 1.234.567EUR
0,123	,123EUR	1,234	1,234EUR	1,234

Another option for entering numeric fields with the edit mask is to use an alternative `INPUT` mode, which is called the edit mask free mode. When activated (either at session startup with the profile parameter `EMFM` or in a running Natural session via the terminal command `%FM+`), all or some of the edit mask insert characters may be left out from input.

However, when a contiguous string of insertion characters appears in the edit mask (like `EUR` in the example below), you may only supply or leave out the string completely. The number of optional or mandatory digits (edit-mask character `Z` and `9`) to be supplied is not affected.

Example with Edit Mask Free Mode activated:

```
SET GLOBALS ID=; DC=,
SET CONTROL 'FM+'         /* activate numeric Edit Mask Free Mode
RESET N (N7,3)
INPUT N (AD=M EM=Z'.'ZZZ'.'ZZZ,999EUR)
END
```

Input value	can be entered as:	leads to an error if entered as:
`1`	1 1,0 001 1,00EUR 0.001 1,EUR	`1EUR`
`1234567`	1234567 1.234.567 1234.567 1234567,0 1.234.567,0 1.234.567,EUR	`1.234.567EUR`

Input value	can be entered as:	leads to an error if entered as:
	1.234.567,0EUR 1.234.567,000EUR	
1,234	1,234 1,234EUR 001,234 0.001,234EUR 00001,234EUR	1,234EU

Note: The edit mask free mode applies only for INPUT, but is ignored in a MOVE EDITED statement.

SB - Selection Box

Selection boxes in an INPUT statement are available on mainframe computers only. On Windows, selection boxes may be defined in the map editor only. On Linux, selection boxes cannot be defined and are ignored, if they are imported from a Windows or mainframe environment.

Selection boxes can be attached to input fields. They are a comfortable alternative to help routines attached to fields, since you can code a selection box direct in your program. You do not need an extra program as with help routines.

For more information, see the session parameter SB in the *Parameter Reference*.

Error Correction

If the value entered in an input field does not correspond to the format or edit mask of the field, Natural displays an error message (without terminating the program execution) and positions the cursor in the field in error. The user may then enter a valid value, whereupon processing continues.

Split-Screen Feature

In general, each INPUT statement generates a new page (or terminal screen) of output. Any INPUT statement which is specified within an AT END OF PAGE statement will not produce a new screen. This feature allows for the creation of a split screen where the upper portion of the screen may be used to display multiple lines and the lower portion can be used to create an input map for communication. The profile parameter PS (page size) should be used, either in a SET GLOBALS or FORMAT statement, to set the logical page size to ensure that the input map is built on the same physical screen.

The first `INPUT` line will be placed after the last displayed line. If the `NO ERASE` option is used, the first `INPUT` line will be placed at the top of the page.

System Variables with the INPUT Statement

For information on relevant system variables, see the section *Input/Output Related System Variables* in the *System Variables* documentation.

83 INPUT Syntax 1 - Dynamic Screen Layout Specification

This form of the `INPUT` statement is used to create a layout of an `INPUT` screen, or to create an `INPUT` data layout which is to be read in batch mode from a sequential input file.

```
INPUT  [WINDOW='window-name'] [NO ERASE]
       [(statement-parameters)]
       [WITH-TEXT-option]
       [MARK-option]
       [ALARM-option]
 {                 {  { 'text'  }  [(attributes)]                                 }
 {  [ nX   ]       {  { 'c'(n)  }                                                 }
 {  [ nT   ]       {                                                              }
 {  [ x/y  ] ...   {  [ '_' ] [ *IN    ]                                          }  ...
 {  [ /    ]       {  [ '=' ] [ *OUT   ]   operand1 [(parameter)]                 }
 {                 {          [ *OUTIN ]                                          }
```

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

INPUT Syntax 1 - Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1		S	A	G	N	A	U	N	P	I	F	B	D	T	L		G		yes	yes

Syntax Element Description:

Syntax Element	Description
`INPUT WINDOW='window-name'`	**INPUT WINDOW='*window-name*' Option:** With this option, you indicate that the `INPUT` statement is to be executed for the specified window. The specified window must be defined in a `DEFINE WINDOW` statement; see ***Example 2 - INPUT Statement with DEFINE WINDOW Statement***. The specified window is only active for the duration of that `INPUT` statement, and is automatically deactivated when the `INPUT` statement has been executed. See also the statements `DEFINE WINDOW` and `SET WINDOW`.
`NO ERASE`	**NO ERASE Option:**

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td></td><td>This option causes a screen map of an <code>INPUT</code> statement to be overlaid onto an existing screen without erasing the screen contents.

Screen as used here refers to a logical screen rather than a physical screen.

All unprotected fields that existed on the screen are converted to protected (display only) fields. The old data remain on the screen until the new layout is displayed. If a field from the new screen content partially overlays an existing field, the one character before the new field and the next character in the existing field will be replaced by a blank.</td></tr>
<tr><td>statement-parameters</td><td>Statement Parameter(s):

One or more parameters, enclosed within parentheses, may be specified immediately after the <code>INPUT</code> statement or an element being displayed.

For a list of parameters that can be specified with the <code>INPUT</code> statement, refer to the section Statement Parameters.

Each parameter specified in this manner will override any previous parameter specified in a <code>GLOBALS</code> command, <code>SET GLOBALS</code> or <code>FORMAT</code> statement. If more than one parameter is specified, one or more blanks must be present between each entry. An entry may not be split between two statement lines.

The parameter settings applied here will only be regarded for variable fields, but they have no effect on text-constants. If you would like to set field attributes for a text-constant, they have to be set explicitly for this element.

Example:

<pre>
DEFINE DATA LOCAL
1 VARI (A4)   INIT <'1234'>   /*   Output
END-DEFINE   /*   Produced
*   /*   ---------
INPUT   'Text'   VARI   /*   Text 1234
INPUT (AD=U)   'Text'   VARI   /*   Text <u>1234</u>
INPUT   'Text' (AD=U)   VARI (AD=U)   /*   <u>Text</u> <u>1234</u>
INPUT   'Text' (AD=U)   VARI   /*   <u>Text</u> 1234
END
</pre>
Examples of using parameters at the statement and element level are provided below.</td></tr>
<tr><td>WITH TEXT-option</td><td>WITH TEXT Option:
This option is used to provide text which is to be displayed in the message line; see WITH TEXT Option below.</td></tr>
<tr><td>MARK-option</td><td>MARK Option:

See the section MARK Option below.</td></tr>
<tr><td>ALARM-option</td><td>Alarm Option:</td></tr>
</table>

Syntax Element	Description
	See the section *Alarm Option* below.
Other syntax elements (`nX`, `nT`, `x/y`, `operand1`, etc.)	**Field Positioning, Text Specification, Attribute Assignment:** See the section *Field Positioning, Text Specification, Attribute Assignment* below.

Statement Parameters

Parameters that can be specified with the INPUT statement		Specification (S = at statement level, E = at element level)
`AD`	Attribute Definition	SE
`AL`	Alphanumeric Length for Output	SE
`CD`	Color Definition	SE
`CV`	Control Variable	SE
`DF`	Date Format	SE
`DL`	Display Length for Output	SE
`DY`	Dynamic Attributes	SE
`EM`	Edit Mask	SE
`EMU`	Unicode Edit Mask	E
`FL`	Floating Point Mantissa Length	SE
`HE`	Helproutine	SE
`IP`	Input Prompting Text	SE
`LS`	Line Size	S
`MC`	Multiple-Value Field Count	S
`MS`	Manual Skip	S
`NL`	Numeric Length for Output	SE
`PC`	Periodic Group Count	S
`PM`	Print Mode *	SE
`PS`	Page Size **	S
`SB`	Selection Box	E
`SG`	Sign Position	SE
`ZP`	Zero Printing	SE

* The `PM` session parameter may not be specified for text constants.

** The `PS` session parameter setting is not considered if the number of occurrences of an array exceeds the PS value.

The individual session parameters are described in the *Parameter Reference*.

WITH TEXT Option

```
[WITH] TEXT { * operand1 } [(attributes)][,operand3] ... 7
            {   operand2 }
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
operand1	C	S					N	P	I		B*							yes	yes
operand2	C	S				A												yes	yes
operand3	C	S				A	N	P	I	F	B	D	T	L				yes	yes

* Format B of `operand1` may be used only with a length of less than or equal to 4.

`WITH TEXT` is used to provide text which is to be displayed in the message line. This is usually a message indicating what action should be taken to process the screen or to correct an error.

Syntax Element Description:

Syntax Element	Description
operand1	**Message Text Number:** *operand1* represents the number of a message text that is to be retrieved from a Natural message file. You can retrieve either user-defined messages or Natural system messages: ■ If you specify a positive value of up to four digits (for example: `954`), you will retrieve user-defined messages. ■ If you specify a negative value of up to four digits (for example: `-954`), you will retrieve Natural system messages. See also ***Example 4 - WITH TEXT Options*** in the description of the `REINPUT` statement.
operand2	**Message Text:** *operand2* represents the message to be placed in the message line. See also ***Example 4 - WITH TEXT Options*** in the description of the `REINPUT` statement.
attributes	**Output Attributes:** It is possible to assign various output attributes for *operand1/2*. These attributes and the syntax that may be used are described in the section ***Output Attributes*** below.
operand3	**Dynamic Replacement of Message Text:** *operand3* represents a numeric or text constant or the name of a variable.

Syntax Element	Description
	The values provided are used to replace parts of a message text that are either specified with `operand1` or `operand2`. The notation `:n:` is used within the message text as a reference to `operand3` contents, where `n` represents the `operand3` occurrence (1 - 7). See also ***Example 4 - WITH TEXT Options*** in the description of the `REINPUT` statement. **Note:** Multiple specifications of `operand3` must be separated from each other by a comma. If the comma is used as a decimal character (as defined with the session parameter `DC`) and numeric constants are specified as `operand3`, put blanks before and after the comma so that it cannot be misinterpreted as a decimal character. Alternatively, multiple specifications of `operand3` can be separated by the input delimiter character (as defined with the session parameter `ID`); however, this is not possible in the case of `ID=/` (slash), because the slash has a different meaning in the `INPUT` statement syntax. Leading zeros or trailing blanks will be removed from the field value before it is displayed in a message.

Output Attributes

attributes indicates the output attributes to be used for text display. Attributes can be:

```
{ AD=ad-value ... }
{ CD=cd-value     }
{ PM=pm-value ... } ...

{ ad-value }
{ cd-value } ...
```

Where:

ad-value, *cd-value* and *pm-value* denote the possible values of the corresponding session parameters `AD`, `CD` and `PM` described in the relevant sections of the *Parameter Reference* documentation.

The compiler actually accepts more than one attribute value for an output field. For example, you can specify: `AD=BDI`. In such a case, however, only the last value applies. In the given example, only the value `I` becomes effective and the output field is displayed intensified.

For an alphanumeric/Unicode constant (Natural data format A or U), you can specify *ad-value* and/or *cd-value* without preceding `CD=` or `AD=`, respectively. The single value entered is then checked against all possible `CD` values first. For example: a value of `IRE` will be interpreted as intensified/red but not as intensified/right-justified/mandatory. You cannot combine a single *cd-value* or *ad-value* with a value preceded by `CD=` or `AD=`.

MARK Option

With the `MARK` option, you can cause the cursor to be placed at any non-protected field on screen. In addition, you can specify the position of the cursor within that field. By default, that is, when the `MARK` option is omitted, the cursor is placed at the beginning of the first non-protected field.

```
MARK [POSITION operand4 [IN]] [FIELD]  { operand1   }
                                       { *fieldname }
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
operand4	C	S					N	P	I									yes	yes
operand1	C	S	A				N	P	I									yes	yes

Syntax Element Description:

Syntax Element	Description
operand1	**Field Reference Number:** *operand1* specifies the number of the field where the cursor is to be positioned in. Each field attribute `AD=A` or `AD=M` (that is, non-protected field) specified in an `INPUT` statement is assigned a field reference number, beginning with 1.
**fieldname*	**Field Name for Referencing:** Instead of the field reference number, the field name may be used to position to a field, using the **fieldname* notation.
operand4	**Cursor Position within Referenced Field:** With `MARK POSITION`, you can have the cursor placed at a specific position - as specified with *operand4* - within a field specified with *operand1* or **fieldname*. *operand4* must not contain decimal digits.

Examples:

```
MARK #NUMBER                 /* Field number
MARK 3                       /* Third map field
MARK *#FIELD1                /* Map field
MARK POSITION 3 IN #NUMBER /* Third character in field number
```

See also ***Example 3 - INPUT Statement with MARK POSITION Option*** at the end of this section.

ALARM Option

This option causes the sound alarm feature of the terminal to be activated when the `INPUT` statement is executed. The appropriate hardware must be available to be able to use this feature.

```
[AND] [SOUND] ALARM
```

Default Prompting Text

Unless the session parameter `IP` (input prompting) is set to `IP=OFF`, the field name of the field used in an `INPUT` statement will be displayed preceding the field value (forms mode) or as a prompting keyword to select the field (keyword/delimiter mode). This default field name may be overridden by specifying either a `'`*`text`*`'` element (which replaces the default name) or `'-'` (which suppresses the display of the default field name) immediately preceding the field name.

Field Positioning, Text Specification, Attribute Assignment

Several notations are available for field positioning, attribute assignment, and text creation.

Syntax Element Description:

Syntax Element	Description
*n*X	**Insert Option:** This option causes *n* spaces to be inserted between fields.
*n*T	**Tabulator Option:** This option causes positioning (tabulation) to print position *n*.
x/*y*	**Positioning Option:** Places the next element on line *x*, beginning in column *y*. *y* must not be zero. Backward positioning in the same line is not permitted.
'*text*'	**Write Protection:** Causes *text* to be displayed write protected; see also *Text Notation, Defining a Text to Be Used with a Statement*.
'*c*' (*n*)	**Character Repetition:** Identical to '*text*', except that the character *c* is displayed *n* times. *n* must be `1 - 132`; see also *Text Notation, Defining a Character to Be Displayed n Times before a Field Value*.
attributes	**Display Attributes:** Attributes to be used for display. See ***Attributes*** below.
'-'	**Minus Sign:** When placed before a field, '-' suppresses the generation of a field name as prompting text.

Syntax Element	Description
	Note: Any text string before a field will replace the field name as prompting text.
`'='`	**Equal Sign:** When placed before a field, `'='` results in the display of the field heading followed by the field contents.
`'/'`	**Slash Sign:** When placed between fields or text elements, `'/'` causes positioning to the beginning of the next print line. The contents of fields may be specified for input, output only, and output for modification using the attribute settings `AD=A`, `AD=O`, and `AD=M` respectively. The default is `AD=A`. All fields specified with `AD=A` (input only) or `AD=M` (output for modification) will create unprotected fields on the screen. A value for such a field may be entered by the user. For TTY devices, output for modification fields will occupy twice the size of the field (one for output, one for input) so that a new value may be entered. An input field (with `AD=A` or `AD=M`) specified as non-displayable will always start on a new line on a TTY device. Example: `INPUT #A (AD=A) #B (AD=O) #C (AD=M)` `#A` is an input field which is unprotected, i.e., a value is to be entered for the field. `#B` is a field which is to be displayed write-protected, that is, no value may be entered for the field. `#C` is a field whose current value is to be displayed, and the value may be modified by entering a new value for the field.
`*IN`, `*OUT` and `*OUTIN`	**Field Attribute Definition:** Equivalent to the attributes `AD=A`, `AD=O`, `AD=M` respectively. **Note:** If a non-modifiable system variable is used in an `INPUT` statement, the value will be displayed as an output-only field `AD=O` or `*OUT` attribute.
operand1	**Field(s) to be Used:** *operand1* represents the field to be used. Database fields or user-defined variables may be specified. Natural directly maps the content of each field from the data area to the `INPUT` statement, no move operation is necessary. When the content of a database field is modified as a result of `INPUT` processing, only the value as contained in the data area is modified. Appropriate database `UPDATE` / `STORE` statements must be used to change the content of the database. When the name of a group of database fields is referenced in an `INPUT` statement, all fields belonging to that group will be individually used as input fields.

Syntax Element	Description
	When reference is made to a range of occurrences within an array, all occurrences are individually processed as input fields, but no prompting text will be created for each individual occurrence, only for the first one. On mainframe computers, arrays with ranges that allow to vary the number of occurrences at execution time may not be specified.
parameter(s)	**Parameter(s):** One or more parameters, enclosed within parentheses, may be specified immediately after *operand1* (see table and example below). Each parameter specified will override any previous parameter specified in a `GLOBALS` command, `SET GLOBALS` (in Reporting Mode) or `FORMAT` statement. If more than one parameter is specified, they must be separated by one or more blanks from one another. Each parameter specification must not be split between two statement lines. The parameter settings applied here will only be regarded for variable fields, but they have no effect on text constants. If you would like to set field attributes for a text-constant, they have to be set explicitly for this element. For information on the individual parameters, see the table in the section ***Statement Parameters***. **Note:** The session parameter `EM` will be referenced dynamically in the DDM if an edit mask is defined for a database field. Edit masks may be specified for output and input fields. When an edit mask is defined for an input field, the data for the field must be entered according to the edit mask specification.

Attributes

The following attributes may be used:

```
[        { B }  ]  [        { BL }  ]  [       ] { C } ]
[        { C }  ]  [        { GR }  ]  [       ] { D } ]
[        { D }  ]  [        { NE }  ]  [       ] { I } ]
[ [AD=]  { I }  ]  [ [CD=]  { PI }  ]  [ [PM=] ] { N } ]
[        { N }  ]  [        { RE }  ]  [       ]       ]
[        { U }  ]  [        { TU }  ]  [       ]       ]
[        { V }  ]  [        { YE }  ]  [       ]       ]
            1                  2                   3
```

1. Display attributes; see the session parameter `AD` (in the *Parameter Reference*).
2. Color attributes; see the session parameter `CD` (in the *Parameter Reference*).
3. Print mode attributes; see the session parameter `PM` (in the *Parameter Reference*).

Examples - Syntax 1

Example 1 - INPUT Statement

```
** Example 'IPTEX1': INPUT
************************************************************************
DEFINE DATA LOCAL
1 #FNC  (A1)
END-DEFINE
*
INPUT 10X 'SELECTION MENU FOR EMPLOYEES SYSTEM' /
      10X '-' (35) //
      10X 'ADD      (A)' /
      10X 'UPDATE   (U)' /
      10X 'DELETE   (D)' /
      10X 'STOP     (.)' //
      10X 'PLEASE ENTER FUNCTION: ' #FNC
*
DECIDE ON EVERY VALUE OF #FNC
  VALUE 'A'   /* invoke the object containing the add function here
    WRITE 'Add function selected.'
  VALUE 'U'   /* invoke the object containing the update function here
    WRITE 'Update function selected.'
  VALUE 'D'   /* invoke the object containing the delete function here
    WRITE 'Delete function selected.'
  VALUE '.'
    STOP
  NONE
    REINPUT 'Please enter a valid function.' MARK *#FNC
END-DECIDE
*
END
```

Output of Program IPTEX1:

```
SELECTION MENU FOR EMPLOYEES SYSTEM
-----------------------------------

ADD      (A)
UPDATE   (U)
DELETE   (D)
STOP     (.)

PLEASE ENTER FUNCTION:
```

Example 2 - INPUT Statement with DEFINE WINDOW Statement

```
** Example 'INPEX1': INPUT (with DEFINE WINDOW statement)
************************************************************************
DEFINE DATA LOCAL
1 #STRING (A15)
END-DEFINE
*
DEFINE WINDOW WIND1
  SIZE 10 * 40
  BASE 5 / 10
  FRAMED ON POSITION TEXT
*
INPUT WINDOW='WIND1'
      'PLEASE ENTER HERE:' / #STRING
*
END
```

Output of Program INPEX1:

```
+----------------------------------Top+
! PLEASE ENTER HERE:                  !
! #STRING                             !
!                                     !
!                                     !
!                                     !
!                                     !
!                                     !
!                                     !
+-------------------------------Bottom+
```

Example 3 - INPUT Statement with MARK POSITION Option

```
** Example 'INPEX2': INPUT (with POSITION)
************************************************************************
DEFINE DATA LOCAL
1 #START (A30)
END-DEFINE
*
ASSIGN #START = 'EXAM_'
*
INPUT (AD=M) MARK POSITION 5 IN *#START
     / 'PLEASE COMPLETE START VALUE FOR SEARCH'
     / 5X #START
END
```

Output of Program INPEX2:

```
PLEASE COMPLETE START VALUE FOR SEARCH
       #START EXAM[]
```

84 INPUT Syntax 2 - Using Predefined Map Layout

This form of the `INPUT` statement is used to perform input processing using a map layout that has been created using the Natural map editor.

Map layouts can be used in two ways:

- the program does not provide a parameter list;
- the program does provide a parameter list (*operand1*).

```
INPUT [WINDOW='window-name'] [WITH-TEXT-option]
 [MARK-option]
 [ALARM-option]
 [USING] MAP map-name [NO ERASE]
[               operand1 ...      ]
[               NO PARAMETER      ]
```

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

INPUT USING MAP without Parameter List

The following requirements must be met when `INPUT USING MAP` is used without parameter list:

- The *map-name* must be specified as an alphanumeric constant (up to 8 characters).
- The map used in this manner must have been created prior to the compilation of the program which references the map.
- The names of the fields to be processed are taken dynamically from the map source definition at compilation time. The field names used in both program and map must be identical.
- All fields to be referenced in the `INPUT` statement must be accessible at that point.
- In structured mode, fields must have been previously defined (database fields must be properly referenced to processing loops or views).
- In reporting mode, user-defined variables may be newly defined in the map.
- When the map layout is changed, the programs using the map need not be recataloged. However, when array structures or names, formats/lengths of fields are changed, or fields are added/deleted in the map, the programs using the map must be recataloged.
- The map source must be available at program compilation; otherwise the `INPUT USING MAP` statement cannot be compiled.

  Note: If you wish to compile the program even if the map is not yet available, specify `NO PARAMETER`: the `INPUT USING MAP` can then be compiled even if the map is not yet available.

INPUT Fields Defined in the Program

By specifying the names of the fields to be processed within the program (`operand1`), it is possible to have the names of the fields in the program differ from the names of the fields in the map.

The sequence of fields in the program must match the map sequence. Please note that the map editor sorts the fields as specified in the map in alphabetical order by field name. For more information, see the map editor description in your Natural Editors documentation.

When the layout of the map is changed, the program using the map need not be recataloged. However, when field names, field formats/lengths, or array structures in the map are changed or fields are added or deleted in the map, the program must be recataloged.

A check is made at execution time to ensure that the format and length of the fields as specified in the program match the fields as specified in the map. If both layouts do not agree, an error message is produced.

INPUT Syntax 2 - Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
map-name	C	S				A	U												yes	no
operand1		S	A			A	U	N	P	I	F	B	D	T	L	C			yes	yes

Syntax Element Description:

Syntax Element	Description
`INPUT WINDOW='`*`window-name`*`'`	**INPUT WINDOW='*window-name*' Option:** This option is described under ***Syntax 1*** of the `INPUT` statement.
`WITH TEXT/MARK/ALARM-options`	**WITH TEXT/MARK/ALARM Options:** These options are described under *Syntax 1* of the `INPUT` statement; see ***WITH TEXT Option***, ***MARK Option***, ***ALARM Option***.
`USING MAP` *`map-name`*	**USING MAP Clause:** `USING MAP` invokes a map definition which has been previously stored in a Natural system file using the map editor.

Syntax Element	Description
	The *map-name* may be a 1- to 8-character alphanumeric constant or user-defined variable. If a variable is used, it must have been previously defined. The map name may contain an ampersand (&); at execution time, this character will be replaced by the one-character code corresponding to the current value of the Natural system variable `*LANGUAGE`. This feature allows the use of multi-lingual maps. The execution of the `INPUT` statement causes the corresponding map to replace the current contents of the screen, unless the `NO ERASE` option is specified, in which case the map will overlay the current contents of the screen.
`NO ERASE`	**NO ERASE Option:** This option is described under ***Syntax 1*** of the `INPUT` statement; see `NO ERASE`.
operand1	**Field Specification:** A list of database fields and/or user-defined variables. The fields must agree in number, sequence, format, length and (for arrays) number of occurrences with the fields in the referenced map; otherwise, an error occurs. When the content of a database field is modified as a result of `INPUT` processing, only the value as contained in the data area is modified. Appropriate database `UPDATE` / `STORE` statements must be used to change the content of the database.

Using the INPUT Statement in Non-Screen Modes

You can change the input mode with the session parameter `IM`.

Forms Mode

In forms mode (profile/session parameter `IM=F`), Natural will display all output text of the map layout on the terminal field by field according to the positioning parameters. This permits the user to enter data on a field by field basis. When all data are entered, the hardcopy output is produced exactly as it would have appeared on the screen.

In forms mode, entering `%R` permits the operator to retype the entire form in case of an error. The input is processed as in the first execution of the `INPUT` statement.

Keyword/Delimiter Mode

In keyword/delimiter mode (profile/session parameter `IM=D`), data can be entered using keywords or positional input values.

General Validation Rules

Data entered in keyword/delimiter mode are validated as for screen mode. An error message will be returned if an attempt is made to enter more characters than defined for a field.

If the `INPUT` statement is to be processed in keyword/delimiter mode on a buffered (3270-type) terminal or a workstation, all data to be assigned to one `INPUT` statement must be entered on one screen. ENTER is only to be used when all data to the `INPUT` statement have been entered.

Keyword Input

Using keyword input, the terminal operator may enter data for the individual fields using the prompting text that, in forms mode, would have been displayed before the value as a keyword to identify the field. The keyword must be followed by the input assign character (`IA` parameter), followed immediately by the data. Any spaces following the assign character are taken as data up to the delimiter character (`ID` parameter). A delimiter character is not required after the last data element. Keyword data for the different fields may be entered in any order separated by the delimiter character. If the operator types in a keyword which is not defined in the `INPUT` statement, an error message will be returned. Data need not be entered for all input fields. Fields for which no data are entered are set to blank for alphanumeric fields and zero for numeric and hexadecimal fields.

A keyword and the corresponding input field must be on the same logical line. If their aggregate length exceeds the line size, adjust the line size (`LS` parameter) accordingly so that keyword and field fit onto one line.

Indexed Input

Using indexed input, the terminal operator may enter data for the individual input fields using their ordinal values prefixed with a percent character (%). This index specification must be followed by the input assign character (`IA` parameter), followed immediately by the data.

Indexed data for the different fields may be entered in any order separated by the delimiter character (`ID` parameter). If the specified ordinal value does not correspond to that of any existing input field, an error message will be returned. Data need not be entered for all input fields. Fields for which no data are entered are set to blank for alphanumeric fields and zero for numeric and hexadecimal fields.

Positional Input

Using positional value input, the terminal operator enters only data for all input fields separated by the currently defined input delimiter character (`ID` parameter). The sequence of fields for input must correspond to the sequence of the fields in the `INPUT` statement.

The user may switch from positional to keyword input by entering a number of values in positional input separated by the delimiter character and then switching to keyword mode for selected fields by specifying keywords in front of the values.

After a keyword has been used to position to a field, any non-keyword input following the keyword will be processed as positional input to be assigned to fields following the previously selected field in the `INPUT` statement.

Example of Keyword, Indexed and Positional Input

If you execute the following program

```
***** Program PGM1 *****
DEFINE DATA LOCAL
1 #F1 (A10)
1 #F2 (A10)
1 #F3 (A10)
END-DEFINE
INPUT (IP=ON) /  'FLD1' #F1
              /  'FLD2' #F2
              /  'FLD3' #F3
WRITE 'FLD1' #F1
    / 'FLD2' #F2
    / 'FLD3' #F3
END
```

from the command line with any of the following commands, assuming the comma (`,`) is used as the delimiter character

`PGM1 FLD1=AA,FLD3=CC`	keyword input
`PGM1 %1=AA,%3=CC`	indexed input
`PGM1 AA,,CC`	positional input
`PGM1 AA,FLD3=CC`	positional input combined with keyword
`PGM1 AA,FLD2=,CC`	positional input combined with keyword
`PGM1 AA,%3=CC`	combined positional and indexed input

you will always receive the following output

```
FLD1 AA
FLD2
FLD3 CC
```

Processing Data from the Natural Stack

Data elements that have been placed in the Natural stack via a `FETCH`, `RUN` or `STACK` statement will be processed by the next `INPUT` statement encountered for execution.

The `INPUT` statement will process the data in keyword/delimiter mode as described above.

If data elements are not available to fill all input fields, fields will be filled with blank/zero depending on the field format. If more data elements are specified than input fields exist, the remaining data are ignored.

When a field is filled with data from the stack, the field attributes do not apply to the data.

The Natural system variable `*DATA` may be referenced to determine the number of data elements currently available in the Natural stack.

Using the INPUT Statement in Batch Mode

The following topics are covered below:

- In Batch Forms Mode
- In Batch Keyword/Delimiter Mode

In Batch Forms Mode

A data record is read for each line containing one or more `AD=A` and/or `AD=M` fields, and the data contained in the record are assigned to the appropriate field (or fields).

Input data fields are assumed to be contiguous. Unless the delimiter character is used, input data must be entered in the exact length according to the internal definition of the field. For numeric fields, space must be allowed for a sign (if `SG=ON`) and decimal point when appropriate.

Data may optionally be entered using the delimiter character to separate the values of the individual fields. In this case, data need not be entered in the exact number of positions according to the internal definition but are processed from left to right beginning in position 1. The rules for data entry are the same as described under ***Entering Data in Response to an INPUT Statement***. In addition, the assign character may be used to skip a field.

In Batch Keyword/Delimiter Mode

Keyword/delimiter mode, when used in batch mode, functions the same as **keyword/delimiter mode** as used for stack input.

XI

85 INSERT (SQL)

Common Set Syntax:

```
INSERT INTO table-name { (*) [VALUES-clause]                }
                       { [(column-list)] VALUE-LIST        }
```

Extended Set Syntax:

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

Belongs to Function Group: ***Database Access and Update***

Function

The SQL `INSERT` statement is used to add one or more new rows to a table.

Syntax Description

Syntax Element	Description
`INTO` *`table-name`*	**INTO Clause:** In the `INTO` clause, the table is specified into which the new rows are to be inserted. See further information on *`table-name`*.
`column-list`	**Column List:** Syntax: *`column-name`*`...` In the *`column-list`*, one or more *`column-names`* can be specified, which are to be supplied with values in the row currently inserted. If a *`column-list`* is specified, the sequence of the columns must match with the sequence of the values either specified in the *`insert-item-list`* or contained in the specified view (see below). If the *`column-list`* is omitted, the values in the *`insert-item-list`* or in the specified view are inserted according to an implicit list of all the columns in the order they exist in the table.
`VALUES-clause`	**Values Clause:** With the `VALUES` clause, you insert a *single* row into the table. See ***VALUES Clause*** below.

VALUES Clause

With the `VALUES` clause, you insert a *single* row into the table. Depending on whether an asterisk (*) or a *`column-list`* has been specified, the `VALUES` clause can take one of the following forms:

VALUES Clause with Preceding Asterisk Notation

```
VALUES (VIEW view-name)
```

If asterisk notation is specified, a view *must* be specified in the `VALUES` clause. With the field values of this view, a new row is inserted into the specified table using the field names of the view as column names of the row.

VALUES Clause with Preceding Column List

```
VALUES ( { VIEW view-name      } )
         { insert-item-list    }
```

If a *`column-list`* is specified and a view is referenced in the `VALUES` clause, the number of items specified in the column list must correspond to the number of fields defined in the view within the *`VALUE-LIST`*.

If no *`column-list`* is specified, the fields defined in the view are inserted according to an implicit list of all the columns in the order they exist in the specified table.

VALUE-LIST

Common Set Syntax:

```
{ VALUES { (VIEW view-name)    } }
         { (insert-item-list)  }
```

Extended Set Syntax:

```
{ VALUES                 { (VIEW view-name)    }        }
{                        { (insert-item-list)  }        }
{                                                       }
{                                 RR                    }
{ select-expression [ WITH  {     RS     } ]            }
{                                 CS                    }
```

Syntax Description:

Syntax Element	Description
`VIEW` *`view-name`*	**View Name:** With the field values of this view, a new row is inserted into the specified table using the field names of the view as column names of the row.

`insert-item-list`	**INSERT Single Row:**
	
	In the *`insert-item-list`*, you can specify one or more values to be assigned to the columns specified in the *`column-list`*. The sequence of the specified values must match the sequence of the columns.
	
	If no *`column-list`* is specified, the values in the *`insert-item-list`* are inserted according to an implicit list of all the columns in the order they exist in the table.
	
	The values to be specified in the *`insert-item-list`* can be *`constants`*, *`parameters`*, *`special-registers`* or `NULL`.
	
	See the section *Basic Syntactical Items* for information on *`view-name`*, *`constant`* and *`parameter`*. See also the information on *`special-register`*.
	
	If the value `NULL` has been assigned, this means that the addressed field is to receive no value (not even the value `0` or "blank").
	
	Example - INSERT Single Row:
	
	`...`
	`INSERT INTO SQL-PERSONNEL (NAME,AGE)`
	`  VALUES ('ADKINSON',35)`
	`...`

`select-expression`	**INSERT Multiple Rows:** This clause belongs to the **SQL Extended Set**. With a *`select-expression`*, you insert *`multiple`* rows into a table. The *`select-expression`* is evaluated and each row of the result table is treated as if the values in this row were specified as values in a ***VALUES Clause*** of a single-row `INSERT` operation. For further information, see ***Select Expressions***. Example - Insert Multiple Rows:

<table>
<tr><th>Syntax Element</th><th colspan="2">Description</th></tr>
<tr><td></td><td colspan="2">

```
...
INSERT INTO SQL-RETIREE (NAME,AGE,SEX)
  SELECT LASTNAME, AGE, SEX
  FROM SQL-EMPLOYEES
  WHERE AGE > 60
...
```

Note: The number of rows that have actually been inserted can be ascertained by using the system variable `*ROWCOUNT`.
</td></tr>
<tr><td rowspan="4"><code>WITH RR/RS/CS</code></td><td colspan="2">

WITH Isolation Level Clause:

This clause belongs to the **SQL Extended Set**.

This clause allows the explicit specification of the isolation level used when locating the rows to be inserted. It is only valid against Db2 databases. When used against other databases, it will cause runtime errors.
</td></tr>
<tr><td><code>CS</code></td><td>Cursor Stability</td></tr>
<tr><td><code>RR</code></td><td>Repeatable Read</td></tr>
<tr><td><code>RS</code></td><td>Read Stability</td></tr>
</table>

86 INTERFACE

```
INTERFACE   interface-name
 [ EXTERNAL]
 [ID interface-GUID]
 [property-definition]
 [method-definition]
END-INTERFACE
```

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

Related Statements: CREATE OBJECT | DEFINE CLASS | INTERFACE | METHOD | PROPERTY | SEND METHOD

Belongs to Function Group: ***Component Based Programming***

Function

In component-based programming, an interface is a collection of methods and properties that belong together semantically and represent a certain feature of a class.

You can define one or several interfaces for a class. Defining several interfaces allows you to structure/group methods according to what they do, for example, you put all methods that deal with persistency (load, store, update) in one interface and put other methods in other interfaces.

The INTERFACE statement is used to define an interface. It may only be used in a Natural class module and can be defined as follows:

- within a DEFINE CLASS statement. This form is used when the interface is only to be implemented in one class, or
- in a copycode which is included by the INTERFACE USING clause of the DEFINE CLASS statement. This form is used when the interface is to be implemented in more than one class.

The properties and methods that are associated with the interface are defined by the property and method definitions.

Syntax Description

Syntax Element	Description
interface-name	**Interface Name:** This is the name to be assigned to the interface. The interface name can be up to a maximum of 32 characters long and must conform to the Natural naming conventions for user-defined variables; see *Naming Conventions for User-Defined Variables* in the *Using Natural* documentation. It must be unique per class and different from the class name. If the interface is planned to be used by clients written in different programming languages, the interface name should be chosen in a way that it does not conflict with the naming conventions that apply in these languages.
EXTERNAL	**EXTERNAL Clause:** This clause is used to indicate that this interface is implemented by the class, but which is originally defined in a different class. The clause is only relevant if the class is to be registered with DCOM. Interfaces with the EXTERNAL clause are ignored when the class is registered with DCOM. It is assumed that the interface is registered by the class that originally defines it.
ID *interface-GUID*	**ID Clause:** This clause is used to assign a globally unique ID to the interface. The *interface-GUID* is the name of a GUID defined in a data area that is included by the LOCAL clause. The *interface-GUID* is a (named) alpha constant. A GUID must be assigned to an interface if the class is to be registered with DCOM.
property-definition	**Property Definition:** The property definition is used to define a property of the interface. See ***Property Definition*** below.
method-definition	**Method Definition:** The method definition is used to define a method for the interface. See ***Method Definition*** below.
END-INTERFACE	**End of INTERFACE Statement:** The Natural reserved word END-INTERFACE must be used to end the INTERFACE statement.

Property Definition

The property definition is used to define a property of the interface.

```
PROPERTY property-name
 [(format-length/array-definition)]
 [ID dispatch-ID]
 [READONLY]
 [IS operand]
END-PROPERTY
```

Properties are attributes of an object that can be accessed by clients. An object that represents an employee might for example have a `Name` property and a `Department` property. Retrieving or changing the name or department of the employee by accessing her `Name` or `Department` property is much simpler for a client than calling one method that returns the value and another method that changes the value.

Each property needs a variable in the object data area of the class to store its value - this is referred to as the object data variable. The property definition is used to make this variable accessible to clients. The property definition defines the name and format of the property and connects it to the object data variable. In the simplest case, the property takes the name and format of the object data variable itself. It is also possible to override the name and format within certain limits.

Syntax Element Description:

Syntax Element	Description
property-name	**Property Name:** This is the name to be assigned to the property. The property name can contain up to a maximum of 32 characters and must conform to the Natural naming conventions for user variables; see *Naming Conventions for User-Defined Variables* in the *Using Natural* documentation. If the property is planned to be used by clients written in different programming languages, the property name should be chosen in a way that it does not conflict with the naming conventions that apply in these languages.
format-length/array-definition	***format-length/array-definition*** **Option:** This option defines the format of the property as it will be seen by clients. If *format-length/array-definition* is omitted, the format-length and array-definition will be taken from the object data variable assigned in the `IS` clause.

Syntax Element	Description
	If *format-length/array-definition* is specified, it must be data transfer-compatible both to and from the format of the object data variable specified in *operand* in the IS clause. In the case of a READONLY property, the data transfer-compatibility needs to hold only in one direction: with the object data variable as source operand and the property as destination operand. If an array-definition is specified, it must be equal in dimensions, occurrences per dimension, lower bounds and upper bounds to the array definition of the corresponding object data variable. This is expressed by specifying an asterisk for each dimension.
ID *dispatch-ID*	**ID Clause:** The ID clause is used to assign a specific numeric identifier to a property. This identifier (*dispatch-ID*) is only relevant if the class is to be registered with DCOM. Normally, Natural automatically assigns a dispatch ID to a property. It is only necessary to explicitly define a specific dispatch ID for a property if the property belongs to an interface with the EXTERNAL clause. (This is an interface that shall be implemented in this class, but which is originally defined in a different class.) In this case the dispatch IDs to be used are usually dictated by the original implementation of the interface. The *dispatch-ID* is a positive, non-zero constant of format I4.
READONLY	**READONLY Option:** If the keyword READONLY is specified, the value of the property can only be read and not set. The format of the object data variable specified in *operand* in the IS clause must be data transfer-compatible to the format specified in *format-length/array-definition*. It does not have to be data transfer-compatible in the inverse direction. If the keyword READONLY is omitted, the property value can be both read and set.
IS *operand*	**IS Clause:** The *operand* in the IS clause assigns an object data variable as the place to store the property value. The assigned object data variable may not be a group. The variable is referenced in normal operand syntax. This means, if the object data variable is an array, it must be referenced with index notation. Only the full index range notation and asterisk notation is allowed. The IS clause should not be used if the INTERFACE statement will be included from a copycode member and reused in several classes. If you want to reuse the INTERFACE statement, you must assign

Syntax Element	Description
	the object data variable in a PROPERTY statement outside the INTERFACE statement. If the IS clause is omitted, the property is connected to the object data variable with the same name as the property. If a variable with this name is not defined or if it is a group, a syntax error results.
END-PROPERTY	**End of Interface Property Definition:** The Natural reserved word END-PROPERTY must be used to end the interface PROPERTY definition.

Examples

Let the object data area contain the following data definitions:

```
1 Salary(p7.2)
1 SalaryHistory(p7.2/1:10)
```

Then the following property definitions are allowed:

```
property Salary
  end-property
  property Pay is Salary
  end-property
  property Pay(P7.2) is Salary
  end-property
  property Pay(N7.2) is Salary
  end-property
  property SalaryHistory
  end-property
  property OldPay is SalaryHistory(*)
  end-property
  property OldPay is SalaryHistory(1:10)
  end-property
  property OldPay(P7.2/*) is SalaryHistory(1:10)
  end-property
  property OldPay(N7.2/*) is SalaryHistory(*)
  end-property
```

The following property definitions are not allowed:

```
/* Not data transfer-compatible. */
  property Pay(L) is Salary
  end-property
  /* Not data transfer-compatible. */
  property OldPay(L/*) is SalaryHistory(*)
  end-property
  /* Not data transfer-compatible. */
  property OldPay(L/1:10) is SalaryHistory(1:10)
  end-property
  /* Assigns an array to a scalar. */
  property OldPay(P7.2) is SalaryHistory(1:10)
  end-property
  /* Takes only a sub-array. */
  property OldPay(P7.2/3:5) is SalaryHistory(*)
  end-property
  /* Index specification omitted in ODA variable SalaryHistory. */
  property OldPay is SalaryHistory
  end-property
  /* Only asterisk notation allowed in property format specification. */
  property OldPay(P7.2/1:10) is SalaryHistory(*)
  end-property
```

Method Definition

The method definition is used to define a method for the interface.

```
METHOD method-name
 [ID dispatch-ID]
 [IS subprogram-name]
[  PARAMETER  { USING parameter-data-area }  ]...
              { data-definition ...       }
END-METHOD
```

To make the interface reusable in different classes, include the interface definition from a copycode and define the subprogram after the interface definition with a `METHOD` statement. Then you can implement the method differently in different classes.

Syntax Element Description:

Syntax Element	Description
method-name	**Method Name:** This is the name to be assigned to the method. The method name can contain a maximum of up to 32 characters and must conform to the Natural naming conventions; see *Naming Conventions for User-Defined Variables* in the *Using Natural* documentation. It must be unique per interface.

Syntax Element	Description
	If the method is planned to be used by clients written in different programming languages, the method name should be chosen in a way that it does not conflict with the naming conventions that apply in these languages.
`ID` *`dispatch-ID`*	**ID Clause:** The `ID` clause is used to assign a specific numeric identifier to a method. This identifier (the so-called dispatch ID) is only relevant if the class is to be registered with DCOM. Normally, Natural automatically assigns a dispatch ID to a method. It is only necessary to explicitly define a specific dispatch ID for a method if the method belongs to an interface with the `EXTERNAL` clause. (This is an interface that shall be implemented in this class, but which is originally defined in a different class.) In this case, the dispatch IDs to be used are usually dictated by the original implementation of the interface. The dispatch ID is a positive, non-zero constant of format I4.
`IS` *`subprogram-name`*	**IS Clause:** This clause can be used to specify the name of the subprogram that implements the method. The name of the subprogram consists of up to 8 characters. The default is method-name (if the `IS` clause is not specified).
`PARAMETER`	**PARAMETER Clause:** The `PARAMETER` clause specifies the parameters of the method, and has the same syntax as the `PARAMETER` clause of the `DEFINE DATA` statement. The parameters must match the parameters which are later used in the implementation of the subprogram. This is ensured best by using a parameter data area. Parameters that are marked `BY VALUE` in the parameter data area are input parameters of the method. Parameters which are not marked `BY VALUE` are passed “by reference” and are input/output parameters. This is the default. The first parameter that is marked `BY VALUE RESULT` is returned as the return value for the method. If more than one parameter is marked in this way, the others will be treated as input/output parameters.
`END-METHOD`	**End of Method Definition:** The Natural reserved word `END-METHOD` must be used to end the `METHOD` definition for the interface.

87 LIMIT

```
LIMIT n
```

Related Statements: ACCEPT/REJECT | AT BREAK | AT START OF DATA | AT END OF DATA | BACKOUT TRANSACTION | BEFORE BREAK PROCESSING | DELETE | END TRANSACTION | FIND | GET | GET SAME | GET TRANSACTION | HISTOGRAM | PASSW | PERFORM BREAK PROCESSING | READ | RETRY | STORE | UPDATE

Belongs to Function Group: ***Database Access and Update***

Function

The LIMIT statement is used to limit the number of iterations of a processing loop initiated with a FIND, READ, or HISTOGRAM statement.

The limit remains in effect for all subsequent processing loops in the program until it is overridden by another LIMIT statement.

The LIMIT statement does not apply to individual statements in which a limit is explicitly specified (for example, FIND (*n*) ...).

If the limit is reached, processing stops and a message is displayed; see also the session parameter LE which determines the reaction when the limit for the processing loop is exceeded.

If no LIMIT statement is specified, the default global limit defined with the Natural profile parameter LT during Natural installation will be used.

Record Counting

To determine whether a processing loop has reached the limit, each record read in the loop is counted against the limit. If the processing loop has reached the limit, the following will apply:

- A record that is rejected because of criteria specified in a FIND or READ statement WHERE clause is *not* counted against the limit.
- A record that is rejected as a result of an ACCEPT/REJECT statement is counted against the limit.

Syntax Description

Syntax Element	Description
`LIMIT` *n*	**Limit Specification:** The limit *n* must be specified as a numeric constant in the range from `0 - 4294967295` (leading zeros are optional). The processing loop is not entered if the limit is set to zero.

Examples

- Example 1 - LIMIT Statement
- Example 2 - LIMIT Statement (Valid for Two Database Loops)

Example 1 - LIMIT Statement

```
** Example 'LMTEX1': LIMIT
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 CITY
END-DEFINE
*
LIMIT 4
*
READ EMPLOY-VIEW BY NAME STARTING FROM 'BAKER'
  DISPLAY NOTITLE
          NAME PERSONNEL-ID CITY *COUNTER
END-READ
*
END
```

Output of Program LMTEX1:

```
        NAME         PERSONNEL         CITY            CNT
                        ID
-------------------- --------- -------------------- -----------

BAKER                20016700  OAK BROOK                      1
BAKER                30008042  DERBY                          2
BALBIN               60000110  BARCELONA                      3
BALL                 30021845  DERBY                          4
```

Example 2 - LIMIT Statement (Valid for Two Database Loops)

```
** Example 'LMTEX2': LIMIT (valid for two database loops)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
END-DEFINE
*
LIMIT 3
*
FIND EMPLOY-VIEW WITH NAME > 'A'
  READ EMPLOY-VIEW BY NAME STARTING FROM 'BAKER'
    DISPLAY NOTITLE 'CNT(0100)' *COUNTER(0100)
                    'CNT(0110)' *COUNTER(0110)
  END-READ
END-FIND
*
END
```

Output of Program LMTEX2:

```
 CNT(0100)   CNT(0110)
----------- -----------

          1           1
          1           2
          1           3
          2           1
          2           2
          2           3
          3           1
          3           2
          3           3
```

88 LOOP

```
[CLOSE] LOOP [(r)]
```

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

Belongs to Function Group: ***Reporting Mode Statements***

Function

The `LOOP` statement is used to close a processing loop. It causes processing of the current pass through the loop to be terminated and control to be returned to the beginning of the processing loop.

When the processing loop for which the `LOOP` statement is issued is terminated (that is, when all records have been processed or iterations have been performed), execution continues with the statement after the `LOOP` statement.

The `LOOP` statement is used with the following statements: `CALL FILE`, `CALL LOOP`, `FIND`, `FOR`, `HISTOGRAM`, `PARSE XML`, `READ`, `READ RESULT SET` (SQL), `READ WORK FILE`, `REPEAT`, `SELECT` (SQL), `SORT`, `UPLOAD PC FILE`.

Database Variable References

A `LOOP` statement, in addition to closing a processing loop, will eliminate all field references to `FIND`, `FIND FIRST`, `FIND UNIQUE`, `READ` and `GET` statements contained within the loop.

A field within a view may be referenced outside the processing loop by using the view name as a qualifier.

Restriction

- This statement is for reporting mode only.
- A `LOOP` statement may not be specified based on a conditional statement such as `IF` or `AT BREAK`.

Syntax Description

Syntax Element	Description
`LOOP (r)`	**Statement Reference Notation:** The `LOOP` statement may be specified with a statement label or reference number (notation `(r)`), in which case all inner loops up to and including the loop initiated by the statement referenced will be closed. If no statement reference is specified, the innermost active processing loop will be closed.

Notes:

1. In reporting mode, any processing loop which is currently active, that is, which has not explicitly been closed with a `LOOP` statement, will be closed automatically by an `END` statement.
2. You can omit the `LOOP` statement. But with respect to good coding practice, you are not recommended to do so.

Examples

Example 1

```
FIND ...
  READ ...
    READ ...
LOOP (0010)    /* closes all loops
```

Example 2

```
FIND ...
  READ ...
    READ ...
    LOOP       /* closes loop initiated on line 0030
  LOOP         /* closes loop initiated on line 0020
LOOP           /* closes loop initiated on line 0010
```

89 METHOD

```
METHOD method-name
 OF [INTERFACE] interface-name
 IS subprogram-name
END-METHOD
```

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

Related Statements: CREATE OBJECT | DEFINE CLASS | INTERFACE | PROPERTY | SEND METHOD

Belongs to Function Group: *Component Based Programming*

Function

The METHOD statement assigns a subprogram as the implementation to a method, *outside* an interface definition. It is used if the interface definition in question is included from a copycode and is to be implemented in a class-specific way.

The METHOD statement may only be used within the DEFINE CLASS statement and after the interface definition. The interface and method names specified must be defined in the INTERFACE clause of the DEFINE CLASS statement.

Syntax Description

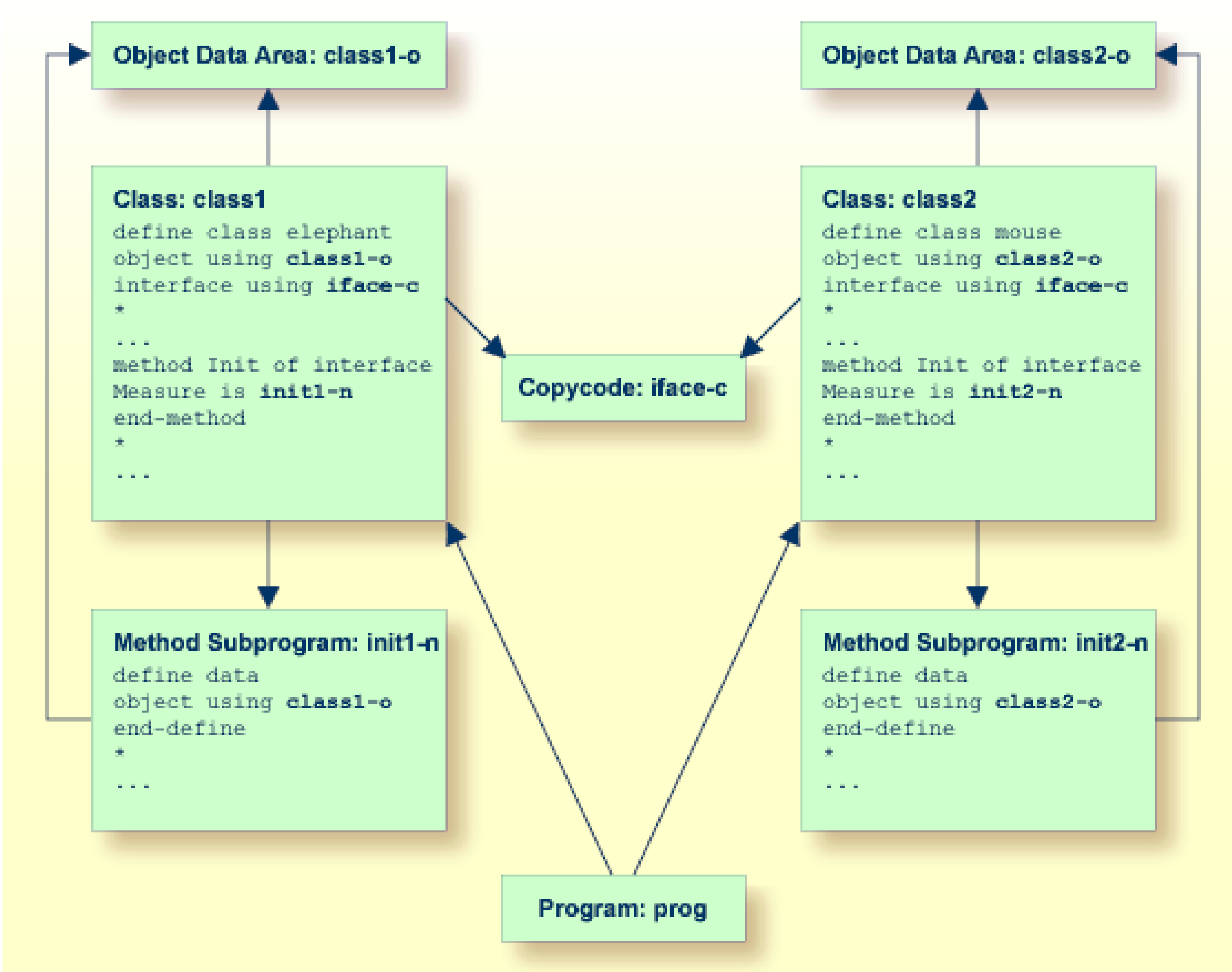
Syntax Element	Description
method-name	**Method Name:** This is the name assigned to the *method*.
OF *interface-name*	**Interface Name:** This is the name assigned to the *interface*.
IS *subprogram-name*	**IS Clause:** This clause can be used to specify the name of the subprogram that implements the method. The name of the subprogram consists of up to 8 characters. The default is *method-name* (if the IS clause is not specified).
END-METHOD	**End of Method Statement:** The Natural reserved word END-METHOD must be used to end the METHOD statement.

Example

The following example shows how the same interface is implemented differently in two classes and how the `PROPERTY` statement and the `METHOD` statement are used to achieve this.

The interface `Measure` is defined in the copycode `iface-c`. The classes `Elephant` and `Mouse` implement both the interface `Measure`. Therefore, they both include the copycode `iface-c`. But the classes implement the property `Height` using different variables from their respective object data areas, and they implement the method `Init` with different subprograms. They use the `PROPERTY` statement to assign the selected data area variable to the property and the `METHOD` statement to assign the selected subprogram to the method.

Now the program `prog` can create objects of both classes and initialize them using the same method `Init`, leaving the specifics of the initialization to the respective class implementation.



The following shows the complete contents of the Natural modules used in the example above:

Copycode: iface-c

```
interface Measure
*
property Height(p5.2)
end-property
*
property Weight(i4)
end-property
*
method Init
end-method
*
end-interface
```

Class: class1

```
define class elephant
object using class1-o
interface using iface-c
*
property Height of interface Measure is height
end-property
*
property Weight of interface Measure is weight
end-property
*
method Init of interface Measure is init1-n
end-method
*
end-class
end
```

LDA Object Data: class1-o

```
*   *** Top of Data Area ***
  1 HEIGHT                  P 5.2
  1 WEIGHT                  I 2
*   *** End of Data Area ***
```

Method Subprogram: init1-n

```
define data
object using class1-o
end-define
*
height := 17.3
weight := 120
*
end
```

Class: class2

```
define class mouse
object using class2-o
interface using iface-c
*
property Height of interface Measure is size
end-property
*
property Weight of interface Measure is weight
end-property
*
method Init of interface Measure is init2-n
end-method
*
end-class
end
```

LDA Object Data: class2-o

```
*   *** Top of Data Area ***
  1 SIZE                    P 3.2
  1 WEIGHT                  I 1
*   *** End of Data Area ***
```

Method Subprogram: init2-n

```
define data
object using class2-o
end-define
*
size := 1.24
weight := 2
*
end
```

Program: prog

```
define data local
1 #o handle of object
1 #height(p5.2)
1 #weight(i4)
end-define
*
create object #o of class 'Elephant'
send "Init" to #o
#height := #o.Height
#weight := #o.Weight
write #height #weight
*
```

```
create object #o of class 'Mouse'
send "Init" to #o
#height := #o.Height
#weight := #o.Weight
write #height #weight
*
end
```

90 MOVE

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

Related Statements: ADD | COMPRESS | COMPUTE | DIVIDE | EXAMINE | MULTIPLY | RESET | SEPARATE | SUBTRACT

Belongs to Function Group: *Arithmetic and Data Movement Operations*

Function

The MOVE statement is used to move the value of an operand to one or more operands (field or array).

A Natural system function may be used only if the MOVE statement is specified in conjunction with an AT BREAK, AT END OF DATA or AT END OF PAGE statement.

See also the section *Rules for Arithmetic Assignment* in the *Programming Guide*.

Syntax 1 - MOVE

```
MOVE [ROUNDED] operand1 [(parameter)] TO operand2 ...
```

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

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A		N	A	U	N	P	I	F	B	D	T	L	C	G	O	yes	no
operand2		S	A		M	A	U	N	P	I	F	B	D	T	L	C	G	O	yes	yes

Syntax Element Description:

Syntax Element	Description
MOVE ROUNDED	**MOVE ROUNDED Option:** This option causes *operand2* to be rounded. ROUNDED is ignored if *operand2* is not numeric or if the source operand has the same or less precision digits than the target operand. See also *Example 1 - Various Samples of MOVE Statement Usage*.
operand1	**Source and Target Operands:**

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td>operand2</td><td>operand1 is the source operand whose value is moved to the target operand operand2.

For the rules for data transfer and information on data conversion and transfer compatibility, see the section Data Transfer in the Programming Guide.

If operand2 is a dynamic variable, its length may be modified by the MOVE operation. The current length of a dynamic variable can be ascertained by using the system variable *LENGTH. For general information on the dynamic variable, see the section Using Dynamic and Large Variables in the Programming Guide.

A MOVE statement with multiple target operands is identical to the corresponding individual MOVE statements:

<code>MOVE #SOURCE TO #TARGET1 #TARGET2</code>

is identical to

<code>MOVE #SOURCE TO #TARGET1</code>
<code>MOVE #SOURCE TO #TARGET2</code></td></tr>
<tr><td>parameter</td><td>Parameter Option:

parameter either specifies the session parameter PM or the session parameter DF:
<table>
<tr><td>PM=I</td><td>Right-to-Left Display Option:

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

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

<code>MOVE 'XYZ' TO #A</code>
<code>MOVE #A (PM=I) TO #B</code>

PM=I can only be specified if operand2 has alphanumeric/Unicode format (Natural data format A or U).

Any trailing blanks in operand1 will be removed , then the value is reversed and moved to operand2. If operand1 is not of alphanumeric/Unicode format, the value will be converted to alphanumeric/Unicode format before it is reversed.

See also the use of PM=I in conjunction with MOVE LEFT/RIGHT JUSTIFIED.</td></tr>
<tr><td></td><td></td></tr>
<tr><td>DF=S|I|L</td><td>Date Format:</td></tr>
</table></td></tr>
</table>

Syntax Element	Description	
		If *operand1* is a date variable and *operand2* is an alphanumeric/Unicode field, you can specify the session parameter `DF` as parameter for this date variable.

Syntax 2 - MOVE SUBSTRING

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

   TO { operand2                                    }
      { SUBSTRING (operand2,operand5,operand6)      } ...
```

This syntax only applies if you want to move only part of the field contents (a substring) of a source and/or target operand. Otherwise, ***Syntax 1*** applies.

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

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A			A	U					B							yes	no
operand2		S	A			A	U					B							yes	no
operand3	C	S						N	P	I		B*							yes	no
operand4	C	S						N	P	I		B*							yes	no
operand5	C	S						N	P	I		B*							yes	no
operand6	C	S						N	P	I		B*							yes	no

* See text.

Syntax Element Description:

Syntax Element	Description
`MOVE SUBSTRING`	**MOVE SUBSTRING:** Without the `SUBSTRING` option, the whole content of a field is moved. The `SUBSTRING` option allows you to move only a certain part of an alphanumeric/ Unicode or a binary field. After the field name (*operand1*) in the `SUBSTRING` clause, you specify first the starting position (*operand3*) and then the length (*operand4*) of the field portion to be moved.

Syntax Element	Description

	If the underlying field format of *operand1* is
	
	■ alphanumeric/Unicode (A) or binary (B), then the values supplied with *operand3* or *operand4* are considered as byte numbers;
	■ Unicode (U), then the values supplied with *operand3* or *operand4* are considered as number of Unicode code units; that is, as double-bytes.
	
	For example, to move the 5th to 12th position inclusive of the value in a field `#A` into a field `#B`, you would specify:
	
	`MOVE SUBSTRING(#A,5,8) TO #B`
	
	If *operand1* is a dynamic variable, the specified field portion to be moved must be within its current length; otherwise, a runtime error will occur.
	
	Also, you can move a value of an alphanumeric/Unicode or binary field into a certain part of the target field. After the field name (*operand2*) in the `SUBSTRING` clause you specify first the starting position (*operand5*) and then the length (*operand6*) of the field portion into which the value is to be moved.
	
	If the underlying field format of *operand2* is
	
	■ alphanumeric/Unicode (A/U) or binary (B), then the values supplied with *operand5* or *operand6* are considered as byte numbers;
	■ Unicode (U), then the values supplied with *operand3* or *operand4* are considered as number of Unicode code units; that is, as double-bytes.
	
	For example, to move the value of a field `#A` into the 3rd to 6th position inclusive of a field `#B`, you would specify:
	
	`MOVE #A TO SUBSTRING(#B,3,4)`
	
	If *operand2* is a dynamic variable, the specified starting position (*operand5*) must not be greater than the variable's current length plus 1; a greater starting position will lead to a runtime error, because it would cause an undefined gap within the content of *operand2*.
	
	If *operand3*/*operand5* or *operand4*/*operand6* is a binary variable, it may be used only with a length of less than or equal to 4.
	
	If you omit *operand3*/*operand5*, the starting position is assumed to be `1`. If you omit *operand4*/*operand6*, the length is assumed to range from the starting position to the end of the field.
	
	If *operand2* is a dynamic variable and the specified starting position (*operand5*) is the variable's current length plus 1, which means that the `MOVE` operation is used to increase the length of the variable, *operand6* must be specified in order to determine the new length of the variable.

Syntax Element	Description
	Note: `MOVE` with the `SUBSTRING` option is a byte-by-byte move (that is, the rules described under *Rules for Arithmetic Assignment* in the *Programming Guide* do *not* apply).
parameter	**Parameter Option:** See *parameter* in *Syntax 1*.

Syntax 3 - MOVE BY NAME / POSITION

```
MOVE  BY { [NAME]    } operand1 TO operand2
         { POSITION  }
```

Operand Definition Table:

Operand	Possible Structure	Possible Formats	Referencing Permitted	Dynamic Definition
operand1	G		yes	no
operand2	G		yes	no

Syntax Element Description:

Syntax Element	Description
MOVE BY NAME *operand1* TO *operand2*	**MOVE BY NAME Option:** This option is used to move individual fields contained in a data structure to another data structure, independent of their position in the structure. A field is moved only if its name appears in both structures (this includes `REDEFINE`d fields as well as fields resulting from a redefinition). The individual fields may be of any format. The operands can also be views. **Note:** The sequence of the individual moves is determined by the sequence of the fields in *operand1*. See also ***Example 2 - MOVE BY NAME Statement***. **MOVE BY NAME with Arrays:** If the data structures contain arrays, these will internally be assigned the index `(*)` when moved; this may lead to an error if the arrays do not comply with the rules for assignment operations with arrays; see the section *Processing of Arrays* in the *Programming Guide*. See also ***Example 3 - MOVE BY NAME with Arrays***.

Syntax Element	Description
`MOVE BY POSITION` *`operand1`* `TO` *`operand2`*	**MOVE BY POSITION Option:** This option allows you to move the contents of fields in a group to another group, regardless of the field names. The values are moved field by field from one group to the other in the order in which the fields are defined (this does not include fields resulting from a redefinition). The individual fields may be of any format. The number of fields in each group must be the same; also, the level structure and array dimensions of the fields must match. Format conversion is done according to the rules for arithmetic assignment; see the section *Rules for Arithmetic Assignments* in the *Programming Guide*. The operands can also be views. See also ***Example 4 - MOVE BY POSITION***.

Syntax 4 - MOVE EDITED (Edit Mask Specified with operand2)

```
MOVE EDITED operand1 TO operand2  { (EM=value)  }
                                  { (EMU=value) }
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A			A	U					B							yes	no
operand2		S	A			A	U	N	P	I	F	B	D	T	L				yes	yes

Syntax Element Description:

Syntax Element	Description
`MOVE EDITED`	**MOVE EDITED Option:** If an edit mask is specified for *`operand2`*, the value of *`operand1`* will be placed into *`operand2`* using this edit mask. The edit mask can be considered as an *input* edit mask for *`operand2`*, that is used to specify at which positions in the alphanumeric/Unicode contents of *`operand1`* the significant input data for *`operand2`* can be found. If the edit mask refers more characters or digits than existent in *`operand2`*, it is truncated accordingly. The length of *`operand1`* may not be smaller than the length of the input value represented by the edit mask. If *`operand1`* is longer than the edit mask length, all the overhanging data is ignored.

Syntax Element	Description

	Under the pre-condition not to have an *operand1* length larger than the edit mask length, you may regard a
	
	`MOVE EDITED operand1 TO operand2 (EM=value)`
	
	operation like the execution of
	
	`STACK TOP DATA operand1`
	`INPUT operand2 (EM=value)`
	
	See also ***Example 1 - Various Samples of MOVE Statement Usage***.

EM	**Edit Mask:** For details on edit masks, see the session parameter EM in the *Parameter Reference*.
EMU	**Unicode Edit Mask:** For details on Unicode edit masks, see the session parameter EMU in the *Parameter Reference*.

Syntax 5 - MOVE EDITED (Edit Mask Specified with operand1)

```
MOVE EDITED operand1 { (EM=value)  } TO operand2
                     { (EMU=value) }
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A		N	A	U	N	P	I	F	B	D	T	L				yes	no
operand2		S	A			A	U					B							yes	yes

Syntax Element Description:

Syntax Element	Description
MOVE EDITED	**MOVE EDITED Option:** If an edit mask is specified for *operand1*, the edit mask will be applied to *operand1* and the result will be moved to *operand2*. The edit mask can be considered as an *output* edit mask for *operand1*, that is used to create an alphanumeric/Unicode string with the layout and length described by the edit mask. Besides data characters or digits originating from *operand1*, you may include additional decoration characters into the output string.

Syntax Element	Description

	If the edit mask refers more characters or digits than existent in `operand1`, it is truncated accordingly. The length of the created output string (resulting from `operand1` value after the edit mask has been applied) must not exceed the length of `operand2`.
	
	Under the pre-condition not to have an `operand2` length smaller than the edit mask length, you may regard a
	
	`MOVE EDITED operand1 (EM=value) TO operand2`
	
	operation like a
	
	`WRITE operand1 (EM=value)`
	
	that does not write the output to the screen, but fills it into variable `operand2`.
	
	See also ***Example 1 - Various Samples of MOVE Statement Usage***.

EM	**Edit Mask:** For details on edit masks, see the session parameter `EM` in the *Parameter Reference*.
EMU	**Unicode Edit Mask:** For details on Unicode edit masks, see the session parameter `EMU` in the *Parameter Reference*.

Syntax 6 - MOVE LEFT / RIGHT JUSTIFIED

```
MOVE { LEFT  } [JUSTIFIED] operand1 [(parameter)] TO operand2
     { RIGHT }
```

Operand Definition Table:

Operand	Possible Structure				Possible Formats	Referencing Permitted	Dynamic Definition
operand1	C	S	A		N A U N P I F B D T L	yes	no
operand2		S	A		A U	yes	yes

Syntax Element Description:

Syntax Element	Description
MOVE LEFT / RIGHT JUSTIFIED	**MOVE LEFT / RIGHT JUSTIFIED Options:** This option is used to cause the values to be moved to be left- or right-justified in *operand2*. MOVE LEFT/RIGHT JUSTIFIED cannot be used if *operand2* is a dynamic variable.
MOVE LEFT JUSTIFIED	**MOVE LEFT Option:** With MOVE LEFT JUSTIFIED, any leading blanks in *operand1* are removed before the value is placed left-justified into *operand2*. The remainder of *operand2* will then be filled with blanks. If the value is longer than *operand2*, the value will be truncated on the right-hand side.
MOVE RIGHT JUSTIFIED	**RIGHT JUSTIFIED Option:** With MOVE RIGHT JUSTIFIED, any trailing blanks in *operand1* are truncated before the value is placed right-justified into *operand2*. The remainder of *operand2* will then be filled with blanks. If the value is longer than *operand2*, the value will be truncated on the left-hand side. See also ***Example 1 - Various Samples of MOVE Statement Usage***.
parameter	**Parameter:** When you use MOVE LEFT/RIGHT JUSTIFIED in conjunction with PM=I, the move is performed in the following steps: 1. If *operand1* is not of alphanumeric/Unicode format, the value is converted to alphanumeric/Unicode format. 2. Any trailing blanks in *operand1* are removed. 3. In the case of LEFT JUSTIFIED, any leading blanks in *operand1* are also removed. 4. The value is reversed, and then moved to *operand2*. 5. If applicable, the remainder of *operand2* is filled with blanks, or the value is truncated (see above).

Syntax 7 - MOVE NORMALIZED

The MOVE NORMALIZED statement converts a Unicode string into the “Unicode Normalization Form C” (NFC). The resulting Unicode string does no longer contain combining sequences for characters which are available as pre-composed characters.

If the format of the target operand is not Unicode itself, an implicit conversion from Unicode into the target operand takes place - during this conversion the default code page (see system variable *CODEPAGE) will be used.

For further information on the `MOVE NORMALIZED` statement, see the section *Statements* in the *Unicode and Code Page Support* documentation.

Syntax Diagram:

```
MOVE NORMALIZED operand1 TO operand2
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A				U												yes	no
operand2		S	A			A	U												yes	yes

Syntax Element Description:

Syntax Element	Description
`MOVE NORMALIZED`	**MOVE NORMALIZED Option:** This option is used to convert Unicode fields with potentially unnormalized content into the "Unicode Normalization Form C" (NFC). This composite form of a Unicode string does not contain combining sequences for characters which are available as pre-composed characters. See also: ***http://www.unicode.org/reports/tr15/#Canonical_Composition_Examples*** ("Normalization Forms D and C Examples"). Example: `MOVE NORMALIZED #SCR TO #TGT`
operand1	**Source Operand:** *operand1* contains the Unicode string to be converted.
operand2	**Target Operand:** *operand2* receives the converted Unicode string.

Example:

Some code points have different representations in Unicode. For example, the German letter `'Ä'`: the decomposed representation in Unicode is `U+0041` followed by `U+0308` and uses a combining character (`U+0308`); another representation is the pre-composed character `U+00C4`. The `MOVE NORMALIZED` statement converts the Unicode representation with combining characters into a normalized Unicode representation using pre-composed characters, where possible.

Syntax 8 - MOVE ENCODED

This section explains the syntax of the `MOVE ENCODED` statement. For information on the purpose of this statement, see the section *Statements* in the *Unicode and Code Page Support* documentation.

Syntax Diagram:

```
MOVE ENCODED
  operand1 [[IN] CODEPAGE operand2] TO
  operand3 [[IN] CODEPAGE operand4]
  [GIVING operand5]
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A			A	U	B											yes	no
operand2		S				A	U												yes	no
operand3		S				A	U	B											yes	yes
operand4		S	A			A	U												yes	no
operand5		S							I4										yes	yes

Syntax Element Description:

Syntax Element	Description
`MOVE ENCODED`	**MOVE ENCODED Option:** This option converts a character string, encoded in one code page, into the equivalent character string of another code page. **Note:** Natural uses the International Components for Unicode (ICU) library for Unicode conversion. For more information, see the ICU User Guide at ***http://userguide.icu-project.org/***.
operand1	**Source Operand:** *operand1* contains the string to be converted.
`CODEPAGE` *operand2*	**Code Page of Source Operand:** As *operand2*, you specify the code page of *operand1*. Can only be supplied if *operand1* is of format A or B. See Note 1.
`TO` *operand3*	**Target Operand:**

Syntax Element	Description
	`operand3` receives the converted string. If the conversion result does not fit into the target field, the result is padded or truncated, respectively, and as padding character the blank of the resulting code page is used. If the target field is defined as a dynamic variable, no padding or truncation is needed, since the length of the dynamic variable is automatically adjusted to the length of the conversion result.
`CODEPAGE operand4`	**Code Page of Target Operand:** As `operand4`, you specify the code page of `operand3`. Can only be supplied if `operand3` is of format A or B. See Note 1.
`GIVING operand5`	**GIVING Clause:** If you omit this clause, a Natural error message is returned if an error occurs. If you specify the keyword `GIVING`, `operand5` returns `0` or the Natural error code instead of the Natural error message. If the target gets truncated, no Natural error message is given, but when the keyword `GIVING` is used, `operand5` will contain an appropriate error code to indicate truncation.

Notes:

1. If a code page operand is not supplied, then the default code page (value of the system variable `*CODEPAGE`) is used.
2. If the session parameter `CPCVERR` in the statement `SET GLOBALS` or in the system command `GLOBALS` is set to `ON`, an error is output if at least one character of the source field could not be converted properly into the destination code page, but was replaced in the target field by a substitution character.

Examples of MOVE ENCODED:

```
MOVE ENCODED A-FIELD1 TO A-FIELD2
```

Invalid: This results in a syntax error, since the code page names are taken by default and are the same for `operand1` and `operand3`.

```
MOVE ENCODED A-FIELD1 CODEPAGE 'IBM01140' TO A-FIELD2 CODEPAGE 'IBM01140'
```

Invalid: This results in an error, since the coded code page names are the same for *operand1* and *operand3*.

```
MOVE ENCODED A-FIELD1 CODEPAGE 'IBM01140' TO A-FIELD2 CODEPAGE 'IBM037'
```

Valid: The string in `A-FIELD1` which is coded in IBM01140 is converted into `A-FIELD2` which is coded in IBM037.

```
MOVE ENCODED U-FIELD TO U-FIELD
```

Invalid: This results in an error, since at least one operand must be of format A or B.

```
MOVE ENCODED U-FIELD TO A-FIELD
```

Valid: The Unicode string in `U-FIELD` which, considered to be encoded in UTF-16, is converted into the alphanumeric `A-FIELD` in the default code page (`*CODEPAGE`).

```
MOVE ENCODED A-FIELD TO U-FIELD
```

Valid: The string in `A-FIELD` which, considered to be encoded in the default code page (`*CODEPAGE`), is converted into the Unicode field `U-FIELD`.

```
MOVE ENCODED A100-FIELD CODEPAGE 'IBM1140' TO A50-FIELD CODEPAGE 'IBM037'
```

Valid: Conversion is done from `A100-FIELD` (format/length: A100) to `A50-FIELD` (format/length: A50), using the relevant code pages. The target is truncated. No Natural error message is returned.

```
MOVE ENCODED A100-FIELD CODEPAGE 'IBM1140' TO A50-FIELD
CODEPAGE 'IBM037' GIVING RC-FIELD
```

Valid: Conversion is done from `A100-FIELD` (format/length: A100) to `A50-FIELD` (format/length: A50), using the relevant code pages. The target is truncated. Since a `GIVING` clause is supplied, the `RC-FIELD` receives an error code, indicating that a value truncation has taken place.

Syntax 9 - MOVE ALL

The `MOVE ALL` statement enables you to move repeatedly the content of *operand1* to *operand2* until the complete target field is full or the `UNTIL` value (*operand7*) is reached.

Using a `SUBSTRING` **Clause**, you may limit the `MOVE ALL` operation to just segments of the source and target field.

Syntax Diagram:

```
MOVE ALL { operand1
         { SUBSTRING (operand1,operand3,operand4) }

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

      [UNTIL operand7]
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A			A	U	N[1]				B							yes	no
operand2		S	A			A	U					B							yes	yes
operand3	C	S						N	P	I		B[2]							yes	no
operand4	C	S						N	P	I		B[2]							yes	no
operand5	C	S						N	P	I		B[2]							yes	no
operand6	C	S						N	P	I		B[2]							yes	no
operand7	C	S						N	P	I									yes	no

[1] A numeric format (N) for *operand1* is permitted only when used without the `SUBSTRING` clause.

[2] If *operand3*/*operand5* or *operand4*/*operand6* is a binary variable, it may be used only with a length of less than or equal to 4.

Syntax Element Description:

Syntax Element	Description
operand1	**Source Operand:** The source operand contains the value to be moved. All digits of a numeric operand including leading zeros are moved.
`TO` *operand2*	**Target Operand:** The target operand is not reset before the execution of the `MOVE ALL` operation. This is of particular importance when using the `UNTIL` option since data previously in *operand2* is retained if not explicitly overlaid during the `MOVE ALL` operation.
`UNTIL` *operand7*	**UNTIL Option:** The `UNTIL` option can be used to limit the `MOVE ALL` operation to a given number of positions in *operand2*. *operand3* is used to specify the number of positions. The `MOVE ALL` operation is terminated when this value is reached. If *operand7* is greater than the length of *operand2*, the `MOVE ALL` operation is terminated when *operand2* is full.

Syntax Element	Description
	The `UNTIL` option may also be used to assign an initial value to a dynamic variable: if *operand2* is a dynamic variable, its length after the `MOVE ALL` operation will correspond to the value of *operand7*. The current length of a dynamic variable can be ascertained by using the system variable `*LENGTH`. For general information on dynamic variables, see *Usage of Dynamic Variables*. **Note:** The `UNTIL` option is not allowed when a `SUBSTRING` clause is used for the target operand.
`SUBSTRING`	**SUBSTRING Clause:** The `SUBSTRING` clause enables you to select a fixed segment of the source or target variable in a `MOVE ALL` statement - whereas, without the `SUBSTRING` clause, the whole content of the source or target variable is processed. *operand3* and *operand4* describe the start position and length of the *operand1* segment used as source value. *operand5* and *operand6* describe the start position and length of the *operand2* segment which is filled by the operation. If the start position (*operand3* or *operand5*) is omitted, then position 1 is assumed by default. If the substring length (*operand4* or *operand6*) is omitted, then the remaining length of the field is assumed. If `SUBSTRING` is used for the source field, the start value and length (*operand3* and *operand4*) must describe a data segment which is completely inside *operand1*. If `SUBSTRING` is used for the target field the following rules apply: ■ If *operand2* is a fixed length variable, the range described by the start-value and length (*operand5* and *operand6*) has completely to reside within the field extent. ■ If *operand2* is a dynamic length variable, the start value (*operand5*) can either point into or immediately behind the current field length (`*LENGTH + 1`). When the end of the `SUBSTRING` range is within the allocated field data, the operation is processed in the same way as for a fixed length field. When the `SUBSTRING` end exceeds the current field size, the dynamic variable is expanded to this extent. See also *Examples of SUBSTRING Clause Usage* below.

Examples of SUBSTRING Clause Usage

```
DEFINE DATA LOCAL
1 ALFA (A10) INIT <'AAAAAAAAAA'>
1 DYN (A) DYNAMIC INIT <'1234567890'>
1 #VAL (A4) INIT <'1234'>
END-DEFINE
```

Statement	Result	
	Before	After
MOVE ALL SUBSTRING (#VAL,1,2) TO ALFA	AAAAAAAAAA	1212121212
MOVE ALL '123' TO SUBSTRING (ALFA,3,5)	AAAAAAAAAA	AA12312AAA
MOVE ALL 'x' TO SUBSTRING (DYN,7,3)	1234567890 (*LENGTH=10)	123456xxx0 (*LENGTH=10)
MOVE ALL 'xyz' TO SUBSTRING (DYN,7,6)	1234567890 (*LENGTH=10)	123456xyzxyz (*LENGTH=12)
MOVE ALL 'xyz' TO SUBSTRING (DYN,11,4)	1234567890 (*LENGTH=10)	1234567890xyzx (*LENGTH=14)

Examples

Example 1 - Various Samples of MOVE Statement Usage

```
** Example 'MOVEX1': MOVE
************************************************************************
DEFINE DATA LOCAL
1 #A (N3)
1 #B (A5)
1 #C (A2)
1 #D (A7)
1 #E (N1.0)
1 #F (A5)
1 #G (N3.2)
1 #H (A6)
END-DEFINE
*
MOVE 5 TO #A
WRITE NOTITLE 'MOVE 5 TO #A'       30X '=' #A
*
MOVE 'ABCDE' TO #B #C #D
WRITE 'MOVE ABCDE TO #B #C #D'     20X '=' #B '=' #C '=' #D
*
MOVE -1  TO #E
WRITE 'MOVE -1  TO #E'             28X '=' #E
*
MOVE ROUNDED 1.995 TO #E
```

```
WRITE 'MOVE ROUNDED 1.995 TO #E'  18X '=' #E
*
*
MOVE RIGHT JUSTIFIED 'ABC' TO #F
WRITE 'MOVE RIGHT JUSTIFIED ''ABC'' TO #F'       10X '=' #F
*
MOVE EDITED '003.45' TO #G (EM=999.99)
WRITE 'MOVE EDITED ''003.45'' TO #G (EM=999.99)'  4X '=' #G
*
MOVE EDITED 123.45 (EM=999.99) TO #H
WRITE 'MOVE EDITED 123.45 (EM=999.99) TO #H'      6X '=' #H
*
END
```

Output of Program MOVEX1:

```
MOVE 5 TO #A                              #A:    5
MOVE ABCDE TO #B #C #D                    #B: ABCDE #C: AB #D: ABCDE
MOVE -1  TO #E                            #E: -1
MOVE ROUNDED 1.995 TO #E                  #E:  2
MOVE RIGHT JUSTIFIED 'ABC' TO #F          #F:   ABC
MOVE EDITED '003.45' TO #G (EM=999.99)    #G:    3.45
MOVE EDITED 123.45 (EM=999.99) TO #H      #H: 123.45
```

Example 2 - MOVE BY NAME

```
** Example 'MOVEX2': MOVE BY NAME
************************************************************************
DEFINE DATA LOCAL
1 #SBLOCK
  2 #FIELDA (A10) INIT <'AAAAAAAAAA'>
  2 #FIELDB (A10) INIT <'BBBBBBBBBB'>
  2 #FIELDC (A10) INIT <'CCCCCCCCCC'>
  2 #FIELDD (A10) INIT <'DDDDDDDDDD'>
1 #TBLOCK
  2 #FIELD1 (A15) INIT <' '>
  2 #FIELDA (A10) INIT <' '>
  2 #FIELD2 (A10) INIT <' '>
  2 #FIELDB (A10) INIT <' '>
  2 #FIELD3 (A20) INIT <' '>
  2 #FIELDC (A10) INIT <' '>
END-DEFINE
*
MOVE BY NAME #SBLOCK TO #TBLOCK
*
WRITE NOTITLE 'CONTENTS OF #TBLOCK AFTER MOVE BY NAME:'
       // '=' #TBLOCK.#FIELD1
        / '=' #TBLOCK.#FIELDA
        / '=' #TBLOCK.#FIELD2
        / '=' #TBLOCK.#FIELDB
        / '=' #TBLOCK.#FIELD3
```

```
        / '=' #TBLOCK.#FIELDC
*
END
```

Contents of #TBLOCK after MOVE BY NAME Processing:

```
CONTENTS OF #TBLOCK AFTER MOVE BY NAME:

#FIELD1:
#FIELDA: AAAAAAAAAA
#FIELD2:
#FIELDB: BBBBBBBBBB
#FIELD3:
#FIELDC: CCCCCCCCCC
```

Example 3 - MOVE BY NAME with Arrays

```
DEFINE DATA LOCAL
  1 #GROUP1
    2 #FIELD (A10/1:10)
  1 #GROUP2
    2 #FIELD (A10/1:10)
END-DEFINE
...
MOVE BY NAME #GROUP1 TO #GROUP2
...
```

In this example, the `MOVE` statement would internally be resolved as:

```
MOVE #GROUP1.#FIELD (*) TO #GROUP2.#FIELD (*)
```

If part of an indexed group is moved to another part of the same group, this may lead to unexpected results as shown in the example below.

```
DEFINE DATA LOCAL
  1 #GROUP1 (1:5)
    2 #FIELDA (N1) INIT <1,2,3,4,5>
    2 REDEFINE #FIELDA
      3 #FIELDB (N1)
END-DEFINE
...
MOVE BY NAME #GROUP1 (2:4) TO #GROUP1 (1:3)
...
```

In this example, the `MOVE` statement would internally be resolved as:

```
MOVE #FIELDA (2:4) TO #FIELDA (1:3)
MOVE #FIELDB (2:4) TO #FIELDB (1:3)
```

First, the contents of the occurrences 2 to 4 of `#FIELDA` are moved to the occurrences 1 to 3 of `#FIELDA`; that is, the occurrences receive the following values:

Occurrence:	1.	2.	3.	4.	5.
Value before:	1	**2**	**3**	**4**	5
Value after:	**2**	**3**	**4**	4	5

Then the contents of the occurrences 2 to 4 of `#FIELDB` are moved to the occurrences 1 to 3 of `#FIELDB`; that is, the occurrences receive the following values:

Occurrence:	1.	2.	3.	4.	5.
Value before:	2	**3**	**4**	**4**	5
Value after:	**3**	**4**	**4**	4	5

Example 4 - MOVE BY POSITION

```
DEFINE DATA LOCAL
  1 #GROUP1
    2 #FIELD1A (N5)
    2 #FIELD1B (A3/1:3)
    2 REDEFINE #FIELD1B
      3 #FIELD1BR (A9)
  1 #GROUP2
    2 #FIELD2A (N5)
    2 #FIELD2B (A3/1:3)
    2 REDEFINE #FIELD2B
      3 #FIELD2BR (A9)
END-DEFINE
...
MOVE BY POSITION #GROUP1 TO #GROUP2
...
```

In this example, the content of `#FIELD1A` is moved to `#FIELD2A`, and the content of `#FIELD1B` to `#FIELD2B`; the fields `#FIELD1BR` and `#FIELD2BR` are not affected.

Example 5 - MOVE ALL

```
** Example 'MOAEX1': MOVE ALL
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 CITY
1 VEH-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 4
RD. READ EMPLOY-VIEW BY NAME
  SUSPEND IDENTICAL SUPPRESS
  /*
  FD. FIND VEH-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      MOVE ALL '*' TO FIRST-NAME (RD.)
      MOVE ALL '*' TO CITY (RD.)
      MOVE ALL '*' TO MAKE (FD.)
    END-NOREC
    /*
    DISPLAY NOTITLE (ES=OFF IS=ON ZP=ON AL=15)
            NAME (RD.) FIRST-NAME (RD.)
            CITY (RD.)
            MAKE (FD.) (IS=OFF)
    /*
  END-FIND
END-READ
END
```

Output of Program MOAEX1:

```
     NAME         FIRST-NAME         CITY            MAKE
--------------- --------------- --------------- ---------------

ABELLAN         *************** *************** ***************
ACHIESON        ROBERT          DERBY           FORD
ADAM            *************** *************** ***************
ADKINSON        JEFF            BROOKLYN        GENERAL MOTORS
```

91 MOVE INDEXED

The `MOVE INDEXED` statement is supported for compatibility reasons only.

Caution: In contrast to a `MOVE` statement with array operands, checks for out-of-bound index values are not possible when a `MOVE INDEXED` statement is executed. As a consequence, when executing an incorrect `MOVE INDEXED` statement, you may unintentionally destroy user data.

Therefore, Software AG strongly recommends that you replace `MOVE INDEXED` statements by `MOVE` statements.

See the statement `MOVE`.

92 MULTIPLY

Related Statements: ADD | COMPRESS | COMPUTE | DIVIDE | EXAMINE | MOVE | MOVE ALL | RESET | SEPARATE | SUBTRACT

Belongs to Function Group: ***Arithmetic and Data Movement Operations***

Function

The `MULTIPLY` statement is used to multiply two operands. Depending on the syntax used, the result of the multiplication may be stored in *operand1* or *operand3*.

If a database field is used as the result field, the multiplication results in an update only to the internal value of the field as used within the program. The value for the field in the database remains unchanged.

For multiplications involving arrays, see also *Rules for Arithmetic Assignments*, *Arithmetic Operations with Arrays* (in the *Programming Guide*).

Two different structures are possible for this statement.

Syntax 1 - MULTIPLY Statement without GIVING Clause

When Syntax 1 used, the result of the multiplication can be stored in *operand1*.

```
MULTIPLY [ROUNDED] operand1 BY  { (arithmetic-expression) }
                                { operand2                }
```

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

Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
operand1		S	A		M	N	P	I	F									yes	no
operand2	C	S	A		N	N	P	I	F									yes	no

Syntax Element Description:

Syntax Element	Description
arithmetic-expression	See ***Arithmetic Expression*** in the COMPUTE statement.
operand1 BY *operand2*	**Operands:** *operand1* is the multiplicand, *operand2* is the multiplier. The result is stored in *operand1*, hence the statement is equivalent to: `operand1 := operand1 * operand2`
ROUNDED	**ROUNDED Option:** If you specify the keyword ROUNDED, the value will be rounded before it is assigned to *operand1* or *operand3*. For information on rounding, see *Rules for Arithmetic Assignment, Field Truncation and Field Rounding* in the *Programming Guide*.

Syntax 2 - MULTIPLY Statement with GIVING Clause

When Syntax 2 is used, the result of the multiplication can be stored in *operand3*.

```
MULTIPLY   { (arithmetic-expression) }  BY  { (arithmetic-expression) }  GIVING
[ROUNDED]  { operand1                }      { operand2                }  operand3
```

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

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A		M			N	P	I	F								yes	no
operand2	C	S	A		N			N	P	I	F								yes	no
operand3		S	A		M	A	U	N	P	I	F	B*		T					yes	yes

* Format B of *operand3* may be used only with a length of less than or equal to 4.

Syntax Element Description:

Syntax Element	Description
arithmetic-expression	See ***Arithmetic Expression*** in the `COMPUTE` statement.
operand1 BY *operand2* GIVING *operand3*	**Operands:** *operand1* is the multiplicand, *operand2* is the multiplier. The result will be stored in *operand3*, hence the statement is equivalent to: `operand3 := operand1 * operand2`
ROUNDED	**ROUNDED Option:** If you specify the keyword `ROUNDED`, the value will be rounded before it is assigned to *operand1* or *operand3*. For information on rounding, see *Rules for Arithmetic Assignment, Field Truncation and Field Rounding* in the *Programming Guide*.

Example

```
** Example 'MULEX1': MULTIPLY
************************************************************************
DEFINE DATA LOCAL
1 #A      (N3) INIT <20>
1 #B      (N5)
1 #C      (N3.1)
1 #D      (N2)
1 #ARRAY1 (N5/1:4,1:4) INIT (2,*) <5>
1 #ARRAY2 (N5/1:4,1:4) INIT (4,*) <10>
END-DEFINE
*
MULTIPLY #A BY 3
WRITE NOTITLE 'MULTIPLY #A BY 3'            25X '=' #A
*
MULTIPLY #A BY 3 GIVING #B
WRITE 'MULTIPLY #A BY 3 GIVING #B'          15X '=' #B
*
MULTIPLY ROUNDED 3 BY 3.5 GIVING #C
WRITE 'MULTIPLY ROUNDED 3 BY 3.5 GIVING #C'  6X '=' #C
*
MULTIPLY 3 BY -4 GIVING #D
WRITE 'MULTIPLY  3 BY -4 GIVING #D'         14X '=' #D
*
MULTIPLY -3 BY -4 GIVING #D
WRITE 'MULTIPLY -3 BY -4 GIVING #D'         14X '=' #D
*
MULTIPLY  3 BY  0 GIVING #D
WRITE 'MULTIPLY  3 BY  0 GIVING #D'         14X '=' #D
*
```

```
WRITE / '=' #ARRAY1 (2,*) '=' #ARRAY2 (4,*)
MULTIPLY #ARRAY1 (2,*) BY #ARRAY2 (4,*)
WRITE / 'MULTIPLY #ARRAY1 (2,*) BY #ARRAY2 (4,*)'
      / '=' #ARRAY1 (2,*) '=' #ARRAY2 (4,*)
*
END
```

Output of Program MULEX1:

```
MULTIPLY #A BY 3                         #A:   60
MULTIPLY #A BY 3 GIVING #B               #B:    180
MULTIPLY ROUNDED 3 BY 3.5 GIVING #C      #C:   10.5
MULTIPLY  3 BY -4 GIVING #D              #D: -12
MULTIPLY -3 BY -4 GIVING #D              #D:  12
MULTIPLY  3 BY  0 GIVING #D              #D:   0

#ARRAY1:      5      5      5      5 #ARRAY2:     10     10     10     10

MULTIPLY #ARRAY1 (2,*) BY #ARRAY2 (4,*)
#ARRAY1:     50     50     50     50 #ARRAY2:     10     10     10     10
```

93 NEWPAGE

```
NEWPAGE [(rep)] [ EVEN [IF] TOP [OF] [PAGE]
                  [ IF   ] LESS [THAN] operand1 [LINES] [LEFT] ]
                  [ WHEN ]
[WITH] TITLE title-definition
```

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

Related Statements: `AT END OF PAGE` | `AT TOP OF PAGE` | `CLOSE PRINTER` | `DEFINE PRINTER` | `DISPLAY` | `EJECT` | `FORMAT` | `PRINT` | `SKIP` | `SUSPEND IDENTICAL SUPPRESS` | `WRITE` | `WRITE TITLE` | `WRITE TRAILER`

Belongs to Function Group: ***Creation of Output Reports***

Function

The `NEWPAGE` statement is used to cause an advance to a new page. `NEWPAGE` also causes any `AT END OF PAGE` and `WRITE TRAILER` statements to be executed. A default title containing the date, time of day, and page number will appear on each new page unless a `WRITE TITLE`, `WRITE NOTITLE`, or `DISPLAY NOTITLE` statement is specified to define specific title processing.

Notes:

1. The advance to a new page is not performed at the time when the `NEWPAGE` statement is executed. It is performed only when a subsequent statement which produces output is executed.
2. If the `NEWPAGE` statement is not used, page advance is controlled automatically based on the Natural profile/session parameter `PS` (Page Size for Natural Reports).

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
operand1	C	S				N	P	I										yes	no

Syntax Element Description:

Syntax Element	Description
`(rep)`	**Report Specification:** The notation `(rep)` may be used to specify the identification of the report for which the `NEWPAGE` statement is applicable. A value in the range `0 - 31` or a logical name which has been assigned using the `DEFINE PRINTER` statement may be specified. If `(rep)` is not specified, the `NEWPAGE` statement will be applicable to the first report (Report 0). For information on how to control the format of an output report created with Natural, see *Report Format and Control* (in the *Programming Guide*).
`EVEN IF TOP OF PAGE`	**EVEN IF TOP OF PAGE Option:** This option is used to cause a new page (with corresponding `AT TOP OF PAGE` and page title processing) to be generated, even if a new page was initiated immediately before the `NEWPAGE` statement was encountered.
`WHEN LESS THAN` *`operand1`* `LINES LEFT`	**WHEN LESS THAN ... LINES LEFT Option:** This option is used to cause a new page to be generated when there are less than *`operand1`* lines left on the current page (current line count compared with value for the Natural profile/session parameter `PS`).
`WITH TITLE` *`title-definition`*	**WITH TITLE Option:** This option can be used to specify a title which is to be written to the new page generated. The *`title-definition`* is specified using the same syntax as described for the `WRITE TITLE` statement, except that the `SKIP` clause is not allowed in a `NEWPAGE WITH TITLE` *`title-definition`* statement.

Example

```
** Example 'NWPEX1': NEWPAGE
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 SALARY    (1)
  2 CURR-CODE (1)
END-DEFINE
*
LIMIT 15
READ EMPLOY-VIEW BY CITY FROM 'DENVER'
```

```
  DISPLAY CITY (IS=ON) NAME SALARY (1) CURR-CODE (1)
  AT BREAK OF CITY
    SKIP 1
    /*
    NEWPAGE WHEN LESS THAN 10 LINES LEFT
    WRITE '****************************************'
      /   'SUMMARY FOR ' OLD(CITY)
      /   '****************************************'
      /   '****************************************'
      /   'SUM OF SALARIES:' SUM(SALARY(1))
      /   'AVG OF SALARIES:' AVER(SALARY(1))
      /   '****************************************'
    NEWPAGE
    /*
  END-BREAK
END-READ
END
```

Output of Program NWPEX1 - Page 1:

```
Page      1                                                05-01-18  10:01:45

        CITY                NAME            ANNUAL   CURRENCY
                                            SALARY     CODE
-------------------- -------------------- ---------- --------

DENVER               TANIMOTO                  33000 USD
                     MEYER                     50000 USD

****************************************
SUMMARY FOR  DENVER
****************************************
****************************************
SUM OF SALARIES:      83000
AVG OF SALARIES:      41500
****************************************
```

Output of Program NWPEX1 - Page 2:

```
Page      2                                                05-01-18  10:01:45

        CITY                NAME            ANNUAL   CURRENCY
                                            SALARY     CODE
-------------------- -------------------- ---------- --------

DERBY                DEAKIN                     8750 UKL
                     GARFIELD                   6750 UKL
                     MUNN                       8800 UKL
                     MUNN                       5650 UKL
                     GREBBY                     9550 UKL
                     WHITT                      8650 UKL
```

```
                    PONSONBY                   5500 UKL
                    MAGUIRE                    4150 UKL
                    HEYWOOD                    3900 UKL
                    BRYDEN                     6750 UKL
                    SMITH                     39000 UKL
                    CONQUEST                  45000 UKL
                    ACHIESON                  11300 UKL

****************************************
SUMMARY FOR  DERBY
****************************************
```

Output of Program NWPEX1 - Page 3:

```

-------------------- -------------------- ---------- --------

DERBY               DEAKIN                     8750 UKL
                    GARFIELD                   6750 UKL
                    MUNN                       8800 UKL
                    MUNN                       5650 UKL
                    GREBBY                     9550 UKL
                    WHITT                      8650 UKL
                    PONSONBY                   5500 UKL
                    MAGUIRE                    4150 UKL
                    HEYWOOD                    3900 UKL
                    BRYDEN                     6750 UKL
                    SMITH                     39000 UKL
                    CONQUEST                  45000 UKL
                    ACHIESON                  11300 UKL

****************************************
SUMMARY FOR  DERBY
****************************************
****************************************
SUM OF SALARIES:    163750
AVG OF SALARIES:     12596
****************************************
```

94 OBTAIN

```
OBTAIN  operand1 ...
```

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

Belongs to Function Group: **Reporting Mode Statements**

Function

The `OBTAIN` statement is used in reporting mode to cause one or more fields to be read from a file. The `OBTAIN` statement does not generate any executable code in the Natural object program. It is primarily used to read a range of values of a multiple-value field or a range of occurrences of a periodic group so that portions of these ranges may be subsequently referenced in the program.

An `OBTAIN` statement is *not* required for each database field to be referenced in the program since Natural automatically reads each database field referenced in a subsequent statement (for example, a `DISPLAY` or `COMPUTE` statement).

When multiple-value or periodic-group fields in the form of an array are referenced, the array must be defined with an `OBTAIN` statement to ensure that it is built for all occurrences of the fields. If individual multiple-value or periodic-group fields are referenced before the array is defined, the fields will not be placed within the array and will exist independent of the array. The fields will contain the same value as the corresponding occurrence within the array.

Individual occurrences of multiple-value or periodic-group fields or subarrays can be held within a previously defined array if the array dimensions of the second individual occurrence or array are contained within the initial array.

References to multiple-value or periodic-group fields with unique variable index cannot be contained in an array of values. If individual occurrences of an array are to be processed with a variable index, the index expression must be prefixed with the unique variable index to denote the individual array.

Restriction

The `OBTAIN` statement is for reporting mode only.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1		S	A	G		A	U	N	P	I	F	B	D	T	L				yes	yes

Syntax Element Description:

Syntax Element	Description
operand1	**Fields to be Read:** With *operand1* you specify the field(s) to be made available as a result of the `OBTAIN` statement.

Examples:

```
READ FINANCE OBTAIN CREDIT-CARD (1-10)
DISPLAY CREDIT-CARD (3-5) CREDIT-CARD (6-8)
SKIP 1 END
```

The above example results in the first 10 occurrences of the field `CREDIT-CARD` (which is contained in a periodic group) being read and occurrences 3-5 and 6-8 being displayed where the subsequent subarrays will reside in the initial array (1-10).

```
READ FINANCE
MOVE 'ONE' TO CREDIT-CARD (1)
DISPLAY CREDIT-CARD (1) CREDIT-CARD (1-5)
```

Output:

```
    CREDIT-CARD       CREDIT-CARD
------------------ ----------------



ONE                 DINERS CLUB
                    AMERICAN EXPRESS





ONE                 AVIS
                    AMERICAN EXPRESS


```

```
ONE                 HERTZ
                    AMERICAN EXPRESS



ONE                 UNITED AIR TRAVEL
```

The first reference to `CREDIT-CARD (1)` is not contained within the array. The array which is defined after the reference to the unique occurrence (1) cannot retroactively include a unique occurrence or an array which is shorter than the one being defined.

```
READ FINANCE
OBTAIN CREDIT-CARD (1-5)
MOVE 'ONE' TO CREDIT-CARD (1)
DISPLAY CREDIT-CARD (1) CREDIT-CARD (1-5)
```

Output:

```
    CREDIT-CARD      CREDIT-CARD
------------------ ----------------


ONE                 ONE
                    AMERICAN EXPRESS



ONE                 ONE
                    AMERICAN EXPRESS



ONE                 ONE
                    AMERICAN EXPRESS



ONE                 ONE
```

The individual reference to `CREDIT-CARD (1)` is contained within the array defined in the `OBTAIN` statement.

```
MOVE (1) TO INDEX
READ FINANCE
DISPLAY CREDIT-CARD (1-5) CREDIT-CARD (INDEX)
```

Output:

```
    CREDIT-CARD       CREDIT-CARD
------------------- ----------------



DINERS CLUB         DINERS CLUB
AMERICAN EXPRESS




AVIS                AVIS
AMERICAN EXPRESS




HERTZ              HERTZ
AMERICAN EXPRESS




UNITED AIR TRAVEL  UNITED AIR TRAVEL
```

The reference to `CREDIT-CARD` using the variable index notation is not contained within the array.

```
RESET A(A20) B(A20) C(A20)
MOVE 2 TO I (N3)
MOVE 3 TO J (N3)
READ FINANCE
 OBTAIN CREDIT-CARD (1:3) CREDIT-CARD (I:I+2) CREDIT-CARD (J:J+2)
 FOR K (N3) = 1 TO 3
  MOVE CREDIT-CARD (1.K) TO A
  MOVE CREDIT-CARD (I.K) TO B
  MOVE CREDIT-CARD (J.K) TO C
  DISPLAY A B C
 LOOP /* FOR
LOOP / * READ
END
```

Output:

```
         A                    B                    C
------------------- ------------------- -------------------
CARD 01             CARD 02             CARD 03
CARD 02             CARD 03             CARD 04
CARD 03             CARD 04             CARD 05
```

The three arrays may be accessed individually by using the unique base index as qualifier for the index expression.

Invalid Example 1

```
READ FINANCE
OBTAIN CREDIT-CARD (1-10)
FOR I 1 10
MOVE CREDIT-CARD (I) TO A(A20)
WRITE A
END
```

The above example will produce error message NAT1006 (value for variable index = 0) because, at the time the record is read (`READ`), the index `I` still contains the value 0.

In any case, the above example would not have printed the first 10 occurrences of `CREDIT-CARD` because the individual occurrence with the variable index cannot be contained in the array and the variable index (`I`) is only evaluated when the next record is read.

The following is the correct method of performing the above:

```
READ FINANCE
OBTAIN CREDIT-CARD (1-10)
FOR I 1 10
MOVE CREDIT-CARD (1.I) TO A (A20)
WRITE A
END
```

Invalid Example 2

```
READ FINANCE
FOR I 1 10
WRITE CREDIT-CARD (I)
END
```

The above example will produce error message NAT1006 because the index `I` is zero when the record is read in the `READ` statement.

The following is the correct method of performing the above:

```
READ FINANCE
FOR I 1 10
GET SAME
WRITE CREDIT-CARD (0030/I)
END
```

The `GET SAME` statement is necessary to reread the record after the variable index has been updated in the `FOR` loop.

Examples

Example 1 - OBTAIN Statement

```
** Example 'OBTEX1': OBTAIN
************************************************************************
RESET #INDEX (I1)
*
LIMIT 5
READ EMPLOYEES BY CITY
  OBTAIN SALARY (1:4)
  /*
  IF SALARY (4) GT 0 DO
    WRITE '=' NAME / 'SALARIES (1:4):' SALARY (1:4)
    FOR #INDEX 1 TO 4
      WRITE 'SALARY' #INDEX  SALARY (1.#INDEX)
    LOOP
    SKIP 1
  DOEND
LOOP
*
END
```

Output of Program OBTEX1:

```
Page      1                                                 05-02-08  13:37:48

NAME: SENKO
SALARIES (1:4):      31500      29900      28100      26600
SALARY     1      31500
SALARY     2      29900
SALARY     3      28100
SALARY     4      26600

NAME: HAMMOND
```

```
SALARIES (1:4):      22000      20200      18700      17500
SALARY    1      22000
SALARY    2      20200
SALARY    3      18700
SALARY    4      17500
```

Example 2 - OBTAIN Statement with Multiple Ranges

```
** Example 'OBTEX2': OBTAIN (with multiple ranges)
************************************************************************
RESET #INDEX (I1) #K (I1)
*
#INDEX  := 2
#K      := 3
*
LIMIT 2
*
READ EMPLOYEES BY CITY
  OBTAIN SALARY (1:5)
         SALARY (#INDEX:#INDEX+3)
  /*
  IF SALARY (5) GT 0 DO
    WRITE '=' NAME
    WRITE 'SALARIES (1-5):' SALARY (1:5) /
    WRITE 'SALARIES (2-5):' SALARY (#INDEX:#INDEX+3)
    WRITE 'SALARIES (2-5):' SALARY (#INDEX.1:4) /
    WRITE 'SALARY 3:' SALARY (3)
    WRITE 'SALARY 3:' SALARY (#K)
    WRITE 'SALARY 4:' SALARY (#INDEX.#K)
  DOEND
LOOP
```

Output of Program OBTEX2:

```
Page      1                                                  05-02-08  13:38:31

NAME: SENKO
SALARIES (1-5):      31500      29900      28100      26600      25200

SALARIES (2-5):      29900      28100      26600      25200
SALARIES (2-5):      29900      28100      26600      25200

SALARY 3:      28100
SALARY 3:      28100
SALARY 4:      26600
```

For further examples of using the `OBTAIN` statement, see *Referencing a Database Array* in the *Programming Guide*.

95 ON ERROR

Structured Mode Syntax

```
ON ERROR
 statement ...
END-ERROR
```

Reporting Mode Syntax

```
ON ERROR  { statement ...                }
          { DO statement ... DOEND       }
```

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

Related Statements: `DECIDE FOR` | `DECIDE ON` | `IF` | `IF SELECTION`

Function

The `ON ERROR` statement is used to intercept execution time errors which would otherwise result in a Natural error message, followed by termination of Natural program execution, and a return to command input mode.

When the `ON ERROR` statement block is entered for execution, the normal flow of program execution has been interrupted and cannot be resumed except for Natural error 3145 (record requested in hold), in which case a `RETRY` statement will cause processing to be resumed exactly where it was suspended.

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

Restriction

Only one `ON ERROR` statement is permitted in a Natural object.

Syntax Description

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td>statement...</td><td>Defining the ON ERROR Processing:

To define the processing that shall take place when an ON ERROR condition has been encountered, you can specify one or multiple statements.

Exiting from an ON ERROR Block:

An ON ERROR block may be exited by using a FETCH, STOP, TERMINATE, RETRY, ESCAPE ROUTINE or ESCAPE MODULE statement. If the block is not exited using one of these statements, standard error message processing is performed and program execution is terminated.</td></tr>
<tr><td>END-ERROR</td><td rowspan="2">End of ON ERROR Statement Block:

In structured mode, the Natural reserved word END-ERROR must be used to end an ON ERROR statement block.

In reporting mode, use the DO ... DOEND statements to supply one or multiple statements, and to end the ON ERROR statement. If you specify only a single statement, you can omit the DO ... DOEND statements. With respect to good coding practice, this is not recommended.</td></tr>
<tr><td>statement ...
DO statement ... DOEND</td></tr>
</table>

ON ERROR Processing within Objects on Different Levels

In an object call hierarchy created by means of CALLNAT, PERFORM or FETCH RETURN statements, each object may contain an ON ERROR statement.

When an error occurs, Natural will trace back the call hierarchy and select the first ON ERROR statement encountered in an object for execution.

For further information, see *Processing of Application Errors* in the *Programming Guide*.

System Variables

The following Natural system variables can be used in conjunction with the `ON ERROR` statement (as shown in the **Example** below):

System Variable	Explanation
`*ERROR-NR`	Contains the number of the error detected by Natural.
`*ERROR-LINE`	Contains the line number of the statement which caused the error.
`*PROGRAM`	Contains the name of the Natural object that is currently being executed.

Example

```
** Example 'ONEEX1': ON ERROR
**
**
CAUTION: Executing this example will modify the database records!
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
*
1 #NAME (A20)
1 #CITY (A20)
END-DEFINE
*
REPEAT
  INPUT 'ENTER NAME:' #NAME
  IF #NAME = ' '
    STOP
  END-IF
  FIND EMPLOY-VIEW WITH NAME = #NAME
    INPUT (AD=M) 'ENTER NEW VALUES:' ///
                 'NAME:' NAME /
                 'CITY:' CITY
    UPDATE
    END TRANSACTION
    /*
    ON ERROR
      IF *ERROR-NR = 3009
        WRITE 'LAST TRANSACTION NOT SUCCESSFUL'
            / 'HIT ENTER TO RESTART PROGRAM'
        FETCH 'ONEEX1'
      END-IF
```

```
      WRITE 'ERROR' *ERROR-NR 'OCCURRED IN PROGRAM' *PROGRAM
            'AT LINE' *ERROR-LINE
      FETCH 'MENU'
    END-ERROR
    /*
  END-FIND
END-REPEAT
END
```

96 OPEN CONVERSATION

```
OPEN CONVERSATION USING [SUBPROGRAMS] operand1 ...
```

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

Related Statements: `CLOSE CONVERSATION` | `DEFINE DATA CONTEXT`

Belongs to Function Group: ***Natural Remote Procedure Call***

Function

The statement `OPEN CONVERSATION` is used in conjunction with the Natural RPC (Remote Procedure Call). It allows the RPC Client to open a conversation and specify the remote subprograms to be included in the conversation.

When the `OPEN CONVERSATION` statement is executed, it assigns a unique ID identifying the conversation to the system variable `*CONVID`.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
operand1	C	S	A			A												yes	no

Syntax Element Description:

Syntax Element	Description
operand1	**Subprogram Names:** As *operand1* you specify the names of the remote subprograms to be included in the conversation. The name of a subprogram can be specified either as a constant of 1 to 8 characters, or as an alphanumeric variable of length 1 to 8.

Further Information and Examples

See the following sections in the *Natural RPC (Remote Procedure Call)* documentation:

- *Natural RPC Operation in Conversational Mode*
- *Using a Conversational RPC*

97 OPTIONS

```
OPTIONS parameter ...
```

Function

The `OPTIONS` statement can be used to specify compilation options as parameters for the current Natural object. These are the same options that can be specified within a Natural session with the `COMPOPT` system command.

Note: No mainframe-specific options are available. For compatibility reasons, for example, when programming a cross-platform application, such options are ignored during compile time.

Processing of Multiple OPTIONS Statements

If multiple `OPTIONS` statements are specified within the same Natural object, the option settings take effect immediately. However, this is not the case with the options `PSIGNF`, `TSENABL` and `GFID`. For these options, the option value specified with the *last* `OPTIONS` statement applies.

XII

98 PARSE XML

```
PARSE XML operand1 [INTO [PATH operand2] [NAME operand3] [VALUE operand4]]
 [[NORMALIZE] NAMESPACE operand5 PREFIX operand6]
   statement...
END-PARSE                         (structured mode only)
LOOP                              (reporting mode only)
```

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

Related Statement: REQUEST DOCUMENT

Belongs to Function Group: ***Internet and XML***

Function

The PARSE XML statement allows you to parse XML documents from a Natural program. See also *Statements for Internet and XML Access* in the *Programming Guide*.

It is recommended that you use dynamic variables when using the PARSE statement, because it is impossible to determine the length of a static variable. Using static variables could in turn lead to the truncation of the value that is to be written into the variable.

For information on Unicode support, see PARSE XML in the *Unicode and Code Page Support* documentation.

Mark-Up

The following are markings used in path strings to represent the different data types in an XML document (on ASCII-based systems):

Marking	XML Data	Location in Path String
?	Processing instruction (except for <?XML...?>)	end
!	Comment	end
C	CDATA section	end
@	Attribute (on mainframes: § or @, depending on session code page and terminal emulation)	before the attribute name
/	Closing tag and/or parent name separator in a path	end or between parent names
$	Parsed data - character data string	end

By using this additional markup in the path string, one can more easily identify the different elements of the XML document in the output document.

Global Namespace

To specify the global namespace, use a colon (:) as prefix and an empty URI.

Related System Variables

The following Natural system variables are automatically created for each `PARSE XML` statement issued:

- `*PARSE-TYPE`
- `*PARSE-LEVEL`
- `*PARSE-ROW`
- `*PARSE-COL`
- `*PARSE-NAMESPACE-URI`

The notation `(r)` after `*PARSE-TYPE`, `*PARSE-LEVEL`, `*PARSE-ROW`, `*PARSE-COL` and `*PARSE-NAMESPACE-URI` is used to indicate the label or statement number of the statement in which the `PARSE` was issued. If `(r)` is not specified, the corresponding system variable represents the system variable of the XML data currently being processed in the active `PARSE` processing loop.

For more information on these system variables, see the *System Variables* documentation.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S				A	U	B											yes	no
operand2		S				A	U	B											yes	yes
operand3		S				A	U	B											yes	yes
operand4		S				A	U	B											yes	yes
operand5		S	A			A	U	B											yes	yes
operand6		S	A			A	U	B											yes	yes

Syntax Element Description:

Syntax Element	Description
operand1	**XML Document:** *operand1* represents the XML document in question. The XML document may not be changed while it is being parsed. If you try to change the XML document during parsing (by writing into it, for example), an error message will be displayed.
operand2	**Path:** *operand2* represents the PATH of the data in the XML document. The PATH contains the name of the identified XML part, the names of all parents, as well as the type of the XML part. **Note:** The information given with PATH can be used to easily fill a tree view. See also ***Example 1 - Using** operand2*.
operand3	**Data Element Name:** *operand3* represents the NAME of a data element in the XML document. If NAME has no value, then the dynamic variable associated with it will be set to *length()=0, which is a static variable filled with a blank. See also ***Example 2 - Using** operand3*.
operand4	**Data Element Content:** *operand4* represents the content (VALUE) of a data element in the XML document. If there is no value, a given dynamic variable will be set to *length()=0, which is a static variable filled with a blank. See also ***Example 3 - Using** operand4*.
operand5 and *operand6* NORMALIZE NAMESPACE PREFIX	**Namespace URI and Prefix:** The NAMESPACE URI or Uniform Resource Identifier (*operand5*) and the namespace PREFIX (*operand6*) are copied during runtime. Therefore, modifying the namespace mapping arrays inside the PARSE XML loop will not affect the parser. *operand5* and *operand6* are one-dimensional arrays with an equal number of occurrences. Namespace normalization is a feature of the PARSE statement. XML is capable of defining namespaces for the element names: <myns:myentity xmlns:myns="http://myuri" /> The NAMESPACE definition consists of two parts: - a namespace PREFIX (which is, in this case, myns) and - a URI (myuri) to define the namespace.

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td></td><td>
The namespace <code>PREFIX</code> is part of the element name. This means, that for the <code>PARSE</code> statement, and especially for operand2, the generated <code>PATH</code> strings depend on the namespace <code>PREFIX</code>. If the path inside a Natural program is used to indicate specific tags, then this will fail if an XML document uses the correct <code>NAMESPACE</code> (URI), but with a different <code>PREFIX</code>.
With namespace normalization, all namespace <code>PREFIX</code>es can be set to defaults which have been defined in the <code>NAMESPACE</code> clause. The first entry will be the one used if a URI is specified more than once. If more than one <code>PREFIX</code> is used in the XML document, then only the first one will be taken into account for the output. The rest will be ignored.
The <code>NAMESPACE</code> clause contains pairs of namespace URIs and prefixes. For example:
<pre>uri(1) := 'http://namespaces.softwareag.com/natural/demo'
pre(1) := 'nat:'</pre>
If <code>NAMESPACE</code> is defined inside an XML document, the parser checks to see if that namespace (URI) exists in the normalization table. The prefix of the normalization table is used for all output data from the <code>PARSE</code> statement, instead of the namespace defined in the XML document.
See also:

Example 4 - Using operand5 and operand6
Example 5 - Using operand5 and operand6 with Namespace Normalization

Additional Information Concerning PREFIX:
In addition, the following applies to the prefix definition:

The prefix definition in the namespace normalization array always has to end in a colon (:), since this is the string that will be replaced.
A <code>PREFIX</code> or a URI may only occur once in a namespace normalization array.
If a <code>PREFIX</code> or the <code>NAMESPACE</code> URI contains trailing blanks (e.g. when using a static variable), the trailing blanks will be removed before the external parser is called.
If the <code>PREFIX</code> definition at the namespace normalization only contains a colon (:), the <code>NAMESPACE PREFIX</code> will be reduced to a colon (:).
If the <code>PREFIX</code> definition at the namespace normalization is empty, then the <code>NAMESPACE PREFIX</code> will be deleted.

</td></tr>
<tr><td><code>END-PARSE</code></td><td rowspan="2">
End of PARSE XML Statement:
In structured mode, the Natural reserved keyword <code>END-PARSE</code> must be used to end the <code>PARSE XML</code> statement.
In reporting mode, the Natural statement <code>LOOP</code> is used to end the <code>PARSE XML</code> statement.
</td></tr>
<tr><td><code>LOOP</code></td></tr>
</table>

Examples

Example 1 - Using operand2

The following XML code

```
myxml := '<?xml version="1.0" encoding="ISO-8859-1" ?>'-
         '<employee personnel-id="30016315" >'-
         '<full-name>'-
         '<!--this is just a comment-->'-
         '<first-name>RICHARD</first-name>'-
         '<name>FORDHAM</name>'-
         '</full-name>'-
         '</employee>'
```

processed by the following Natural code:

```
PARSE XML myxml INTO PATH mypath
  PRINT mypath
END-PARSE
```

produces the following output:

```
employee
employee/@personnel-id
employee/full-name
employee/full-name/!
employee/full-name/first-name
employee/full-name/first-name/$
employee/full-name/first-name//
employee/full-name/name
employee/full-name/name/$
employee/full-name/name//
employee/full-name//
employee//
```

Example 2 - Using operand3

The following XML code

```
myxml := '<?xml version="1.0" encoding="ISO-8859-1" ?>'-
         '<employee personnel-id="30016315" >'-
         '<full-name>'-
         '<!--this is just a comment-->'-
         '<first-name>RICHARD</first-name>'-
         '<name>FORDHAM</name>'-
         '</full-name>'-
         '</employee>'
```

processed by the following Natural code:

```
PARSE XML myxml INTO PATH mypath NAME myname
  DISPLAY (AL=39) mypath myname
END-PARSE
```

Note: produces the following output:

```
               MYPATH                                 MYNAME
---------------------------------    ---------------------------------



employee                             employee
employee/@personnel-id               personnel-id
employee/full-name                   full-name
employee/full-name/!
employee/full-name/first-name        first-name
employee/full-name/first-name/$
employee/full-name/first-name//      first-name
employee/full-name/name              name
employee/full-name/name/$
employee/full-name/name//            name
employee/full-name//                 full-name
employee//                           employee
```

Example 3 - Using operand4

The following XML code

```
myxml := '<?xml version="1.0" encoding="ISO-8859-1" ?>'-
         '<employee personnel-id="30016315" >'-
         '<full-name>'-
         '<!--this is just a comment-->'-
         '<first-name>RICHARD</first-name>'-
         '<name>FORDHAM</name>'-
         '</full-name>'-
         '</employee>'
```

processed by the following Natural code:

```
PARSE XML myxml INTO PATH mypath VALUE myvalue
  DISPLAY (AL=39) mypath myvalue
END-PARSE
```

produces the following output:

```
               MYPATH                                   MYVALUE
----------------------------------    -----------------------------------



employee
employee/@personnel-id                30016315
employee/full-name
employee/full-name/!                  this is just a comment
employee/full-name/first-name
employee/full-name/first-name/$       RICHARD
employee/full-name/first-name//
employee/full-name/name
employee/full-name/name/$             FORDHAM
employee/full-name/name//
employee/full-name//
employee//
```

Example 4 - Using operand5 and operand6

The following XML code

```
myxml := '<?xml version="1.0" encoding="ISO-8859-1" ?>'-
         '<nat:employee nat:personnel-id="30016315"'-
         ' xmlns:nat="http://namespaces.softwareag.com/natural/demo">'-
         '<nat:full-Name>'-
         '<nat:first-name>RICHARD</nat:first-name>'-
         '<nat:name>FORDHAM</nat:name>'-
         '</nat:full-Name>'-
         '</nat:employee>'
```

processed by the following Natural code:

```
PARSE XML myxml INTO PATH mypath
  PRINT mypath
END-PARSE
```

produces the following output:

```
nat:employee
nat:employee/@nat:personnel-id
nat:employee/@xmlns:nat
nat:employee/nat:full-Name
nat:employee/nat:full-Name/nat:first-name
nat:employee/nat:full-Name/nat:first-name/$
nat:employee/nat:full-Name/nat:first-name//
nat:employee/nat:full-Name/nat:name
nat:employee/nat:full-Name/nat:name/$
nat:employee/nat:full-Name/nat:name//
nat:employee/nat:full-Name//
nat:employee//
```

Example 5 - Using operand5 and operand6 with Namespace Normalization

Using `NORMALIZE NAMESPACE`, the same XML document as in Example 4 with a different `NAMESPACE PREFIX` would produce exactly the same output.

XML code:

```
myxml := '<?xml version="1.0" encoding="ISO-8859-1" ?>'-
         '<natural:employee natural:personnel-id="30016315"'-
         ' xmlns:natural="http://namespaces.softwareag.com/natural/demo">'-
         '<natural:full-Name>'-
         '<natural:first-name>RICHARD</natural:first-name>'-
         '<natural:name>FORDHAM</natural:name>'-
         '</natural:full-Name>'-
         '</natural:employee>'
```

Natural code:

```
uri(1) := 'http://namespaces.softwareag.com/natural/demo'
pre(1) := 'nat:'
*
PARSE XML myxml INTO PATH mypath NORMALIZE NAMESPACE uri(*) PREFIX pre(*)
  PRINT mypath
END-PARSE
```

Output of above program:

```
nat:employee
nat:employee/@nat:personnel-id
nat:employee/@xmlns:nat
nat:employee/nat:full-Name
nat:employee/nat:full-Name/nat:first-name
nat:employee/nat:full-Name/nat:first-name/$
nat:employee/nat:full-Name/nat:first-name//
nat:employee/nat:full-Name/nat:name
nat:employee/nat:full-Name/nat:name/$
nat:employee/nat:full-Name/nat:name//
nat:employee/nat:full-Name//
nat:employee//
```

99 PASSW

```
PASSW=operand1
```

Related Statements: ACCEPT/REJECT | AT BREAK | AT START OF DATA | AT END OF DATA | BACKOUT TRANSACTION | BEFORE BREAK PROCESSING | DELETE | END TRANSACTION | FIND | HISTOGRAM | GET | GET SAME | GET TRANSACTION | LIMIT | PERFORM BREAK PROCESSING | READ | RETRY | STORE | UPDATE

Belongs to Function Group: ***Database Access and Update***

Function

The PASSW statement is used to specify a default password for access to Adabas or VSAM files which have been password-protected.

Note: This password can be overwritten using the PASSWORD clause of the database access statements FIND, GET, HISTOGRAM, READ, STORE.

Natural Security Considerations

In the security profile of a library, you can specify a default Adabas password (as described in the *Natural Security* documentation); this password applies to all database access statements for which neither an individual password is specified nor a PASSW statement applies. It applies within the library in whose security profile it is specified, and also remains in effect in other libraries you subsequently log on to and in whose security profiles no password is specified.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S				A													yes	no

Syntax Element Description:

Syntax Element	Description
operand1	**Password:** The password (*operand1*) may be specified as an alphanumeric constant or the content of an alphanumeric variable. It may consist of up to 8 characters, and must not contain special characters or embedded blanks. If the password is specified as a constant, it must be enclosed in apostrophes. The password specified with the PASSW statement applies to all database access statements (FIND, GET, HISTOGRAM, READ, STORE) for which no individual password is specified. It remains in effect until another password is specified in the execution of a subsequent PASSW statement or the Natural session is terminated. A password specified with a specific database access statement applies only to that statement, not to any subsequent statement.

100 PERFORM

```
PERFORM            [ operand1  [ (AD= { M } ) ] ]
subroutine-name    [                  { O }     ]
                   [                  { A }     ]
                   [ nX                         ] ...
```

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

Related Statements: `CALL` | `CALL FILE` | `CALL LOOP` | `CALLNAT` | `DEFINE SUBROUTINE` | `ESCAPE` | `FETCH`

Belongs to Function Group: *Invoking Programs and Routines*

Function

The `PERFORM` statement is used to invoke a Natural **subroutine**.

Nested PERFORM Statements

The invoked subroutine may contain a `PERFORM` statement to invoke another subroutine (the number of nested levels is limited by the size of the required memory).

A subroutine may invoke itself (recursive subroutine). If database operations are contained within an external subroutine that is invoked recursively, Natural will ensure that the database operations are logically separated.

Parameter Transfer with Dynamic Variables

See the statement `CALLNAT`.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A	G		A	U	N	P	I	F	B	D	T	L	C	G	O	yes	yes

Syntax Element Description:

Syntax Element	Description
subroutine-name	**Subroutine to be Invoked:** For a subroutine name (maximum 32 characters), the same naming conventions apply as for user-defined variables. The subroutine name is independent of the name of the module in which the subroutine is defined (it may but need not be the same). The subroutine to be invoked must be defined with a `DEFINE SUBROUTINE` statement. It may be an inline or external subroutine (see `DEFINE SUBROUTINE` statement). Within one object, no more than 50 external subroutines may be referenced. **Data Available in a Subroutine** ■ **Inline Subroutines** No explicit parameters can be passed from the invoking object to an inline subroutine. An inline subroutine has access to the currently established global data area as well as the local data area defined within the same object module. ■ **External Subroutines** An external subroutine has access to the currently established global data area. Moreover, parameters can be passed with the `PERFORM` statement from the invoking object to the external subroutine (see *operand1*); thus, you may reduce the size of the global data area.
operand1	**Parameters to be Passed:** When an external subroutine is invoked with the `PERFORM` statement, one or more parameters (*operand1*) can be passed with the `PERFORM` statement from the invoking object to the external subroutine. For an inline subroutine, *operand1* cannot be specified. If parameters are passed, the structure of the parameter list must be defined in a `DEFINE DATA` statement. By default, the parameters are passed "by reference", that is, the data are transferred via address parameters, the parameter values themselves are not moved. However, it is also possible to pass parameters "by value", that is, pass the actual parameter values. To do so, you define these fields in the `DEFINE DATA PARAMETER` statement of the subroutine with the option `BY VALUE` or `BY VALUE RESULT`. ■ If parameters are passed "by reference" the following applies: The sequence, format and length of the parameters in the invoking object must match exactly the sequence, format and length of the `DEFINE DATA PARAMETER` structure of the invoked subroutine. The names of the variables in the invoking object and the subroutine may be different. ■ If parameters are passed "by value" the following applies: The sequence of the parameters in the invoking object must match exactly the sequence in the `DEFINE DATA PARAMETER` structure of the invoked subroutine. Formats and lengths of the

Syntax Element	Description
	variables in the invoking object and the subroutine may be different; however, they have to be data transfer compatible. The names of the variables in the invoking object and the subroutine may be different. If parameter values that have been modified in the subroutine are to be passed back to the invoking object, you have to define these fields with `BY VALUE RESULT`. With `BY VALUE` (without `RESULT`) it is not possible to pass modified parameter values back to the invoking object (regardless of the `AD` specification; see also **below**). **Note:** With `BY VALUE`, an internal copy of the parameter values is created. The subroutine accesses this copy and can modify it, but this will not affect the original parameter values in the invoking object. With `BY VALUE RESULT`, an internal copy is likewise created; however, after termination of the subroutine, the original parameter values are overwritten by the (modified) values of the copy. For both ways of passing parameters, the following applies: ■ In the parameter data area of the invoked subroutine, a redefinition of groups is only permitted within a `REDEFINE` block. ■ If an array is passed, its number of dimensions and occurrences in the subroutine's parameter data area must be same as in the `PERFORM` parameter list. **Note:** If multiple occurrences of an array that is defined as part of an indexed group are passed with the `PERFORM` statement, the corresponding fields in the subroutine's parameter data area must not be redefined, as this would lead to the wrong addresses being passed. **Note:** Numeric constant parameters are internally represented in packed form (format P). For further information see the *Programming Guide* > *Numeric Constants*.
`AD=`	**Attributes:** If *`operand1`* is a variable, you can mark it in one of the following ways: `AD=O` — Non-modifiable, see session parameter `AD=O`. **Note:** Internally, `AD=O` is processed in the same way as `BY VALUE` (see **Note** under *`operand1`*). `AD=M` — Modifiable, see session parameter `AD=M`. This is the default setting. `AD=A` — Input only, see session parameter `AD=A`. If *`operand1`* is a constant, `AD` cannot be explicitly specified. For constants, `AD=O` always applies.
n`X`	**Parameters to be Skipped:** With the notation *n*`X` you can specify that the next *n* parameters are to be skipped (for example, `1X` to skip the next parameter, or `3X` to skip the next three parameters); this means that for the next *n* parameters no values are passed to the external subroutine.

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

Examples

- Example 1 - PERFORM as Inline Subroutine
- Example 2 - PERFORM as External Subroutine

Example 1 - PERFORM as Inline Subroutine

```
** Example 'PEREX1': PERFORM   (as inline subroutine)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 ADDRESS-LINE  (A20/2)
  2 PHONE
*
1 #ARRAY    (A75/1:4)
1 REDEFINE #ARRAY
  2 #ALINE  (A25/1:4,1:3)
1 #X        (N2) INIT <1>
1 #Y        (N2) INIT <1>
END-DEFINE
*
LIMIT 5
FIND EMPLOY-VIEW WITH CITY = 'BALTIMORE'
  MOVE NAME            TO #ALINE (#X,#Y)
  MOVE ADDRESS-LINE(1) TO #ALINE (#X+1,#Y)
  MOVE ADDRESS-LINE(2) TO #ALINE (#X+2,#Y)
  MOVE PHONE           TO #ALINE (#X+3,#Y)
  IF  #Y = 3
    RESET INITIAL #Y
    /*
    PERFORM PRINT
    /*
  ELSE
    ADD 1 TO #Y
  END-IF
  AT END OF DATA
    /*
    PERFORM PRINT
    /*
  END-ENDDATA
END-FIND
```

```
*
DEFINE SUBROUTINE PRINT
  WRITE NOTITLE (AD=OI) #ARRAY(*)
  RESET #ARRAY(*)
  SKIP 1
END-SUBROUTINE
*
END
```

Output of Program `PEREX1`:

```
JENSON                   LAWLER                   FORREST
2120 HASSELL             4588 CANDLEBERRY AVE     37 TENNYSON DRIVE
 #206                    BALTIMORE                BALTIMORE
998-5038                 629-0403                 881-3609

ALEXANDER                NEEDHAM
409 SENECA DRIVE         12609 BUILDERS LANE
BALTIMORE                BALTIMORE
345-3690                 641-9789
```

Example 2 - PERFORM as External Subroutine

Program containing `PERFORM` statement:

```
** Example 'PEREX2': PERFORM (as external subroutine)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 ADDRESS-LINE  (A20/2)
  2 PHONE
*
1 #ALINE   (A25/1:4,1:3)
1 #X       (N2)            INIT <1>
1 #Y       (N2)            INIT <1>
END-DEFINE
*
LIMIT 5
*
FIND EMPLOY-VIEW WITH CITY = 'BALTIMORE'
  MOVE NAME            TO #ALINE (#X,#Y)
  MOVE ADDRESS-LINE(1) TO #ALINE (#X+1,#Y)
  MOVE ADDRESS-LINE(2) TO #ALINE (#X+2,#Y)
  MOVE PHONE           TO #ALINE (#X+3,#Y)
  IF  #Y = 3
    RESET INITIAL #Y
    /*
    PERFORM PEREX2E #ALINE(*,*)
    /*
  ELSE
```

```
    ADD 1 TO #Y
  END-IF
  AT END OF DATA
    /*
    PERFORM PEREX2E #ALINE(*,*)
    /*
  END-ENDDATA
END-FIND
*
END
```

External subroutine PEREX3 with parameters called by program PEREX2:

```
** Example 'PEREX3': SUBROUTINE (external subroutine with parameters)
************************************************************************
DEFINE DATA
PARAMETER
1 #ALINE  (A25/1:4,1:3)
END-DEFINE
*
DEFINE SUBROUTINE PEREX2E
  WRITE NOTITLE (AD=OI) #ALINE(*,*)
  RESET #ALINE(*,*)
  SKIP 1
END-SUBROUTINE
*
END
```

Output of Program PEREX2:

```
JENSON                    LAWLER                    FORREST
2120 HASSELL              4588 CANDLEBERRY AVE      37 TENNYSON DRIVE
 #206                     BALTIMORE                 BALTIMORE
998-5038                  629-0403                  881-3609

ALEXANDER                 NEEDHAM
409 SENECA DRIVE          12609 BUILDERS LANE
BALTIMORE                 BALTIMORE
345-3690                  641-9789
```

101 PERFORM BREAK PROCESSING

```
PERFORM BREAK [PROCESSING] [(r)]
  AT BREAK statement ...
```

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

Related Statements: ACCEPT/REJECT | AT BREAK | AT START OF DATA | AT END OF DATA | BACKOUT TRANSACTION | BEFORE BREAK PROCESSING | DELETE | END TRANSACTION | FIND | GET | GET SAME | GET TRANSACTION DATA | HISTOGRAM | LIMIT | PASSW | READ | RETRY | STORE | UPDATE

Belongs to Function Group: ***Database Access and Update***

Function

The PERFORM BREAK PROCESSING statement is used to establish break processing in loops created by FOR, REPEAT, CALL LOOP and CALL FILE statements where no automatic break processing is established, or whenever a user-initiated break processing is desired. Unlike automatic break processing which is executed immediately after the record is read, the PERFORM BREAK PROCESSING statement is executed when it is encountered in the normal flow of the program.

This statement causes a check for a break processing condition (based on the value of a control field) and also results in the evaluation of Natural system functions. This check and system function evaluation are performed each time the statement is encountered for execution. This statement may be executed depending on a condition specified in an IF statement.

Syntax Description

Syntax Element	Description
(*r*)	**Statement Reference Notation:** By default, the final PERFORM BREAK condition is true at the end of execution of the program, subprogram or subroutine. The notation (*r*) may be used to relate the final processing of a PERFORM BREAK to a specific loop. In this case the PERFORM BREAK is executed in the loop end handling of this loop; after the final automatic BREAK processing and before the AT END OF DATA statements are executed.
AT BREAK *statement...*	See the syntax of the AT BREAK statement.

Example

```
** Example 'PBPEX1S': PERFORM BREAK PROCESSING  (structured mode)
************************************************************************
DEFINE DATA LOCAL
1 #INDEX (N2)
1 #LINE  (N2) INIT <1>
END-DEFINE
*
FOR #INDEX 1 TO 18
  PERFORM BREAK PROCESSING
  /*
  AT BREAK OF #INDEX /1/
    WRITE NOTITLE / 'PLEASE COMPLETE LINES 1-9 ABOVE' /
    RESET INITIAL #LINE
  END-BREAK
  /*
  WRITE NOTITLE '_' (64) '=' #LINE
  ADD 1 TO #LINE
END-FOR
*
END
```

Output of Program PBPEX1S:

```
________________________________________________________________ #LINE:   1
________________________________________________________________ #LINE:   2
________________________________________________________________ #LINE:   3
________________________________________________________________ #LINE:   4
________________________________________________________________ #LINE:   5
________________________________________________________________ #LINE:   6
________________________________________________________________ #LINE:   7
________________________________________________________________ #LINE:   8
________________________________________________________________ #LINE:   9

PLEASE COMPLETE LINES 1-9 ABOVE

________________________________________________________________ #LINE:   1
________________________________________________________________ #LINE:   2
________________________________________________________________ #LINE:   3
________________________________________________________________ #LINE:   4
________________________________________________________________ #LINE:   5
________________________________________________________________ #LINE:   6
________________________________________________________________ #LINE:   7
________________________________________________________________ #LINE:   8
________________________________________________________________ #LINE:   9

PLEASE COMPLETE LINES 1-9 ABOVE
```

Equivalent reporting-mode example: PBPEX1R.

102 PRINT

```
PRINT [(rep)] [NOTITLE] [NOHDR] [(statement-parameters)]
{ [ nX ]     { 'text' [(attributes)]             } }
{ [ nT ] ... { 'c'(n) [(attributes)]             } } ...
{ [ /  ]     { ['='] operand1 [(parameters)]     } }
```

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

Related Statements: AT END OF PAGE | AT TOP OF PAGE | CLOSE PRINTER | DEFINE PRINTER | DISPLAY | EJECT | FORMAT | NEWPAGE | SKIP | SUSPEND IDENTICAL SUPPRESS | WRITE | WRITE TITLE | WRITE TRAILER

Belongs to Function Group: ***Creation of Output Reports***

Function

The PRINT statement is used to produce output in free format.

The PRINT statement differs from the WRITE statement in the following aspects:

- The output for each operand is written according to the value content rather than the length of the operand. Leading zeros for numeric values and trailing blanks for alphanumeric values are suppressed. The session parameter AD defines whether numeric values are printed left or right justified. With AD=L, the trailing blanks of a numeric value are suppressed. With AD=R, the leading blanks of a numeric value are printed.
- If the resulting output exceeds the current line size (LS parameter), the output is continued on the next line as follows: An alphanumeric constant or the content of an alphanumeric variable (without edit mask) is split at the rightmost blank or character which is neither a letter nor a numeric character contained on the current line. The first part of the split value is output to the current line, and the second part is written to the next line. Leading blanks in the second part are removed. As a consequence, empty lines are suppressed.

  For all other operands, the entire value is written to the next line.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats														Referencing Permitted	Dynamic Definition
operand1		S	A	G	N	A	U	N	P	I	F	B	D	T	L		G	O		yes	no

Syntax Element Description:

Syntax Element	Description
(*rep*)	**Report Specification:** The notation (*rep*) may be used to specify the identification of the report for which the PRINT statement is applicable. A value in the range 0 - 31 or a logical name which has been assigned using the DEFINE PRINTER statement may be specified. If (*rep*) is not specified, the PRINT statement will apply to the first report (Report 0). If this printer file is defined to Natural as PC, the report will be downloaded to the PC, see ***Example 2***. For information on how to control the format of an output report created with Natural, see *Report Format and Control* (in the *Programming Guide*).
NOTITLE	**Default Page Title Suppression:** Natural generates a single title line for each page resulting from a PRINT statement. This title contains the page number, the time of day, and the date. Time of day is set at the beginning of the session (TP mode) or at the beginning of the job (batch mode). This default title line may be overridden by using a WRITE TITLE statement, or it may be suppressed by specifying the NOTITLE clause in the PRINT statement. Examples: ■ Default title will be produced: `PRINT NAME` ■ User title will be produced:

Syntax Element	Description

	`PRINT NAME WRITE TITLE 'user-title'`
	
	■ No title will be produced:
	
	`PRINT NOTITLE NAME`
	
	If the NOTITLE option is used, it applies to all DISPLAY, PRINT and WRITE statements within the same object which write data to the same report.

NOHDR	**Column Header Suppression:** The PRINT statement itself does not produce any column headers. However, if you use the PRINT statement in conjunction with a DISPLAY statement, you can use the NOHDR option of the PRINT statement to suppress the column headers generated by the DISPLAY statement. The NOHDR option only takes effect if the execution of the PRINT statement causes a new page to be output. Without the NOHDR option, the column headers (if any) of the DISPLAY statement would be output on this new page; with NOHDR they will not.
statement-parameters	**Parameter Definition at Statement Level:** One or more parameters, enclosed within parentheses, may be specified at statement level, that is, immediately after the PRINT statement or an element being displayed. Each parameter specified in this manner will override any previous parameter specified in a GLOBALS command, SET GLOBALS (in Reporting Mode only) or FORMAT statement. If more than one parameter is specified, the parameters must be separated from one another by one or more blanks. A parameter entry must not be split between two statement lines. The parameter settings applied here will only be regarded for variable fields, but they have no effect on text-constants. If you would like to set field attributes for a text-constant, they have to be set explicitly for this element, see *Parameter Definition at Element (Field) Level*. See also: ■ *List of Parameters* ■ *Example of Parameter Usage at Statement and Element (Field) Level*
*n*X, *n*T, /	**Field Positioning, Text, Attribute Assignment:** See *Field Positioning, Text, Attribute Assignment* below.

List of Parameters

Parameters that can be specified with the PRINT statement		Specification (S = at statement level, E = at element level)
AD	Attribute Definition	SE
AL	Alphanumeric Length for Output	SE
CD	Color Definition	SE
CV	Control Variable	SE
DF	Date Format	SE
DL	Display Length for Output	SE
DY	Dynamic Attributes	SE
EM	Edit Mask	SE
EMU	Unicode Edit Mask	E
FL	Floating Point Mantissa Length	SE
MC	Multiple-Value Field Count	S
MP	Maximum Number of Pages of a Report	S
NL	Numeric Length for Output	SE
PC	Periodic Group Count	S
PM	Print Mode	SE
SG	Sign Position	SE
ZP	Zero Printing	SE

The individual session parameters are described in the *Parameter Reference*.

Example of Parameter Usage at Statement and Element (Field) Level

```
DEFINE DATA LOCAL
1 VARI (A4)     INIT <'1234'>                  /*     Output
END-DEFINE                                     /*    Produced
*                                              /*    ---------
PRINT           'Text'          VARI           /*    Text 1234
PRINT (PM=I)    'Text'          VARI           /*    Text 4321
PRINT           'Text' (PM=I)   VARI (PM=I)    /*    txeT 4321
PRINT           'Text' (PM=I)   VARI           /*    txeT 1234
END
```

Field Positioning, Text, Attribute Assignment

```
{ [ nX ]      { 'text' [(attributes)]                    } }
{ [ nT ] ...  { 'c'(n) [(attributes)]                    } } ...
{ [ /  ]      { ['='] operand1  [(parameters)]           } }
```

Field Positioning Notations

Syntax Element	Description
*n*X	**Column Spacing:** This notation inserts *n* spaces between columns. `PRINT NAME 5X SALARY`
*n*T	**Tab Setting:** The *n*T notation causes positioning (tabulation) to print position *n*. Backward positioning results in a line advance. In the following example, `NAME` is printed beginning in position 25, and `SALARY` is printed beginning in position 50: `PRINT 25T NAME 50T SALARY`
/	**Line Advance - Slash Notation:** When placed between fields or text elements, a slash (/) causes positioning to the beginning of the next print line. `PRINT NAME / SALARY`

Text/Attribute Assignment

Syntax Element	Description
'*text*'	**Text Assignment:** The character string enclosed by single quotes is displayed.

Syntax Element	Description
	`PRINT 'EMPLOYEE' NAME 'MARITAL/STATUS' MAR-STAT`
`'c' (n)`	**Character Repetition:** The character *c* enclosed by single quotes is displayed *n* times immediately before the field value. `PRINT '*' (5) '=' NAME`
`'='`	**Field Content Positioned behind Field Heading:** When placed before a field, the equal sign `'='` results in the display of the field heading (as defined in the `DEFINE DATA` statement or in the DDM) followed by the field contents. `PRINT '=' NAME`
operand1	**Field to be Printed:** As *operand1* you specify the field to be printed.
parameters	**Parameter Definition at Element (Field) Level:** One or more parameters (see **table** above), enclosed within parentheses, may be specified immediately after *operand1*. Each parameter specified in this manner will override any previous parameter specified **at statement level** or in a `GLOBALS` command, `SET GLOBALS` (in Reporting Mode only) or `FORMAT` statement. If more than one parameter is specified, one or more blanks must be placed between each entry. An entry must not be split between two statement lines. See also: ■ *Statement Parameters* ■ *Example of Parameter Usage at Statement and Element (Field) Level*

Output Attributes

attributes indicates the output attributes to be used for text display. Attributes can be:

```
{ AD=ad-value ...  }
{ CD=cd-value      }
{ PM=pm-value ...  } ...

{ ad-value }
{ cd-value } ...
```

Where:

`ad-value`, `cd-value` and `pm-value` denote the possible values of the corresponding session parameters `AD`, `CD` and `PM` described in the relevant sections of the *Parameter Reference* documentation.

The compiler actually accepts more than one attribute value for an output field. For example, you can specify: `AD=BDI`. In such a case, however, only the last value applies. In the given example, only the value `I` becomes effective and the output field is displayed intensified.

For an alphanumeric/Unicode constant (Natural data format A or U), you can specify `ad-value` and/or `cd-value` without preceding `CD=` or `AD=`, respectively. The single value entered is then checked against all possible `CD` values first. For example: a value of `IRE` will be interpreted as intensified/red but not as intensified/right-justified/mandatory. You cannot combine a single `cd-value` or `ad-value` with a value preceded by `CD=` or `AD=`.

Example

- Example 1 - PRINT Statement
- Example 2 - PRINT Statement with Report to be Downloaded to the PC

Example 1 - PRINT Statement

```
** Example 'PRTEX1': PRINT
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 JOB-TITLE
  2 ADDRESS-LINE (2)
END-DEFINE
*
LIMIT 1
READ EMPLOY-VIEW BY CITY
  /*
  WRITE NOTITLE 'EXAMPLE 1:'
             // 'RESULT OF WRITE STATEMENT:'
```

```
  WRITE       /  NAME  ',' FIRST-NAME ':' JOB-TITLE '*' (30)
  WRITE       / 'RESULT OF PRINT STATEMENT:'
  PRINT       /  NAME  ',' FIRST-NAME ':' JOB-TITLE '*' (30)
  /*
  WRITE      // 'EXAMPLE 2:'
             // 'RESULT OF WRITE STATEMENT:'
  WRITE       /  NAME 60X ADDRESS-LINE (1:2)
  WRITE       / 'RESULT OF PRINT STATEMENT:'
  PRINT       /  NAME 60X ADDRESS-LINE (1:2)
  /*
END-READ
END
```

Output of Program PRTXEX1:

```
EXAMPLE 1:

RESULT OF WRITE STATEMENT:

SENKO                , WILLIE              : PROGRAMMER
******************************

RESULT OF PRINT STATEMENT:

SENKO , WILLIE : PROGRAMMER ******************************


EXAMPLE 2:

RESULT OF WRITE STATEMENT:

SENKO
2200 COLUMBIA PIKE    #914

RESULT OF PRINT STATEMENT:

SENKO                                                            2200 COLUMBIA
PIKE  #914
```

Example 2 - PRINT Statement with Report to be Downloaded to the PC

```
** Example 'PCPIEX1': PRINT to PC
**
** NOTE: Example requires that Natural Connection is installed.
************************************************************************
DEFINE DATA LOCAL
01 PERS VIEW OF EMPLOYEES
  02 PERSONNEL-ID
  02 NAME
  02 CITY
END-DEFINE
```

```
*
FIND PERS WITH CITY = 'NEW YORK'                   /* Data selection
  PRINT (7) 5T CITY 20T NAME 40T PERSONNEL-ID      /* (7) designates
                                                   /* the output file
                                                   /* (here the PC).
END-FIND
END
```

103 PROCESS

```
PROCESS view-name USING {operand1=operand2}, ... [GIVING operand3 ...]
```

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

Function

The `PROCESS` statement is used in conjunction with Entire System Server. Entire System Server allows you to use various operating system facilities such as reading and writing files, VTOC and catalog management, JES queues, etc.

See the section *Getting Started* in the *Entire System Server User's Guide* for further information on the `PROCESS` statement and its individual clauses.

Restriction

This statement is only available with Entire System Server.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S				A		N	P			B							yes	no
operand2	C	S				A	U	N	P			B							yes	no
operand3		S				A		N	P			B							yes	no

Syntax Element Description:

Syntax Element	Description
view-name	**View Name:** Name of the view used by Entire System Server.
`USING`	**USING Clause:** The `USING` clause is used to pass parameters to the Entire System Server processor. This is done by assigning a value (*operand2*) to a field (*operand1*) in a view defined to Entire System Server. See the Entire System Server documentation for view description.

Syntax Element	Description
	Note: Multiple specifications of *operand1*=*operand2* must be separated either by the input delimiter character (as specified with the session parameter `ID`) or by a comma.
`GIVING`	**GIVING Clause:** The `GIVING` clause is used to specify the fields (*operand3*) for which values are to be returned by the Entire System Server processor. Each field must be defined in a view used by Entire System Server.

104 PROCESS COMMAND

CHECK | EXEC | TEXT | HELP Syntax:

```
PROCESS COMMAND ACTION
                  USING   PROCESSOR-NAME=operand1
 ⎧  CHECK  ⎫              COMMAND-LINE (index[:index])=operand2
 ⎨  EXEC   ⎬      GIVING  RESULT-FIELD (index[:index])    (see Syntax Note)
 ⎪  TEXT   ⎪              RETURN-CODE
 ⎩  HELP   ⎭              [NATURAL-ERROR]
```

GET Syntax:

```
PROCESS COMMAND ACTION
 GET USING   PROCESSOR-NAME=operand1
             GETSET-FIELD-NAME=operand3
     GIVING  GETSET-FIELD-VALUE   (see Syntax Note)
             [NATURAL-ERROR]
```

SET Syntax:

```
PROCESS COMMAND ACTION
 SET    USING   PROCESSOR-NAME=operand1
                GETSET-FIELD-NAME=operand3
                GETSET-FIELD-VALUE=operand4
       [GIVING  NATURAL-ERROR]  (see Syntax Note)
```

CLOSE Syntax:

```
PROCESS COMMAND ACTION
 CLOSE     [GIVING NATURAL-ERROR]  (see Syntax Note)
```

Syntax Note:

The `GIVING` option is only required in the reporting mode and if no `VIEW OF COMMAND` has been defined in the `DEFINE DATA` statement.

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

Belongs to Function Group: ***Invoking Programs and Routines***

Function

Once a Natural Command Processor has been created using the Natural utility `SYSNCP`, it can be invoked from a Natural program using the `PROCESS COMMAND` statement.

For details on how to create a Natural Command Processor, refer to the *SYSNCP Utility* documentation.

DDM Used for Command Processing

Important: The word `COMMAND` in the `PROCESS COMMAND` statement is in fact the name of a view. The name of the view that is used need not necessarily be `COMMAND`; however, we recommend the use of `COMMAND` because there exists a DDM (data definition module) that is also called `COMMAND`. This DDM must be referenced within the `DEFINE DATA` statement, for example `COMMAND VIEW OF COMMAND`.

The DDM `COMMAND` has been created specifically for use in conjunction with the `PROCESS COMMAND` statement:

```
DB:    1 File:    1 - COMMAND                          Default Sequence: ?

  TYL  DB  NAME                             F LENG  S D   REMARKS
  ---  --  -------------------------------- - ----  - -  ------------------------
    1  AA  PROCESSOR-NAME                   A    8  N D DE    USING
  M 1  AB  COMMAND-LINE                     A   80  N D MU/DE USING
    1  AF  GETSET-FIELD-NAME                A   32  N D DE    USING
    1  BA  NATURAL-ERROR                    N  4.0  N         GIVING
    1  BB  RETURN-CODE                      A    4  N         GIVING
  M 1  BC  RESULT-FIELD                     A   80  N   MU    GIVING
    1  BD  GETSET-FIELD-VALUE               A   32  N D       USING; GIVING
  ***** DDM OUTPUT TERMINATED ******
```

The fields contained in the DDM correspond to the fields used in the `PROCESS COMMAND` statement. They are explained in ***Syntax Element Description***.

Note: To avoid possible compilation or runtime errors, make sure that the DDM named `COMMAND` is cataloged as type `C` (field `DDM Type` on the SYSDDM Menu) before you use it. (If you re-catalog the DDM, any DBID/FNR specification in the `SYSDDM` utility will be ignored.)

Security Considerations

With Natural Security, it is possible to restrict the usage of certain keywords and/or functions which are defined in a Command Processor. Keywords and/or functions can be allowed/disallowed for a specific user or group of users. See the *Natural Security* documentation for details.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S				A													no	no
operand2	C	S	A	G		A		N											no	no
operand3	C	S				A		N											no	no
operand4	C	S				A		N	P	I									no	no

Syntax Element Description:

Syntax Element	Description
`CHECK`	**CHECK Action:** `CHECK` is used as a precautionary measure to determine if a command is executable with the statement `PROCESS COMMAND EXEC`. It works as follows: for the given Command Processor name, a runtime check is performed in two steps: ▪ It is checked whether the Command Processor exists in the current library or one of its steplibs; ▪ The content of the command line `COMMAND-LINE (1)` is analyzed to determine whether it is acceptable. In addition, the runtime action definitions `R`, `M` and `1-9` are written into `RESULT-FIELD (1:9)`. If the field `NATURAL-ERROR` is specified in the view or in the `GIVING` option, it returns the error code. If this field is not available and the command analysis fails, a Natural system error occurs. **Note:** No `CHECK` is required if you want to perform an `EXEC` action. The `CHECK` is included in an `EXEC` operation.
`EXEC`	**EXEC Action:** `EXEC` works exactly the same as `CHECK` with the addition that the runtime actions are executed as specified in the runtime action editor. Only `COMMAND-LINE (1)` is needed. You can use up to 9 occurrences of `RESULT-FIELD` (however, for optimum performance, you should not use more occurrences than you really need). **Note:** `EXEC` is the only action which can be used to leave the currently active program. This is the case when the runtime action definition contains a `FETCH` or `STOP` statement.

Syntax Element	Description
	See also *Example 1 - PROCESS COMMAND ACTION EXEC*.
TEXT	**TEXT Action:** TEXT delivers general information about the Command Processor and text associated with a keyword or function. For further information, see the following sections under *Input Values for TEXT Actions*: ■ *TEXT for General Information* ■ *TEXT for Keyword Information* ■ *TEXT for Function Information* **Note:** To access texts for keywords and functions, you must have specified Y in the field Catalog user texts on the Processor Header Maintenance 3 screen of the SYSNCP utility, see the section *Miscellaneous Options - Header 3*.
HELP	**HELP Action:** HELP returns a list of all valid keywords, synonyms, and functions for the purpose of, for example, the creation of online help windows. This list is contained in the field(s) of RESULT-FIELD. The type of help returned is dependent on the content of the command lines. ■ COMMAND-LINE (1) must contain the search criteria. ■ COMMAND-LINE (2), if specified, must contain the start value or a search value. ■ COMMAND-LINE (3), if specified, must contain a start value. For further information, see the following sections under *Input Values for HELP Actions*: ■ *HELP for Keywords* ■ *HELP for Synonyms* ■ *HELP for Global Functions* ■ *HELP for Local Functions* ■ *HELP for IKN* ■ *HELP for IFN* **Note:** For optimum performance, the number of occurrences of the field RESULT-FIELD should not exceed the number of lines to be displayed on the screen. At least one occurrence must be used.
GET	**GET Action:** GET reads internal Command Processor information and current Command Processor settings from the dynamically allocated NCPWORK buffer.
SET	**SET Action:** SET modifies internal Command Processor settings in the NCPWORK buffer.

Syntax Element	Description
CLOSE	**CLOSE Action:** CLOSE terminates the use of the Command Processor and releases the Command Processor buffer. When the Command Processor is used during a session and is not released with CLOSE, then there exists a buffer named NCPWORK in your thread. The runtime part of the Command Processor requires this buffer; it can be released using the statement PROCESS COMMAND ACTION CLOSE. If any PROCESS COMMAND statement follows this statement, then the Command Processor buffer will be opened again. See also ***Example 2 - PROCESS COMMAND ACTION CLOSE***.
GIVING	**GIVING Option:** This option is only required in reporting mode and if no VIEW OF COMMAND has been defined in the DEFINE DATA statement. The GIVING option is not available in structured mode, because there exists an implicit GIVING option made up of all fields specified in the DEFINE DATA statement, which are usually referenced in the GIVING option for the reporting mode. This means that in structured mode all field defined in the GIVING option must be defined in the DEFINE DATA statement. **Note:** Specified in the GIVING option are fields to be filled by the Command Processor as a result of the processing of any action.
PROCESSOR-NAME	The name of the Command Processor to be used for processing. The Command Processor specified must be cataloged.
COMMAND-LINE	The command line to be processed by a CHECK or an EXEC action, or the keyword/command for which user text or help text is to be returned to the program by a TEXT or HELP action. Note that this field can contain more than one occurrence.
RESULT-FIELD	Contains information resulting from the use of options that can be specified within a runtime action defined for a Command Processor function (see Runtime Actions in the Natural SYSNCP utility). Note that this field can contain more than one occurrence.
RETURN-CODE	The return code of an operation resulting from an EXEC or a CHECK action as specified within a Runtime Actions definition (see the Natural SYSNCP utility).
NATURAL-ERROR	The Natural error returned for a PROCESS COMMAND action. We recommend that you use this field in the DEFINE DATA statement as it returns the Natural error code for the Command Processor. When the field is absent, Natural runtime error processing is triggered if an error occurs.

Syntax Element	Description
GETSET-FIELD-NAME	The name of the constant or variable that is read when a GET action is performed or that is written with a SET action. For a list of possible values for GETSET-FIELD-NAME, see *Input Values for GETSET-FIELD-NAME*.
GETSET-FIELD-VALUE	The value of the constant or variable specified in the field GETSET-FIELD-NAME which is read when a GET action is performed or which written with a SET action.

This section covers the following topics:

- Input Values for GETSET-FIELD-NAME
- Input Values for TEXT Actions
- Input Values for HELP Actions

Input Values for GETSET-FIELD-NAME

The following values can be used for the GETSET-FIELD-NAME field (A32):

Field Name	Format	G/S*	Content
NAME	A8	G	Name of current Command Processor.
LIBRARY	A8	G	Loaded from library.
FNR	N10	G	Loaded from file.
DBID	N10	G	Loaded from database.
TIMESTMP	A8	G	Time stamp of the current Command Processor.
COUNTER	N10	G	Access counter.
BUFFER-LENGTH	N10	G	Bytes allocated for NCPWORK.
C-DELIMITER	A1	G/S	Multiple command delimiter.
DATA-DELIMITER	A1	G	Delimiter to precede data.
PF-KEY	A1	G/S	PF key may be command (Y/N).
UPPER-CASE	A1	G	Keywords in upper case (Y/N).
UQ-KEYWORDS	A1	G	Keywords unique (Y/N).
IMPLICIT-KEYWORD	A1	G/S	Identifier for implicit keyword entry.
MIN-LEN	N10	G	Minimum length of keywords.
MAX-LEN	N10	G	Maximum length of keywords.
KEYWORD-SEQ	A8	G/S	Keyword sequence.
ALT-KEYWORD-SEQ	A8	G/S	Alternative keyword sequence.
USER-SEQUENCE	A1	G	User may override KEYWORD-SEQ (Y/N).
CURR-LOCATION	N10	G/S	Current location (IFN).
CURR-IKN1	N10	G/S	IKN1 of current location.
CURR-IKN2	N10	G/S	IKN2 of current location.

Field Name	Format	G/S*	Content
CURR-IKN3	N10	G/S	IKN3 of current location.
CHECK-LOCATION	N10	G	Last checked location (IFN).
CHECK-IKN1	N10	G	IKN1 of CHECK-LOCATION.
CHECK-IKN2	N10	G	IKN2 of CHECK-LOCATION.
CHECK-IKN3	N10	G	IKN3 of CHECK-LOCATION.
TOP-IKN1	N10	G	IKN1 of topmost keyword.
TOP-IKN2	N10	G	IKN2 of topmost keyword.
TOP-IKN3	N10	G	IKN3 of topmost keyword.
KEY1-TOTAL	N10	G	Number of keywords of type 1.
KEY2-TOTAL	N10	G	Number of keywords of type 2.
KEY3-TOTAL	N10	G	Number of keywords of type 3.
FUNCTIONS-TOTAL	N10	G	Number of cataloged functions.
LOCAL-GLOBAL-SEQ	A8	G/S	Local/global function validation.
ERROR-HANDLER	A8	G/S	General error program.
SECURITY	A1	G	Natural Security installed (Y/N).
SEC-PREFETCH	A1	G	Natural Security data are to be read (Y/N) or have been read (D = done).
PREFIX1	A1	G	Corresponds to the field Prefix Character 1 on the Processor Header Maintenance 2 screen of the SYSNCP utility, see the section *Keyword Editor Options - Header 2*.
PREFIX2	A1	G	Corresponds to the field Prefix Character 2 on the Processor Header Maintenance 2 screen.
HEX1	A1	G	Corresponds to the field Hex. Replacement 1 on the Processor Header Maintenance 2 screen.
HEX2	A1	G	Corresponds to the field Hex. Replacement 2 on the Processor Header Maintenance 2 screen.
DYNAMIC	A32	G	Dynamic part (:*n*:) of last error message.
LAST	-	G	Last command placed on top of stack as data.
LAST-ALL	-	G	Last commands placed on top of stack as data.
LAST-COM	-	G	Last command moved to *COM.
MULTI	-	G	Places the last of multiple commands as data on top of the stack.
MULTI-COM	-	G	Places the last of multiple commands in the system variable *COM.

*G = Field name can be used with the GET **action**.
*S = Field name can be used with the SET **action**.

Input Values for TEXT Actions

The following input values are provided to return different information from a `TEXT` action:

TEXT for General Information

For general information, `COMMAND-LINE (*)`; that is, all command lines, must be blank. Up to nine fields of `RESULT-FIELD` are returned containing the following information:

RESULT-FIELD	Format	Contents
1	Text (A40)	Header 1 for User Text
2	Text (A40)	Header 2 for User Text
3	Text (A16)	"First Entry used as" text
4	Text (A16)	"Second Entry used as" text
5	Text (A16)	"Third Entry used as" text
6	Numeric (N3)	Number of Entry 1 Keywords
7	Numeric (N3)	Number of Entry 2 Keywords
8	Numeric (N3)	Number of Entry 3 Keywords
9	Numeric (N7)	Number of Cataloged Functions

TEXT for Keyword Information

For keyword information, `COMMAND-LINE (1)` must contain the corresponding keyword; `COMMAND-LINE (2)` can optionally contain the keyword type (1, 2, 3 or P); `COMMAND-LINE (3:6)` must be empty.

RESULT-FIELD	Contents	Format
1	Keyword comment text	Text (A40)
2	Keyword in full length	Text (A16)
3	Keyword in unique short form	Text (A16)
4	"Keyword used as" entry	Text (A16)
5	Internal keyword number (IKN)	Numeric (N4)
6	Minimum length of keyword	Numeric (N2)
7	Maximum length of keyword	Numeric (N2)
8	Keyword type (1, 2, 3, 1S, 2S, 3S, P)	Text (A2)

TEXT for Function Information

For function information, `COMMAND-LINE (1:3)` must contain the keywords which specify the wanted location. `COMMAND-LINE (4:6)` contains the keywords which specify the wanted function. For example, if information about the global command `ADD USER` is to be returned, the command lines 1, 2, 3, and 6 must be blank; the command line 4 must contain the text string `ADD`, and the command line 5 must contain the text string `USER`.

RESULT-FIELD	Format	Contents
1	Text (A40)	Text as defined with the option T in runtime action definition.
2	Numeric (N10)	Internal function number (IFN) of the specified location.
3	Numeric (N10)	Internal function number (IFN) of the specified function.

Input Values for HELP Actions

The following input values are provided to return different information from a `HELP` action:

HELP for Keywords

This action returns an alphabetically sorted list of keywords and/or synonyms with their internal keyword numbers (IKN).

Command Line	Contents	
1	Must begin with indicator `K`.	
	The types of keywords to be returned:	
	`*`	Keywords of all types
	`1`	Keywords with type 1
	`2`	Keywords with type 2
	`3`	Keywords with type 3
	`P`	Keywords with type P (parameter)
	Options:	
	`I`	Return IKN in addition to keywords.
	`T`	Show keyword partially in upper case (to show possible abbreviation).
	`S`	Return synonyms in addition to keywords.
	`X`	Return only synonyms of specified keywords.
	`A`	Internal keywords are also returned.
	`+`	Search does not include start value.
2	Start value for the keyword search (optional). By default, the search begins with the start value. However, if you specify the plus (+) option, the search does not include the start value itself, but begins with the next higher value.	

The field `RESULT-FIELD (1:n)` returns the specified list.

Examples:

```
Command Line 1:    K*X     Returns all synonyms of all keyword types.
```

```
Command Line 1:    K123S   Returns all keywords of type 1, 2 and 3 including ↩
synonyms.
```

HELP for Synonyms

For a given IKN, this action returns the original keyword and all synonyms.

Command Line	Contents	
1	Must begin with the indicator `S`.	
	Option:	
	`T`	Shows keyword partially in upper case (to show possible abbreviation).
2	Internal Keyword Number (IKN) of the keyword in format N4.	

The field `RESULT-FIELD (1)` returns the original keyword. The fields `RESULT-FIELD (2:n)` return associated synonyms for this keyword.

Example:

Input:	Output:

`Command Line 1:    S`
`Command Line 2:    1003`
	`Result-Field 1:    Edit`
	`Result-Field 2:    Maintain`
	`Result-Field 3:    Modify`

HELP for Global Functions

This action returns a list of all global functions.

Command Line	Contents	
1	Must begin with the indicator `G`.	
	Options:	
	`I`	Internal Function Number (IFN) is also returned.
	`T`	Shows keyword partially in upper case (to show possible abbreviation).
	`S`	The keywords returned in `RESULT-FIELD` will be aligned in columns.
	`A`	Internal keywords are also returned.
	`1`	Only functions containing the given keyword of type 1 are to be returned.

Command Line	Contents	
	2	Only functions containing the given keyword of type 2 are to be returned.
	3	Only functions containing the given keyword of type 3 are to be returned.
	+	Search does not include start value.
2	Start value for global function search. Keywords must be given in sequence `123`. By default, the search begins with the start value. However, if you specify the plus (+) option, the search does not include the start value itself, but begins with the next higher value.	
3	Must be blank.	
4	To search only for global functions with a specific keyword, you specify the keyword here. If you specify a keyword, you also have to specify the keyword type (1, 2 or 3) as option (see above).	

The field `RESULT-FIELD (1:n)` returns the specified list.

Example:

Input:	Output:

`Command Line 1:    G`
`Command Line 2:    ADD`
	`Result-Field 1:    ADD CUSTOMER`
	`Result-Field 2:    ADD FILE`
	`Result-Field 3:    ADD USER`

HELP for Local Functions

This action returns a list of all local functions for a specified location.

Command Line	Contents	
1	Must begin with the indicator `L`.	
	Options:	
	I	Internal Function Number (IFN) is also returned.
	T	Shows keyword partially in upper case (to show possible abbreviation).
	S	The keywords returned in `RESULT-FIELD` will be aligned in columns.
	A	Internal keywords are also returned.
	1	Only functions containing given keyword of type 1 are to be returned.
	2	Only functions containing given keyword of type 2 are to be returned.

Command Line	Contents	
	3	Only functions containing given keyword of type 3 are to be returned.
	C	Only those functions are returned which are defined for the current location (command line 3 is ignored).
	F	Invoke "recursive" listing of local functions; that is, all local commands that lead to the current/specified location will be returned.
2	Start value for local function search (optional). Keywords must be given in sequence `123`.	
3	The location for which the list is to be returned. Keywords must be given in sequence `123`. If no location is specified, the current location of the Command Processor will be used.	
4	Keyword restriction (optional): If you specify a keyword, or an IKN with the format N4, only functions with this keyword will be returned.	

The field `RESULT-FIELD (1:n)` returns the specified list.

HELP for IKN

For any given internal keyword numbers (IKN), this action returns the original keyword.

Command Line	Contents	
1	Must start with `IKN`.	
	Options:	
	A	The internal keyword will be shown.
	T	Shows keyword partially in upper case (to show possible abbreviation).
2	The IKN to be translated, in format N4.	

The field `RESULT-FIELD (1)` returns the keyword.

Example:

Input:	Output:
`Command Line 1:     IKN` `Command Line 2:     0000002002`	`Result-Field 1:     CUSTOMER`

HELP for IFN

For any given internal function numbers (IFN), this action returns the keywords of a function.

Command Line	Contents	
1	Must start with IFN.	
	Option:	
	A	Functions with internal keywords will not be suppressed.
2	The IFN to be translated, in format N10.	
3	**Further options:**	
	S	Keywords belonging to the IFN will be returned in `RESULT-FIELD (1:3)`.
	T	Shows keywords partially in upper case (to show possible abbreviations).
	L	IFN will be returned if IFN is used as a location.
	C	IFN will be returned if IFN is used as a command.

The field `RESULT-FIELD(1)` returns the function; if option `S` is used, the function is returned in `RESULT-FIELD (1:3)`.

Example:

Input:	Output:
`Command Line 1:     IFN` `Command Line 2:     0001048578`	`Result-Field 1:     DISPLAY INVOICE`

Examples

In addition to the example programs shown in this section, you can find example programs in the `SYSNCP` system library. These programs all begin with `EXAM`.

You can test all available `PROCESS COMMAND` actions by executing the `EXAM` program in `SYSNCP`. You can then choose an action from a menu.

- Example 1 - PROCESS COMMAND ACTION EXEC

- Example 2 - PROCESS COMMAND ACTION CLOSE

Example 1 - PROCESS COMMAND ACTION EXEC

```
/* EXAM-EXS - Example for PROCESS COMMAND ACTION EXEC (Structured Mode)
/***********************************************************************
DEFINE DATA LOCAL
  01 COMMAND VIEW OF COMMAND
     02 PROCESSOR-NAME
     02 COMMAND-LINE (1)
     02 NATURAL-ERROR
     02 RETURN-CODE
     02 RESULT-FIELD (1)
  01 MSG (A65) INIT <'Please enter a command.'>
END-DEFINE
/*
REPEAT
  INPUT (AD=MIT' ' IP=OFF) WITH TEXT MSG
    'Example for PROCESS COMMAND ACTION EXEC (Structured Mode)' (I)
  / 'Command ==>' COMMAND-LINE (1) (AL=64)
  /*******
 PROCESS COMMAND ACTION EXEC
    USING
      PROCESSOR-NAME = 'DEMO'
      COMMAND-LINE (1) = COMMAND-LINE (1)
  /*******
  COMPRESS 'NATURAL-ERROR =' NATURAL-ERROR TO MSG
END-REPEAT
END
```

Example 2 - PROCESS COMMAND ACTION CLOSE

```
/* EXAM-CLS - Example for PROCESS COMMAND ACTION CLOSE (Structured Mode)
/************************************************************************
DEFINE DATA LOCAL
  01 COMMAND VIEW OF COMMAND
END-DEFINE
/*
PROCESS COMMAND ACTION CLOSE
/*
DEFINE WINDOW CLS
INPUT WINDOW = 'CLS'
  'NCPWORK has just been released.'
/*
END
```

105 PROCESS PAGE

Function

The `PROCESS PAGE` statement constitutes a general interface description to an external rendering engine, such as Natural for Ajax, thus linking the Natural internal data representation with an external data representation. Via this link, data and events, but no rendering information, are sent to and returned from an external, browser-based application.

For further information, refer to the *Natural for Ajax* documentation. The latest Natural for Ajax documentation is always available at ***https://empower.softwareag.com/***.

Syntax 1 - PROCESS PAGE

```
PROCESS PAGE [(parameter)] operand1
 [WITH PARAMETERS
  {[NAME] operand3 [VALUE] operand4 [(parameters)]} ...
 END-PARAMETERS]
 [GIVING operand11]
```

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

Belongs to Function Group: ***Screen Generation for Interactive Processing***

Syntax Description - Syntax 1

Syntax 1 of the `PROCESS PAGE` statement is normally only used inside a Natural adapter. An adapter is a Natural object that forms the interface between Natural application code and web page. It is automatically created/updated by Natural for Ajax when the layout is saved.

Note:

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S				A	U												yes	no
operand2		S	A													C			no	no
operand3	C	S				A	U												yes	no
operand4	C	S	A			A	U	N	P	I	F	B	D	T	L				yes	yes
operand5		S	A													C			no	no

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand11		S								I4									yes	yes

Syntax Element Description:

Syntax Element	Description
parameter	**Attribute Control Variable(s):** The parameter `CV`, enclosed within parentheses, may be specified to reference one or more attribute control variables as specified in *operand2*: `(CV=operand2)` See also *Logical Condition Criteria, MODIFIED Option - Check whether Field Content has been Modified* in the *Programming Guide*.
operand1	**External Page Layout Name:** *operand1* contains the name of the external page layout.
operand2	**Name of Attribute Control Variable(s):** *operand2* contains the name of the attribute control variable, must be of format C and must be either a scalar or a single array occurrence.
operand3	**Name(s) of external Data Field(s):** *operand3* contains the name(s) of the external data field(s) *operand4* will be transferred to/from.
operand4	**Name(s) of Natural Data Field(s):** *operand4* contains the name(s) of the Natural data field(s) which will be transferred.
parameters	**Parameters:** One or more parameters, enclosed within parentheses, may be specified immediately after *operand4*: `EM` or `EMU`: Edit mask used during data transfer. For further information, see the session parameter `EM` in the *Parameter Reference*. For details on Unicode edit masks, see the session parameter `EMU` in the *Parameter Reference*.

<table>
<tr><th>Syntax Element</th><th colspan="2">Description</th></tr>
<tr><td></td><td><code>CV</code></td><td>The parameter <code>CV</code>, enclosed within parentheses, may be specified immediately after operand4 to reference one or more attribute control variables as specified in operand5:

<code>(CV=operand5)</code>

See also Logical Condition Criteria, MODIFIED Option - Check whether Field Content has been Modified in the Programming Guide.</td></tr>
<tr><td>operand5</td><td colspan="2">Name of Attribute Control Variable:
operand5 contains the name of the attribute control variable. The variable must be of format C.

If operand4 is a scalar or a single array occurrence, operand5 must be
■ a scalar
■ or a single array occurrence.

If operand4 is the full range of an array of dimension 1, operand5 must be
■ a scalar
■ or a single array occurrence
■ or the full range of an array of dimension 1 with the same size.

If operand4 is the full range of an array of dimension 2, operand5 must be
■ a scalar
■ or a single array occurrence
■ or the full range of an array of dimension 2 with the same size in both dimensions
■ or the full range of an array of dimension 1 with the same size that operand4 has in dimension 1.

If operand4 is the full range of an array of dimension 3, operand5 must be
■ a scalar
■ or a single array occurrence
■ or the full range of an array of dimension 3 with the same size in all three dimensions
■ or the full range of an array of dimension 2 with the same size that operand4 has in dimension 1 and 2
■ or the full range of an array of dimension 1 with the same size that operand4 has in dimension 1.</td></tr>
<tr><td><code>GIVING</code>
operand11</td><td colspan="2">GIVING Clause:

operand11 contains the Natural error if the request could not be performed.</td></tr>
</table>

Example of an adapter which has been created by Natural for Ajax:

```
* PAGE1: PROTOTYPE       --- CREATED BY Natural for Ajax ---
* PROCESS PAGE USING 'XXXXXXXX' WITH
* INFOPAGENAME RESULT YOURNAME
DEFINE DATA PARAMETER
1 INFOPAGENAME (U) DYNAMIC
1 RESULT (U) DYNAMIC
1 YOURNAME (U) DYNAMIC
END-DEFINE
*
PROCESS PAGE U'/njxdemos/helloworld' WITH
PARAMETERS
 NAME U'infopagename'
  VALUE INFOPAGENAME
 NAME U'result'
  VALUE RESULT
 NAME U'yourname'
  VALUE YOURNAME
END-PARAMETERS
*
*  TODO: Copy to your calling program and implement.
/*/*( DEFINE EVENT HANDLER
* DECIDE ON FIRST *PAGE-EVENT
*  VALUE U'nat:page.end'
*   /* Page closed.
*   IGNORE
*  VALUE U'onHelloWorld'
*   /* TODO: Implement event code.
*   PROCESS PAGE UPDATE FULL
*  NONE VALUE
*   /* Unhandled events.
*   PROCESS PAGE UPDATE
* END-DECIDE
/*/*) END-HANDLER
*
END
```

Syntax 2 - PROCESS PAGE USING

```
PROCESS PAGE USING operand6
[         WITH {operand7} ...  ]
          NO PARAMETER
GIVING operand11]
```

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

Belongs to Function Group: *Screen Generation for Interactive Processing*

Syntax Description - Syntax 2

This syntax is used to perform rich GUI input/output processing using an object of type adapter that has been generated from a page layout created with Natural for Ajax or a similar tool.

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand6	C	S				A													yes	no
operand7		S	A	G		A	U	N	P	I	F	B	D	T	L				yes	yes
operand11		S								I4									yes	yes

Syntax Element Description:

Syntax Element	Description
`USING` *operand6*	**Adapter Name:** Invokes an adapter definition which has been previously stored in a Natural system file. See also *Processing a Rich GUI Page - Adapter* in the *Programming Guide*. The adapter name (*operand6*) may be a 1 to 8 character alphanumeric constant or user-defined variable. If a variable is used, it must have been defined previously. The adapter name may contain an ampersand (&); at execution time, this character will be replaced by the current value of the Natural system variable `*LANGUAGE`. This feature is provided for historical reasons. If you need multi-lingual adapters, use the capability of the external rendering system (for example, Natural for Ajax). **Note:** New applications do not need the ampersand feature to be multilingual. Pages designed, for example, using Natural for Ajax, can hold multilingual information as part of the layout design. See *Multi Language Management* in the *Natural for Ajax* documentation.
operand7	**Field Specification:** A list of database fields and/or user-defined variables, all of which must have been defined previously. The fields must agree in number, sequence, format, length and (for arrays) number of occurrences with the fields in the referenced adapter; otherwise, an error occurs. When the content of a database field is modified as a result of `PROCESS PAGE` processing, only the value as contained in the data area is modified. In order to change the content of the database, appropriate database `UPDATE`/`STORE` statements must be used. See *PROCESS PAGE USING Fields Defined in the Program*.
`NO PARAMETER`	**NO PARAMETER Option:**

Syntax Element	Description
	See *PROCESS PAGE USING without Parameter List*.
GIVING *operand11*	**GIVING Clause:** *operand11* contains the Natural error if the request could not be performed. **Note:** The GIVING clause interrupts the common Natural error handling, if an error occurs while the adapter object is being activated or executed. Instead of back-tracking the Natural modules in order to find an ON ERROR clause, the Natural error code is passed to this variable and execution is continued with the next statement.

PROCESS PAGE USING without Parameter List

The following requirements must be met when PROCESS PAGE USING is used without parameter list:

- The adapter name (*operand6*) must be specified as an alphanumeric constant (up to 8 characters).
- The adapter used in this manner must have been created prior to the compilation of the program which references the adapter.
- The names of the fields to be processed are taken dynamically from the adapter source definition at compilation time. The field names used in both program and adapter must be identical.
- All fields to be referenced in the PROCESS PAGE statement must be accessible at that point.
- In structured mode, fields must have been defined previously (database fields must be properly referenced to processing loops or views).
- When the page layout is changed, the programs using the adapter need not be recataloged. However, when array structures or names, formats/lengths of fields are changed, or fields are added/deleted in the adapter, the programs using the adapter must be recataloged.
- The adapter source must be available at program compilation; otherwise, the PROCESS PAGE USING statement cannot be compiled.

Note: If you wish to compile the program even if the adapter is not yet available, specify NO PARAMETER. The PROCESS PAGE USING statement can then be compiled even if the adapter is not yet available.

PROCESS PAGE USING Fields Defined in the Program

By specifying the names of the fields to be processed within the program (*operand7*), it is possible to have the names of the fields in the program differ from the names of the fields in the adapter.

The sequence of fields in the program must match the sequence in the adapter. If you use Natural maps as adapter objects, note that the map editor sorts the fields as specified in the map in alphabetical order by field name. For more information, see the map editor description in your *Editors* documentation.

When the layout of the adapter is changed, the program using the adapter does not need to be recataloged. However, when field names, field formats/lengths, or array structures in the adapter are changed or fields are added or deleted in the adapter, the program must be recataloged.

A check is made at execution time to ensure that the format and length of the fields as specified in the program match the fields as specified in the adapter. If both layouts do not agree, an error message is produced.

Syntax 3 - PROCESS PAGE UPDATE

```
PROCESS PAGE UPDATE
    [ FULL ]
    [ DATA ]   [event-option] [GIVING operand11]
```

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

Belongs to Function Group: ***Screen Generation for Interactive Processing***

Syntax Description - Syntax 3

The PROCESS PAGE UPDATE statement is used to return to and re-execute a PROCESS PAGE statement. It is generally used to return from event processing, because the data input processing of the preceding PROCESS PAGE statement was incomplete.

Note: No INPUT, WRITE, PRINT or DISPLAY statements may be executed between a PROCESS PAGE statement and its corresponding PROCESS PAGE UPDATE statement.

The PROCESS PAGE UPDATE statement, when executed, repositions the program status regarding subroutine, special condition and loop processing as it existed when the PROCESS PAGE statement was executed (as long as the status of the PROCESS PAGE statement is still active). If the loop was initiated after the execution of the PROCESS PAGE statement and the PROCESS PAGE UPDATE statement is within this loop, the loop will be discontinued and then restarted after the PROCESS PAGE statement has been reprocessed as a consequence of the PROCESS PAGE UPDATE statement.

If a hierarchy of subroutines was invoked after the execution of the PROCESS PAGE statement, and the PROCESS PAGE UPDATE is performed within a subroutine, Natural will trace back all subroutines automatically and reposition the program status to that of the PROCESS PAGE statement.

It is not possible, however, to have a PROCESS PAGE statement positioned within a loop, a subroutine or a special condition block, and then execute the PROCESS PAGE UPDATE statement when the status under which the PROCESS PAGE statement was executed has already been terminated. An error message will be produced and program execution terminated when this error condition is detected.

Operand Definition Table:

Operand	Possible Structure	Possible Formats	Referencing Permitted	Dynamic Definition
operand11	S	I4	yes	yes

Syntax Element Description:

Syntax Element	Description
`FULL`	**FULL Option:** If you specify the `FULL` option in a `PROCESS PAGE UPDATE` statement, the corresponding `PROCESS PAGE` statement will be re-executed fully: ■ With an ordinary `PROCESS PAGE UPDATE` statement (without `FULL` option), the contents of variables that were changed between the `PROCESS PAGE` and `PROCESS PAGE UPDATE` statement will not be displayed; that is, all variables on the screen will show the contents they had when the `PROCESS PAGE` statement was originally executed. ■ With a `PROCESS PAGE UPDATE FULL` statement, all changes that have been made after the initial execution of the `PROCESS PAGE` statement will be applied to the `PROCESS PAGE` statement when it is re-executed; that is, all variables on the screen contain the values they had when the `PROCESS PAGE UPDATE` statement was executed. The `MODIFIED` status of all control variables is reset. A characteristic of the `PROCESS PAGE UPDATE FULL` statement is that the status of attribute control variables is reset to `NOT MODIFIED`. This is not done with the ordinary `PROCESS PAGE UPDATE` statement. To check if an attribute control variable has been assigned the status `MODIFIED`, use the `MODIFIED` option.
`DATA`	**DATA Option:** The `DATA` option behaves like the `FULL` option, with the exception that the `MODIFIED` status of the control variables is *not* reset.
event-option	**EVENT Option:** See ***EVENT Option*** below.
`GIVING` (*operand11*)	**GIVING Clause:** *operand11* contains the Natural error if the request could not be performed.

Example User Program Fragment:

```
PROCESS PAGE USING "HELLOW-A"
*
/*( DEFINE EVENT HANDLER
DECIDE ON FIRST *PAGE-EVENT
 VALUE U'nat:page.end'
  /* Page closed.
  IGNORE
 VALUE U'onHelloWorld'
  COMPRESS "HELLO WORLD" YOURNAME INTO RESULT
  PROCESS PAGE UPDATE FULL
```

```
 NONE VALUE
  /* Unhandled events.
  PROCESS PAGE UPDATE
END-DECIDE
/*) END-HANDLER
```

EVENT Option

```
AND SEND EVENT operand8
 [WITH PARAMETERS
  {[NAME] operand9 [VALUE] operand10 [ { (EMU=value) } ]}...
                                       { (EM=value)  }
 END-PARAMETERS]
```

With this option, you can advise the external I/O system to run specific functions. These functions are part of the external I/O system or implement special functions regarding the output processing as setting of focus, displaying message boxes, etc.

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand8	C	S				A	U												yes	no
operand9	C	S				A	U												yes	no
operand10	C	S	A			A	U	N	P	I	F	B	D	T	L				yes	yes

Syntax Element Description:

Syntax Element	Description
`AND SEND EVENT` *operand8*	**Event Requested from the External I/O System:** Depending on the implementation of the external I/O system, events are available, refer to *Sending Events to the User Interface* in the *Natural for Ajax* documentation.
`WITH PARAMETERS`	**WITH PARAMETERS Clause:** With this clause, you can specify the following:
`NAME` *operand9*	**External Data Field Name:** *operand9* contains the external name of the data fields *operand10* will be transferred to/from.
`VALUE` *operand10*	**Natural Data Fields:** *operand10* contains the Natural data fields which will be transferred.

Syntax Element	Description
EMU= EM=	**Edit Mask:** Edit mask used during data transfer. For details on edit masks, see the session parameter EM in the *Parameter Reference*. For details on Unicode edit masks, see the session parameter EMU in the *Parameter Reference*.
END-PARAMETERS	**End of WITH PARAMETERS Clause:** The Natural reserved word END-PARAMETERS must be used to end the WITH PARAMETERS clause.

Syntax 4 - PROCESS PAGE MODAL

```
PROCESS PAGE MODAL
 statement ...
END-PROCESS
```

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

Related Statements: PROCESS PAGE

Belongs to Function Group:

- **Screen Generation for Interactive Processing**

Syntax Description - Syntax 4

The PROCESS PAGE MODAL statement is used to initiate a processing block and to control the lifetime of a modal rich GUI window.

Entering the PROCESS PAGE MODAL statement block causes the following actions to be performed:

- Data from Report 0, which is not displayed yet, will be displayed first;
- the system variable *PAGE-LEVEL is incremented;
- the opening of a modal page is prepared. The physical opening of the modal page will be performed with the next PROCESS PAGE USING *operand6* WITH statement, where *operand6* is the name of the adapter to be used.

Leaving the PROCESS PAGE MODAL statement block causes the following actions to be performed:

- If a modal page has been opened for this level, the closing of the modal page is prepared. The physical closing of the modal page will be performed with the next `PROCESS PAGE UPDATE [FULL]` statement;
- the system variable `*PAGE-LEVEL` is decremented, and the system variable `*PAGE-EVENT` is set back to the value it had before the statement block was entered;
- a `nat:page.default` event will be raised in the program that opened the modal page.

Note: No `PRINT`, `WRITE`, `INPUT` or `DISPLAY` statements referring to Report 0 may be executed between a `PROCESS PAGE MODAL` statement and its corresponding `END-PROCESS` statement.

The `PROCESS PAGE MODAL` statement is not valid in batch mode.

Syntax Element Description:

Syntax Element	Description
statement	**Statement(s) to be Executed:** In place of *statement*, you must supply one or several suitable statements, depending on the situation. If you do not want to supply a specific statement, you may insert the `IGNORE` statement.
END-PROCESS	**End of PROCESS PAGE MODAL Statement:** The Natural reserved word `END-PROCESS` must be used to end the `PROCESS PAGE MODAL` statement.

Example:

```
* Name: First Demo/Open modal!
*
PROCESS PAGE USING "EMPTY-A"
*
/*( DEFINE EVENT HANDLER
DECIDE ON FIRST *PAGE-EVENT
  VALUE U'nat:page.end', U'onClose'
    /* Page closed.
    IGNORE
  VALUE U'onNextLevel'
    PROCESS PAGE MODAL
      FETCH RETURN "EMPTY-P"
    END-PROCESS
    PROCESS PAGE UPDATE
  NONE VALUE
    PROCESS PAGE UPDATE
END-DECIDE
/*) END-HANDLER
END
```

Examples

Further examples of using the `PROCESS PAGE` statement are contained in library `SYSEXNJX`.

106 PROCESS SQL (SQL)

```
PROCESS SQL ddm-name <<statement-string>>
```

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

Related Statement: `REQUEST DOCUMENT`

Belongs to Function Group: ***Internet and XML***

Function

The `PROCESS SQL` statement is used to issue SQL statements to the underlying database.

Syntax Description

Syntax Element	Description
ddm-name	**DDM Name:** The name of a data definition module (DDM) must be specified to provide the "address" of the database which executes the stored procedure. For more information, see *ddm-name*.
statement-string	**Statement String:** The statements which can be specified in the *statement-string* are the same statements which can be issued with the SQL statement `EXECUTE`; see also ***Flexible SQL***. **Caution:** To avoid transaction synchronization problems between the Natural environment and the underlying database, the `COMMIT` and `ROLLBACK` statements must not be used within `PROCESS SQL`. The statement string can cover several statement lines without any continuation character to be specified. Comments at the end of a line as well as entire comment lines are possible. The statement string can also include parameters; see ***Parameters in Statement String*** below.

Parameters in Statement String

```
[ :U ]  :host-variable [INDICATOR:host-variable] [LINIDICATOR:host-variable]
[ :G ]
```

Unlike with the ***Parameters*** described in the section ***Basic Syntactical Items***, the `host-variables` used in this context must be prefixed by a colon (:). In addition, they can be preceded by a further qualifier (`:U` or `:G`).

See further details on `host-variable`.

Syntax Element Description:

Syntax Element	Description
`:U:host-variable`	**"USING" Variable:** The prefix `:U` qualifies the host variable as a so-called "USING" variable. Such a variable indicates that its value is to be *passed to* the database. `:U` is the default specification.
`:G:host-variable`	**"GIVING" Variable:** The prefix `:G` qualifies the host variable as a so-called "GIVING" variable. Such a variable indicates that it is to *receive* a value *from* the database.

Entire Access Options

With Entire Access, you can also specify the following as `statement-string`:

- `SET SQLOPTION option = value`
- `SQLCONNECT option = value`
- `SQLDISCONNECT`

These options are only possible with Entire Access, and are described in the section *Accessing Data in an SQL Database* (in the *Programming Guide*).

Examples

Example for Adabas D:

```
PROCESS SQL ADABAS_D_DDM << LOCK TABLE EMPLOYEES IN SHARE MODE >>
```

Example of Calling a Procedure Stored in Adabas D:

The called procedure computes the sum of two numbers.

```
...
COMPUTE #N1 = 1
COMPUTE #N2 = 2
COMPUTE #SUM = 0
...
PROCESS SQL ADABAS_D_DDM << DBPROCEDURE DEMO.SUM (:#N1, :#N2, :G:#SUM) >>
...
WRITE #N1 '+' #N2 ' =' #SUM
...
```

107 PROPERTY

```
PROPERTY property-name
  OF [INTERFACE] interface-name
  IS operand
END-PROPERTY
```

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

Related Statements: CREATE OBJECT | DEFINE CLASS | INTERFACE | METHOD | SEND METHOD

Belongs to Function Group: ***Component Based Programming***

Function

The PROPERTY statement assigns an object data variable operand as the implementation to a property, outside an interface definition.

It is used if the interface definition in question is included from a copycode and is to be implemented in a class-specific way.

It may only be used within the DEFINE CLASS statement and after the interface definitions.

The interface and property names specified must be defined in the INTERFACE clause of the DEFINE CLASS statement.

Syntax Description

Syntax Element	Description
property-name	**Property Name:** This is the name assigned to the property.
OF *interface-name*	**Interface Name:** This is the name assigned to the interface.
IS *operand*	**IS Clause:** The *operand* in the IS clause assigns an object data variable as the place to store the property value.
END-PROPERTY	**End of PROPERTY Statement:** The Natural reserved word END-PROPERTY must be used to end the PROPERTY statement.

Example

The **example** contained in the documentation of the `METHOD` statement shows how the same interface is implemented differently in two classes, and how the `PROPERTY` statement and the `METHOD` statement are used to achieve this.

XIII

108 READ

```
{ READ   } [{ ALL        }] [MULTI-FETCH-clause] [RECORDS] [IN] [FILE] view-name
{ BROWSE } [{ (operand1) }]
  [PASSWORD=operand2]
  [CIPHER=operand3]
  [WITH REPOSITION]
  [sequence/range-specification]
  [STARTING WITH ISN=operand4]
  [[IN] SHARED HOLD [MODE=option]]
  [SKIP [RECORDS] IN HOLD]
  [WHERE logical-condition]
   statement ...
END-READ        (structured mode only)
LOOP            (reporting mode only)
```

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

Related Statements: ACCEPT/REJECT | AT BREAK | AT START OF DATA | AT END OF DATA | BACKOUT TRANSACTION | BEFORE BREAK PROCESSING | GET TRANSACTION DATA | DELETE | END TRANSACTION | FIND | HISTOGRAM | GET | GET SAME | LIMIT | PASSW | PERFORM BREAK PROCESSING | READLOB | RETRY | STORE | UPDATE | UPDATELOB

Belongs to Function Group: ***Database Access and Update***

Function

The READ statement is used to read records from a database. The records can be retrieved in physical sequence, in Adabas ISN sequence, or in the value sequence of a descriptor (key) field. The READ statement causes a processing loop to be initiated.

See also the following sections in the *Programming Guide*:

- *READ Statement*
- *Loop Processing*
- *Referencing of Database Fields Using (r) Notation*

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S						N	P	I		B *							yes	no
operand2	C	S				A													yes	no
operand3	C	S						N											yes	no
operand4	C	S						N	P	I		B *							yes	no

* Format B of `operand1` and `operand4` may be used with a length of less than or equal to 4.

Syntax Element Description:

Syntax Element	Description

operand1	**Number of Records to be Read:**
	
	The number of records to be read may be limited by specifying *operand1* (enclosed in parentheses, immediately after the keyword `READ`) - either as a numeric constant (in the range from `0` to `4294967295`) or as the name of a numeric variable.
	
	Example:
	
	`READ (5) IN EMPLOYEES ...`
	
	`MOVE 10 TO CNT(N2)`
	`READ (CNT) EMPLOYEES  ...`
	
	For this statement, the specified limit has priority over a limit set with a `LIMIT` statement.
	
	If a smaller limit is set with the profile or session parameter `LT`, the `LT` limit applies.
	
	Note:
	
	1. If you wish to read a 4-digit number of records, specify it with a leading zero: (`0`*nnnn*); because Natural interprets every 4-digit number enclosed in parentheses as a line-number reference to a statement.
	2. *operand1* is evaluated when the `READ` loop is entered. If the value of *operand1* is modified within the `READ` loop, this does not affect the number of records read.

Syntax Element	Description
ALL	**ALL Option:** To emphasize that *all* records are to be read, you can optionally specify the keyword ALL. The ALL option is used by default if *operand1* and ALL are omitted.
MULTI-FETCH-clause	**MULTI-FETCH Clause:** See ***MULTI-FETCH Clause*** below.
view-name	**View Name:** As *view-name*, you specify the name of a view, which must have been defined either within a DEFINE DATA statement or outside the program in a global or local data area. In reporting mode, *view-name* is the name of a DDM if no DEFINE DATA LOCAL statement is used.
PASSWORD=*operand2* CIPHER=*operand3*	**PASSWORD and CIPHER Clauses:** These clauses are applicable only to Adabas databases. They cannot be used with Entire System Server. The PASSWORD clause is used to provide a password when retrieving data from a file which is password-protected. The CIPHER clause is used to provide a cipher key when retrieving data from a file which is enciphered. See the statements FIND and PASSW for further information.
WITH REPOSITION	**WITH REPOSITION Option:** This option is used to make the READ statement sensitive for repositioning events. See ***WITH REPOSITION Option***.
sequence/range-specification	**Sequence/Range Specification:** This option specifies the sequence and/or the range of retrieval. See ***Sequence/Range Specification***.
STARTING WITH ISN=*operand4*	**STARTING WITH ISN Clause:** This clause applies only to Adabas databases. **Access to Adabas** This clause can be used in conjunction with a READ statement in physical or in logical (ascending/descending) sequence. The value supplied (*operand4*, range from 0 to 4294967295) represents an Adabas ISN (internal sequence number) and is used to specify a definite record where to start the READ loop.

Syntax Element	Description
	■ **Logical Sequence** Even if documented with an equal character (=), the `READ` statement does not return only those records with exactly the start value in the corresponding descriptor field, but starts a logical browse in ascending or descending order, beginning with the start value supplied. If some records have the same contents in the descriptor field, they will be returned in an ISN-sorted sequence. The `STARTING WITH ISN` clause is some kind of a "second level" selection criterion that applies only if the start value matches the descriptor value for the first record. All records with a descriptor value that is the same as the start value and an ISN that is "less equal"("greater equal" for a descending `READ`) than the start ISN are ignored by Adabas. The first record returned in the `READ` loop is either ■ the first record with descriptor = start value and an ISN "greater" ("less" for a descending `READ`) than the start ISN, ■ or if such a record does not exist, the first record with a descriptor "greater" ("less" for a descending `READ`) than the start value. ■ **Physical Sequence** The records are returned in the order in which they are physically stored. If a `STARTING WITH ISN` clause is specified, Adabas ignores all records until the record with the ISN that is the same as the start ISN is reached. The first record returned is the next record following the ISN=start ISN record. **Examples** This clause may be used for repositioning within a `READ` loop whose processing has been interrupted, to easily determine the next record with which processing is to continue. This is particularly useful if the next record cannot be identified uniquely by any of its descriptor values. It can also be useful in a distributed client/server application where the reading of the records is performed by a server program while further processing of the records is performed by a client program, and the records are not processed all in one go, but in batches. For an example, see the program `REASISND` below.
`[[IN] SHARED HOLD [MODE=`*`option`*`]]`	**SHARED HOLD Clause:** **Note:** This clause can be used only for access to Adabas. This clause can be used to place records being read in a "shared hold" state. A record can be put in shared hold by many users at the same time. As long as a record is in a shared hold state, it is protected from being updated, because it cannot be set into an exclusive hold by

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td></td><td>
parallel users. This ensures data consistency for the record data, as no one can update the record while it is being processed.
Especially if the same record is fetched with multiple statements to read different MU/PE occurrences (<code>GET SAME</code> statement) or to browse over a LOB field in a piecemeal technique (<code>READLOB</code> statement), the shared hold state can guarantee data stability over this transaction without blocking the record for other users.
Although such a hold state is an efficient way to protect read sequences, it is a basic and important matter when to release the record again from this “soft lock”. Since this question depends on individual application aspects, different options can be selected with the <code>MODE</code> subclause.
<table>
<tr><th>MODE Option</th><th>Hold Period</th><th>Explanation</th></tr>
<tr><td><code>C</code></td><td>Only at the moment of reading the record.</td><td>Ensures only that the record version being read has been committed by the last user who updated the record. This option does not really set a lock in hold state, but checks only that the record is not in exclusive hold by another user at time of read.</td></tr>
<tr><td><code>Q</code></td><td>Until the next record in a sequence is read.</td><td>Releases the record from shared hold when
the next record is read in the loop sequence or
the loop is terminated or
an <code>END TRANSACTION</code> or <code>BACKOUT TRANSACTION</code> is executed.</td></tr>
<tr><td><code>S</code></td><td>Until the logical transaction is terminated.</td><td>Releases the record from shared hold when a logical transaction is terminated with an <code>END TRANSACTION</code> or</td></tr>
</table>
</td></tr>
</table>

Syntax Element	Description
	BACKOUT TRANSACTION statement. MODE=Q and MODE=S ensure that the record being read cannot be updated concurrently by other users until it has been released from hold again. If the MODE subclause is not specified, MODE=C is the default. See also ***Example 8 - SHARED HOLD Clause*** below.
SKIP RECORDS IN HOLD	**SKIP RECORDS Clause:** **Note:** This clause can be used only for access to Adabas. Whenever a record is going to be read with hold, a Natural error NAT3145 (Adabas response code 145) might happen if the record is in hold by another user at this time. This occurs if a shared hold is requested and the record is in exclusive hold or if an exclusive hold is requested and the record is in either exclusive or shared hold. Although error NAT3145 is surely the right reaction to assure a "clean data processing", sometimes it might be useful if a record in hold could be skipped. If it is alright that such a record will not be processed and the loop processing should continue, the SKIP RECORDS clause should be used. If the SKIP RECORDS clause is applied, Natural first tries to read the record with hold. If the record is already in hold and the Natural error NAT3145 would occur, - no error processing is initiated; - the record (currently in hold by another user) is instantly re-fetched without hold, but not processed in terms of the program logic; - the record which comes next after the skipped record is read with hold and the processing continues. See also ***Example 9 - SKIP RECORDS Clause***.
WHERE *logical-condition*	**WHERE Clause:** The WHERE clause may be used to specify an additional selection criterion (*logical-condition*) which is evaluated *after* a value has been read and *before* any processing is performed on the value (including the AT BREAK evaluation). The syntax for a *logical-condition* is described in the section *Logical Condition Criteria* in the *Programming Guide*.

Syntax Element	Description
	If a LIMIT statement or a processing limit is specified in a READ statement containing a WHERE clause, records which are rejected as a result of the WHERE clause are not counted against the limit.
END-READ	**End of READ Statement:**
LOOP	In structured mode, the Natural reserved keyword END-READ must be used to end the READ statement. In reporting mode, the Natural statement LOOP is used to end the READ statement.

MULTI-FETCH Clause

Note: This clause can only be used for Adabas databases.

```
            ┌ ON                        ┐
MULTI-FETCH ┤ OFF                       ├
            └ [OF] multi-fetch-factor   ┘
```

Note: [MULTI-FETCH OF *multi-fetch-factor*] is supported for database types ADA/ADA2. The default processing mode is applied; see profile parameter MFSET. The MULTI-FETCH clause is ignored in case Adabas LA or large objects fields are used or a view size greater than 64KB is defined.

For more information, see the section *MULTI-FETCH Clause* (Adabas) in the *Programming Guide*.

WITH REPOSITION Option

Note: This option can only be applied if the underlying database is Adabas.

With a WITH REPOSITION option, you can make a READ statement sensitive for repositioning events. This allows you to reposition to another start value within an active READ loop. Processing of the READ statement then continues with the new start value.

A repositioning event is triggered by one of two ways when you use a READ statement with the WITH REPOSITION option:

1. When an ESCAPE TOP REPOSITION statement is executed. At execution of an ESCAPE TOP REPOSITION statement, Natural makes an instant branch to the loop begin and performs a restart; that is, the database repositions to a new record in the file according to the current content of the search value variable. At the same time, the loop-counter *COUNTER is reset to zero.
2. When a READ loop tries to fetch the next record from the database and the value of the system variable *COUNTER is 0.

Note: If `*COUNTER` is set to `0` within the active `READ` loop, processing of the current record is continued; no instant branch to the loop begin is performed. You cannot trigger a reposition event in this fashion on Natural for Windows or Natural for Linux and Cloud. This functionality has only been retained for compatibility with earlier versions of Natural for Mainframes. Therefore, it is not recommended that you use this process.

Functional Considerations

- If the `READ` statement has a loop-limit (e.g. `READ (10) EMPLOYEES WITH REPOSITION ..`) and a restart event was triggered, the loop gets another 10 new records, no matter how many records where already processed until the repositioning takes place.
- If an `ESCAPE TOP REPOSITION` statement is executed, but the innermost loop is not capable of repositioning (since the `WITH REPOSITION` keyword is not set in the `READ` statement or the posted loop statement is anything else but a `READ`), a corresponding runtime error is issued.
- Since the `ESCAPE TOP` statement does not allow a reference, you can only initiate a reposition event if the innermost processing loop is a `READ ..WITH REPOSITION` statement.
- A reposition event does not trigger the execution of the `AT START OF DATA` section, nor does it trigger the re-evaluation of the loop-limit operand (if it is a variable).
- If the search value was not altered, the loop repositions to the same record like at initial loop start.

Sequence/Range Specification

Three syntax options are available to specify the sequence and/or the range of retrieval.

Syntax Option 1: `READ PHYSICAL`

[IN] [PHYSICAL]	[ASCENDING DESCENDING VARIABLE *operand5* DYNAMIC *operand5*]	[SEQUENCE]

Syntax Option 2: `READ BY ISN`

[IN] [ISN]	[ASCENDING DESCENDING VARIABLE *operand5*]	[SEQUENCE]

```
{ BY   }  ISN  [ [ =                  ]  operand6 ] [ [ THRU      ]  operand7 ]
{ WITH }       [ [ EQ                 ]           ] [ [ ENDING AT ]           ]
               [ [ EQUAL TO           ]           ]
               [ [ [STARTING] FROM    ]           ]
```

Syntax Option 3: `READ BY DESCRIPTOR`

```
[IN] [LOGICAL]  [ ASCENDING          ]  [SEQUENCE]
                [ DESCENDING         ]
                [ VARIABLE operand5  ]
                [ DYNAMIC operand5   ]

{ BY   }  descriptor  { [ [ =                ]  operand6 ] [ [ THRU      ]  operand7 ] }
{ WITH }              { [ [ EQ               ]           ] [ [ ENDING AT ]           ] }
                      { [ [ EQUAL TO         ]           ]                             }
                      { [ [ [STARTING] FROM  ]           ]                             }
                      {                                                                }
                      {   [ =                ]  operand6 TO operand7                   }
                      {   [ EQ               ]                                         }
                      {   [ EQUAL TO         ]                                         }
                      {   [ [STARTING] FROM  ]                                         }
                      {                                                                }
                      {   { <                }                                         }
                      {   { LT               }                                         }
                      {   { LESS THAN        }                                         }
                      {   { >                }                                         }
                      {   { GT               }                                         }
                      {   { GREATER THAN     }  operand6                               }
                      {   { <=               }                                         }
                      {   { LE               }                                         }
                      {   { LESS EQUAL       }                                         }
                      {   { >=               }                                         }
                      {   { GE               }                                         }
                      {   { GREATER EQUAL    }                                         }
```

Notes:

1. The syntax options [2] and [3] are not available with Entire System Server.
2. If the comparators of Diagram 3 are used, the options `ENDING AT`, `THRU` and `TO` may not be used. These comparators are also valid for the `HISTOGRAM` statement.

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand5		S				A													yes	no
operand6	C	S				A		N	P	I	F	B*	D	T	L				yes	no
operand7	C	S				A		N	P	I	F	B*	D	T	L				yes	no

* Format B of `operand6` and `operand7` may be used only with a length of less than or equal to 4.

Syntax Element Description:

Syntax Element	Description
`READ IN PHYSICAL SEQUENCE`	**Read in Physical Sequence:** This option is used to read records in the order in which they are physically stored in a database. `PHYSICAL` is the default sequence.
`READ BY ISN`	**Read by ISN:** This option is used to read records in the order of Adabas ISNs (internal sequence numbers). Instead of using the keyword `BY`, you may specify the keyword `WITH`, which has the same effect. For `READ BY ISN`, *operand6* and *operand7* must be provided as a numeric constant or user-defined variable in the range from `0` to `4294967295`. `READ BY ISN` can only be used for Adabas databases. **Note:** For XML databases: `READ BY ISN` is used to read XML objects according to the order of Tamino object IDs.
`READ IN LOGICAL SEQUENCE`	**Read in Logical Sequence:** This option is used to read records in the order of the values of a descriptor (key) field. If you specify a descriptor, the records will be read in the value sequence of the descriptor. A descriptor, subdescriptor, superdescriptor or hyperdescriptor may be used for sequence control. A phonetic descriptor, a descriptor within a periodic group, or a superdescriptor which contains a periodic-group field cannot be used. If you do not specify a descriptor, the default descriptor as specified in the DDM (field `Default Sequence`) will be used. If the descriptor used for sequence control is defined with null-value suppression (Adabas only), any record which contains a null value for the descriptor will not be read. If the descriptor is a multiple-value field (Adabas only), the same record will be read multiple times depending on the number of values present.

Syntax Element	Description
	Note: READ IN LOGICAL SEQUENCE is also discussed in the *Programming Guide*; see *Statements for Database Access*, *READ Statement*.

ASCENDING | DESCENDING | VARIABLE | DYNAMIC SEQUENCE	**Ascending/Descending Order:**
	
	This clause only applies to Adabas, XML databases and SQL databases. In a READ PHYSICAL statement, it can only be applied to Db2 databases. In a READ BY ISN statement, it can only be applied to Adabas for Linux and Windows 7.2 (or higher) or Adabas for Mainframe 8.6 (or higher).
	
	With this clause, you can determine whether the records are to be read in ascending or descending sequence.
	
	■ The default sequence is ascending (which may, but need not, be explicitly specified by using the keyword ASCENDING).
	■ If the records are to be read in descending sequence, you specify the keyword DESCENDING.
	■ If, instead of determining it in advance, you want to have the option of determining at runtime whether the records are to be read in ascending or descending sequence, you either specify the keyword VARIABLE or DYNAMIC, followed by a variable (*operand5*). *operand5* has to be of format/length A1 and can contain the value "A" (for "ascending") or "D" (for "descending").
	  ■ If the keyword VARIABLE is used, the reading direction (value of *operand5*) is evaluated at start of the READ processing loop and remains the same until the loop is terminated, regardless if the *operand5* field is altered in the READ loop or not.
	  ■ If the keyword DYNAMIC is used, the reading direction (value of *operand5*) is evaluated before every record fetch in the READ processing loop and may be changed from record to record. This allows to change the scroll sequence from ascending to descending (and vice versa) at any place in the READ loop. This option is not allowed in a READ BY ISN statement.
	
	Note: For XML databases: DYNAMIC SEQUENCE is not available.

STARTING FROM ... ENDING AT/TO	**STARTING FROM/ENDING AT Clauses:** The STARTING FROM and ENDING AT clauses are used to limit reading to a set of records based on a user-specified range of values. The STARTING FROM clause (= or EQ or EQUAL TO or [STARTING] FROM) determines the starting value for the read operation. If a starting value is specified, reading will begin with the value specified. If the starting value does not exist in the file, the next higher (or lower for a DESCENDING read) value will be used. If no higher (or lower for DESCENDING) value exists, the loop will not be entered. In order to limit the records to an end value, you may specify an ENDING AT clause with the terms THRU, ENDING AT, or TO that imply an inclusive range. Whenever the read descriptor field exceeds the end value specified, an automatic loop termination is performed. Although the basic functionality of the TO, THRU and ENDING AT keywords looks quite similar, internally they differ in how they work.

Syntax Element	Description
THRU/ENDING AT	**THRU/ENDING AT Option:** If THRU or ENDING AT is used, only the start value is supplied to the database, but the end value check is performed by the Natural runtime system, after the record is returned by the database. If the read direction is ASCENDING, you have to supply the lower value as the start value and the higher value as the end value, since the start value represents the value (and record) returned first in the READ loop. However, if you invoke a backwards read (DESCENDING), the higher value has to appear in the start value and the lower value in the end value. Internally, to determine the end of the range to be read, Natural reads one record beyond the end value. If you have left the READ loop because the end value has been reached, be aware that this last record is in fact not the last record within the demanded range, but the first record beyond that range (except if the file does not contain a further record after the last result record). The THRU and ENDING AT clauses can be used for all databases which support the READ or HISTOGRAM statements.
TO	**TO Option:** If the keyword TO is used, both the start value and the end value are sent to the database, and Natural does not perform checks for value ranges. If the end value is exceeded, the database reacts the same as when "end-of-file" is reached, and the database loop is exited. Since the complete range checking is done by the database, the lower value (of the range) is always supplied in the start value and the higher value filled into the end value, regardless if you are browsing in ASCENDING or DESCENDING order.

Notes on Functional Differences between THRU/ENDING AT and TO

The following list describes the functional differences between the usage of the THRU/ENDING AT and TO options.

THRU/ENDING AT	TO
When the READ loop terminates because the end value has been reached, the view contains the first record "out-of-range".	When the READ loop terminates because the end value has been reached, the view contains the last record of the specified range.
If a end value variable is modified during the READ loop, the new value will be used for end value check on next record being read.	The end value variable will only be evaluated at READ loop start. All further modifications during the READ loop have no effect.
An incorrect range (for example, READ .. = 'B' THRU 'A') does not cause a database error, but just returns no record.	An incorrect range results in a database error (for example, Adabas RC=61), because a value range must not be supplied in descending order.
If a READ .. DESCENDING is used with the start and end values, the start value is used to position in the file, whereas the end value is used by Natural to check for	Since both values are passed to the database, they have to appear in ascending order. In other words, the start value is lower than (or equal to) the end

THRU/ENDING AT	TO
"end-of-range". Therefore the start value is higher than (or equal to) the end value.	value, no matter if you are reading in ascending or descending order.
In order to check for range overflow, the descriptor value has to appear in the underlying database view; that is, it must be returned in the record buffer.	The descriptor is not required in the record fields returned.
The end value check for an Adabas multi-value field (MU-field) or a sub-/super-/hyper-descriptor is not possible and leads to syntax error NAT0160 at program compilation.	You may specify an end value for MU-fields and sub-/super-/hyper-descriptors.
Can be used for all databases.	Can be used for all databases.

Note: The result of `READ`/`HISTOGRAM` `THRU`/`ENDING AT` might differ from the result of `READ`/`HISTOGRAM` `TO` if Natural and the accessed database reside on different platforms with different collating sequences.

System Variables Available with READ

The Natural system variables `*ISN` and `*COUNTER` are available with the `READ` statement.

The format/length of these system variables is P10. This format/length cannot be changed.

The usage of the system variables is illustrated below.

System Variable	Explanation
`*ISN`	The system variable `*ISN` contains the Adabas ISN of the record currently being processed.
`*COUNTER`	The system variable `*COUNTER` contains the number of times the processing loop has been entered.

Examples

- Example 1 - READ Statement
- Example 2 - READ WITH REPOSITION
- Example 3 - Combining READ and FIND Statements
- Example 4 - DESCENDING Option
- Example 5 - VARIABLE Option
- Example 6 - DYNAMIC Option
- Example 7 - STARTING WITH ISN Clause
- Example 8 - SHARED HOLD Clause
- Example 9 - SKIP RECORDS Clause

- Example 10 - READ DESCENDING BY ISN

Example 1 - READ Statement

```
** Example 'REAEX1S': READ (structured mode)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
1 VEHIC-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 3
*
WRITE 'READ IN PHYSICAL SEQUENCE'
READ EMPLOY-VIEW IN PHYSICAL SEQUENCE
  DISPLAY NOTITLE PERSONNEL-ID NAME *ISN *COUNTER
END-READ
*
WRITE / 'READ IN ISN SEQUENCE'
READ EMPLOY-VIEW BY ISN STARTING FROM 1 ENDING AT 3
  DISPLAY         PERSONNEL-ID NAME *ISN *COUNTER
END-READ
*
WRITE / 'READ IN NAME SEQUENCE'
READ EMPLOY-VIEW BY NAME
  DISPLAY         PERSONNEL-ID NAME *ISN *COUNTER
END-READ
*
WRITE / 'READ IN NAME SEQUENCE STARTING FROM ''M'''
READ EMPLOY-VIEW BY NAME STARTING FROM 'M'
  DISPLAY         PERSONNEL-ID NAME *ISN *COUNTER
END-READ
*
END
```

Output of Program REAEX1S:

```
PERSONNEL         NAME             ISN         CNT
   ID
--------- -------------------- ----------- -----------

READ IN PHYSICAL SEQUENCE
50005800  ADAM                           1           1
50005600  MORENO                         2           2
50005500  BLOND                          3           3
```

```
READ IN ISN SEQUENCE
50005800  ADAM                          1           1
50005600  MORENO                        2           2
50005500  BLOND                         3           3

READ IN NAME SEQUENCE
60008339  ABELLAN                     478           1
30000231  ACHIESON                    878           2
50005800  ADAM                          1           3

READ IN NAME SEQUENCE STARTING FROM 'M'
30008125  MACDONALD                   923           1
20028700  MACKARNESS                  765           2
40000045  MADSEN                      508           3
```

Equivalent reporting-mode example: **REAEX1R**.

Example 2 - READ WITH REPOSITION

```
DEFINE DATA LOCAL
1 MYVIEW VIEW OF ...
  2 NAME
1 #STARTVAL (A20) INIT <'A'>
1 #ATTR     (C)
END-DEFINE
...
SET KEY PF3
...
READ MYVIEW WITH REPOSITION BY NAME = #STARTVAL
INPUT (IP=OFF AD=O) 'NAME:' NAME /
    'Enter new start value for repositioning:' #STARTVAL (AD=MT CV=#ATTR) /
    'Press PF3 to stop'
  IF *PF-KEY = 'PF3'
    THEN STOP
  END-IF
  IF #ATTR MODIFIED
    THEN ESCAPE TOP REPOSITION
  END-IF
END-READ
...
```

```
DEFINE DATA LOCAL
1 MYVIEW VIEW OF ...
  2 NAME
1 #STARTVAL (A20) INIT <'A'>
1 #ATTR     (C)
END-DEFINE
...
SET KEY PF3
...
READ MYVIEW WITH REPOSITION BY NAME = #STARTVAL
```

```
  INPUT (IP=OFF AD=O) 'NAME:' NAME /
    'Enter new start value for repositioning:' #STARTVAL (AD=MT CV=#ATTR) /
    'Press PF3 to stop'
  IF *PF-KEY = 'PF3'
    THEN STOP
  END-IF
  IF #ATTR MODIFIED
    THEN RESET *COUNTER
  END-IF
END-READ
...
```

Example 3 - Combining READ and FIND Statements

The following program reads records from the EMPLOYEES file in logical sequential order based on the values of the descriptor NAME. A FIND statement is then issued to the VEHICLES file using the personnel number from the EMPLOYEES file as search criterion. The resulting report shows the name (read from the EMPLOYEES file) of each person read and the model of automobile (read from the VEHICLES file) owned by this person. Multiple lines with the same name are produced if the person owns more than one automobile.

```
** Example 'REAEX2': READ and FIND combination
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 CITY
1 VEH-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 10
*
RD. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  SUSPEND IDENTICAL SUPPRESS
  FD. FIND VEH-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      ENTER
    END-NOREC
    DISPLAY NOTITLE (ES=OFF IS=ON ZP=ON AL=15)
            PERSONNEL-ID (RD.)
            FIRST-NAME (RD.)
            MAKE (FD.) (IS=OFF)

  END-FIND
END-READ
END
```

Output of Program REAEX2:

```
PERSONNEL      FIRST-NAME         MAKE
      ID
-------------- -------------- --------------

20007500        VIRGINIA        CHRYSLER
20008400        MARSHA          CHRYSLER
                                CHRYSLER
20021100        ROBERT          GENERAL MOTORS
20000800        LILLY           FORD
                                MG
20001100        EDWARD          GENERAL MOTORS
20002000        MARTHA          GENERAL MOTORS
20003400        LAUREL          GENERAL MOTORS
30034045        KEVIN           DATSUN
30034233        GREGORY         FORD
11400319        MANFRED
```

Example 4 - DESCENDING Option

```
** Example 'READSCND': READ (with DESCENDING SEQUENCE)
************************************************************************
DEFINE DATA LOCAL
1 EMPL VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BIRTH
END-DEFINE
*
READ (10) EMPL IN DESCENDING SEQUENCE BY NAME FROM 'ZZZ'
  DISPLAY *ISN NAME FIRST-NAME BIRTH (EM=YYYY-MM-DD)
END-READ
END
```

Example 5 - VARIABLE Option

```
** Example 'REAVSEQ':  READ  (with VARIABLE SEQUENCE)
************************************************************************
DEFINE DATA LOCAL
1 EMPL VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BIRTH
*
1 #DIR        (A1)
1 #STARTVALUE (A20)
END-DEFINE
*
SET KEY PF7 PF8
```

```
*
INPUT 'Select READ direction'
   // 'Press' 08T 'PF7' (I)                  21T 'to read backward'
    /        08T 'PF8' (I) 'or' 'ENTER' (I) 21T 'to read forward'
*
IF *PF-KEY = 'PF7'
  MOVE 'D'   TO #DIR
  MOVE 'ZZZ' TO #STARTVALUE
ELSE
  MOVE 'A' TO #DIR
  MOVE 'A' TO #STARTVALUE
END-IF
*
READ (10) EMPL IN VARIABLE #DIR SEQUENCE
               BY NAME FROM #STARTVALUE
  DISPLAY *ISN NAME FIRST-NAME BIRTH (EM=YYYY-MM-DD)
END-READ
END
```

Example 6 - DYNAMIC Option

```
DEFINE DATA LOCAL
1 #DIRECTION (A1) INIT <'A'>   /* 'A' = ASCENDING
1 #EMPVIEW VIEW OF EMPLOYEES
2 NAME
...
END-DEFINE
...
READ #EMPVIEW IN DYNAMIC #DIRECTION SEQUENCE BY NAME = 'SMITH'
   INPUT (AD=O) NAME
       / 'Press PF7 to scroll in DESCENDING sequence'
       / 'Press PF8 to scroll in ASCENDING  sequence'
   ..
   IF *PF-KEY = 'PF7' THEN MOVE 'D' TO #DIRECTION END-IF
   IF *PF-KEY = 'PF8' THEN MOVE 'A' TO #DIRECTION END-IF
END-READ
...
```

Example 7 - STARTING WITH ISN Clause

```
** Example 'REASISND': READ  (with STARTING WITH ISN)
************************************************************************
DEFINE DATA LOCAL
1 EMPL VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BIRTH
*
1 #DIR      (A1)
1 #STARTVAL (A20)
```

```
1 #STARTISN (N8)
END-DEFINE
*
SET KEY PF3 PF7 PF8
*
MOVE 'ADKINSON' TO #STARTVAL
*
READ (9) EMPL BY NAME = #STARTVAL
  WRITE *ISN NAME FIRST-NAME BIRTH (EM=YYYY-MM-DD) *COUNTER
  IF *COUNTER = 5 THEN
    MOVE NAME TO #STARTVAL
    MOVE *ISN TO #STARTISN
  END-IF
END-READ
*
#DIR := 'A'
*
REPEAT
 READ EMPL IN VARIABLE #DIR  BY NAME = #STARTVAL
            STARTING WITH ISN = #STARTISN
    MOVE NAME TO #STARTVAL
    MOVE *ISN TO #STARTISN
    INPUT NO ERASE  (IP=OFF AD=O)
         15/01 *ISN  NAME  FIRST-NAME  BIRTH (EM=YYYY-MM-DD)
           //  'Direction:' #DIR
           //  'Press PF3 to stop'
           /   '      PF7 to go step back'
           /   '      PF8 to go step forward'
           /   '      ENTER to continue in that direction'
    /*
    IF *PF-KEY = 'PF7' AND #DIR = 'A'
      MOVE 'D' TO #DIR
      ESCAPE BOTTOM
    END-IF
    IF *PF-KEY = 'PF8' AND #DIR = 'D'
      MOVE 'A' TO #DIR
      ESCAPE BOTTOM
    END-IF
    IF *PF-KEY = 'PF3'
      STOP
    END-IF
  END-READ
  /*
  IF *COUNTER(0290) = 0
    STOP
  END-IF
END-REPEAT
END
```

Example 8 - SHARED HOLD Clause

```
READ EMPL-VIEW WITH NAME = ...
     IN SHARED HOLD MODE=Q  /* Record in shared hold until next record is read.
   ...
   GET EMPL-VIEW  *ISN      /* The record remains unchanged!
   ...
END-READ
```

Example 9 - SKIP RECORDS Clause

```
READ EMPL-VIEW WITH NAME = ...  /* Records found are put in hold while reading.
     SKIP RECORDS IN HOLD       /* Records already held by other users are skipped
  ...                           /* to prevent error NAT3145.
  UPDATE
  END TRANSACTION
END-READ
```

Example 10 - READ DESCENDING BY ISN

```
** Example 'READVISN': READ with DESCENDING/VARIABLE BY ISN
**
** Note: The ASCENDING and DESCENDING order for READ BY ISN can only be
**       applied to ADABAS 7.2 (or higher) on Linux and Windows.
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
*
1 #DIR-ASCENDING  (A1) CONST<'A'>
1 #DIR-DESCENDING (A1) CONST<'D'>
*
END-DEFINE
*
WRITE 'READ with ASCENDING BY ISN = 500'
READ (5) EMPLOY-VIEW ASCENDING BY ISN = 500
  DISPLAY NOTITLE *ISN NAME FIRST-NAME
END-READ
*
WRITE / 'READ with DESCENDING BY ISN = 500'
READ (5) EMPLOY-VIEW DESCENDING BY ISN = 500
  DISPLAY         *ISN NAME FIRST-NAME
END-READ
*
WRITE / 'READ with VARIABLE ascending BY ISN = 500'
READ (5) EMPLOY-VIEW VARIABLE #DIR-ASCENDING BY ISN = 500
  DISPLAY         *ISN NAME FIRST-NAME
END-READ
*
```

```
WRITE / 'READ with VARIABLE descending BY ISN = 500'
READ (5) EMPLOY-VIEW VARIABLE #DIR-DESCENDING BY ISN = 500
  DISPLAY         *ISN NAME FIRST-NAME
END-READ
END                                                                            *          ↩
```

Output of Program READVISN:

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

READ with ASCENDING BY ISN = 500
        500 JENSEN               HANS
        501 JENSEN               CLAUS
        502 SOERENSEN            KARL
        503 ERIKSSEN             JONAS
        504 ANDERSEN             ANITA

READ with DESCENDING BY ISN = 500
        500 JENSEN               HANS
        499 MARTINEZ             MANUEL
        498 OSEA                 ROBERTO
        497 FERNANDEZ            CARMEN
        496 DE LA IGLESIA        JORGE

READ with VARIABLE ascending BY ISN = 500
        500 JENSEN               HANS
        501 JENSEN               CLAUS
        502 SOERENSEN            KARL
        503 ERIKSSEN             JONAS
        504 ANDERSEN             ANITA

READ with VARIABLE descending BY ISN = 500
        500 JENSEN               HANS
        499 MARTINEZ             MANUEL
        498 OSEA                 ROBERTO
        497 FERNANDEZ            CARMEN
        496 DE LA IGLESIA        JORGE
```

109 READ RESULT SET (SQL)

Common Set Syntax:

READ [(*limit*)] RESULT SET *result-set* INTO	{	VIEW *view-name* *parameter* [, *parameter*]...	} FROM *ddm-name*
[GIVING [:] *sql-code*]			
END-RESULT	(*structured mode only*)		
LOOP	(*reporting mode only*)		

Extended Set Syntax:

READ [(*limit*)] RESULT SET *result-set* INTO	{	VIEW *view-name* *parameter* [, *parameter*]...	} FROM *ddm-name*
[WITH INSENSITIVE SCROLL [:] *scroll-hv*]			
[GIVING [:] *sql-code*]			
END-RESULT	(*structured mode only*)		
LOOP	(*reporting mode only*)		

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

Belongs to Function Group: ***Database Access and Update***

Function

The SQL statement READ RESULT SET can only be used in conjunction with a CALLDBPROC statement. It is used to read a result set which was created by a stored procedure that was invoked by a previous CALLDBPROC statement.

Syntax Description

Syntax Element	Description
limit	**Limit Option:** You can limit the number of rows to be read. You can specify the limit either as a numeric constant (0 - 4294967295) or as a variable of format N, P or I.
result-set	**Result Set:** As *result-set* you specify a result-set locator variable filled by a preceding CALLDBPROC statement. *result-set* has to be a variable of format/length I4.

Syntax Element	Description
	Note: If a syncpoint operation takes place between the `CALLDBPROC` statement and the `READ RESULT SET` statement, the result sets can no longer be accessed by the `READ RESULT SET` statement.
`INTO`	**INTO Clause:** The `INTO` clause is used to specify the target fields in the program which are to be filled with the result set. The `INTO` clause can specify either single parameters or one or more views as defined in the `DEFINE DATA` statement.
`VIEW` *`view-name`*	**VIEW Clause:** *`view-name`* specifies a view whose fields receive the columns of the result set created by the stored procedure invoked via the `CALLDBPROC` statement. The number of columns of the result set must correspond to the number of fields defined in the view (not counting group fields, redefining fields and indicator fields).
`parameter`	**Parameter:** Each *`parameter`* specifies a field which receives a column of the result set created by the stored procedure invoked via the `CALLDBPROC` statement.
`FROM` *`ddm-name`*	**DDM Name:** As *`ddm-name`* you specify the name of the data definition module (DDM) which is used to "address" the database executing the stored procedure. For further information, see *`ddm-name`*.
`WITH INSENSITIVE SCROLL [:]` *`scroll_hv`*	**WITH INSENSITIVE SCROLL Clause:** This clause belongs to the **SQL Extended Set**. This clause is not currently supported. When used, it will cause a compiler error.
`GIVING` *`sqlcode`*	**GIVING *`sqlcode`* Clause:** This clause may be used to obtain the SQLCODE of the SQL "fetch" operation used to process the result set. If this clause is specified and the SQLCODE of the SQL operation is not `0`, no Natural error message will be issued. In this case, the action to be taken in reaction to the SQLCODE value has to be coded in the invoking Natural object. The *`sqlcode`* field has to be a variable of format/length I4. If the `GIVING` *`sqlcode`* clause is omitted, a Natural error message will be issued if the SQLCODE is not `0`.
`END-RESULT`	**End of READ RESULT SET Statement:**

Syntax Element	Description
`LOOP`	In structured mode, the Natural reserved keyword `END-RESULT` must be used to end the `READ RESULT SET` statement. In reporting mode, the Natural statement `LOOP` must be used to end the `READ RESULT SET` statement.

110 READ WORK FILE

Related Statements: CLOSE WORK FILE | DEFINE WORK FILE | WRITE WORK FILE

Belongs to Function Group: *Control of Work Files / PC Files*

Function

The READ WORK FILE statement is used to read data from a non-Adabas physical sequential work file. The data is read sequentially from the work file. How it is read is independent of how it was written to the work file.

READ WORK FILE initiates and executes a processing loop for reading of all records on the work file. Automatic break processing may be performed within a READ WORK FILE loop.

Notes:

1. When an end-of-file condition occurs during the execution of a READ WORK FILE statement, Natural automatically closes the work file.
2. For Entire Connection: If an Entire Connection work file is read, no I/O statement may be placed within the READ WORK FILE processing loop.
3. For Unicode and code page support, see *Work Files and Print Files on Windows and Linux Platforms* in the *Unicode and Code Page Support* documentation.

If an ASCII work file is read, it is possible that an empty record is returned as the last record after the last physical record. This is due to the fact that Natural does not read individual records, but reads larger blocks of the work file in order to optimize file-access performance.

Syntax 1 - READ WORK FILE with Processing Loop

```
READ WORK [FILE] work-file-number
{ RECORD operand1                                                                              }
{                                                                                              }
{          { { [ OFFSET n  ] ... operand2 } ...                                              } }
{ [AND]    {   [ FILLER nX ]                                                              } }
{ [SELECT] {                                                                               } }
{          { [ [ OFFSET n  ] ... operand2 ] ...  [ OFFSET  ] ...  operand4 [AND]           } }
{          {   [ FILLER nX ]                     [ n       ]      ADJUST                   } }
{          {                                     [ FILLER  ]      [OCCURRENCES]            } }
{          {                                     [ nX      ]                               } }
[GIVING LENGTH operand3]
statement ...
```

```
END-WORK                        (structured mode only)
LOOP                            (reporting mode only)
```

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

Syntax 2 - READ WORK FILE without Processing Loop

```
READ WORK [FILE] work-file-number ONCE
  RECORD  operand1
 ⎧                 ⎧ ⎧ ⎡ OFFSET ⎤                 ⎫                                               ⎫⎫
 ⎪                 ⎪ ⎨ ⎢ n      ⎥ ... operand2    ⎬ ...                                           ⎪⎪
 ⎪                 ⎪ ⎩ ⎢ FILLER ⎥                 ⎭                                               ⎪⎪
 ⎪                 ⎪   ⎣ nX     ⎦                                                                 ⎪⎪
 ⎨ [AND]           ⎨                                                                              ⎬⎬
 ⎪ [SELECT]        ⎪                                                                              ⎪⎪
 ⎪                 ⎪ ⎡ ⎡ OFFSET ⎤              ⎤       ⎡ OFFSET n  ⎤        operand4 [AND]        ⎪⎪
 ⎪                 ⎪ ⎢ ⎢ n      ⎥ ... operand2 ⎥ ...   ⎣ FILLER nX ⎦ ...    ADJUST                ⎪⎪
 ⎩                 ⎩ ⎣ ⎢ FILLER ⎥              ⎦                            [OCCURRENCES]         ⎭⎭
                       ⎣ nX     ⎦
[GIVING LENGTH operand3]
[AT [END] [OF] [FILE] statement ... END-ENDFILE]      (structured mode only)
                          ⎧ statement         ⎫
[ AT [END] [OF] [FILE]    ⎨ DO statement ...  ⎬ ]      (reporting mode only)
                          ⎩ DOEND             ⎭
```

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

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1		S	A	G		A	U	N	P	I	F	B	D	T	L	C			yes	yes
operand2		S	A	G		A	U	N	P	I	F	B	D	T	L	C			yes	yes
operand3		S								I									yes	yes
operand4			A			A	U	N	P	I	F	B	D	T	L	C			no	no

See also *Field Lengths*.

Syntax Element Description:

Syntax Element	Description
work-file-number	**Work File Number:** The work file number (as defined to Natural) to be used. The work file number is either ■ a numeric constant in the value range 1:32 or ■ a numeric variable of type (B/N/P/I) defined with a `CONST` clause which assigning a value in range (1:32). Variable is a scalar (non-array) without precision digits for type (N/P), length in between 1-4 for type (B), and no redefinition field.
`ONCE`	**ONCE Option:** `ONCE` is used to indicate that only one record is to be read. No processing loop is initiated (and therefore the loop-closing keyword `END-WORK` or `LOOP` must not be specified). If `ONCE` is specified, the `AT END OF FILE` clause should also be used. If a `READ WORK FILE` statement specified with the `ONCE` option is controlled by a user-initiated processing loop, an end-of-file condition may be detected on the work file before the loop ends. All fields read from the work file still contain the values from the last record read. The work file is then repositioned to the first record which will be read upon the next execution of `READ WORK FILE ONCE`.
`RECORD` *operand1* `FILLER` *nX*	**RECORD Option:** In reporting mode, an operand list (*operand1*) corresponding to the layout of the record must be provided. Specify `FILLER` *nX* if the total length of the operands is less that the work file record length. In structured mode, or if the record to be used is defined using a `DEFINE DATA` statement, only one field (or group) may be specified, and `FILLER` is not permitted. The field (or group) must be long enough to receive the entire work file record. No checking and no conversion is performed by Natural on the data contained in the record. It is the user's responsibility to describe the record layout correctly in order to avoid program abends caused by non-numeric data in numeric fields. Because no checking is performed by Natural, this option is the fastest way to process records from a sequential file. The record area defined by *operand1* is filled with blanks before the record is read. Thus, an end-of-file condition will return a cleared area. Short records will have blanks appended. See ***Overview of RECORD Option Usage*** below.
`SELECT`	**SELECT Option (Default):** If `SELECT` is specified, only the fields specified in the operand list (*operand2*) will be made available. The position of the field in the input record may be indicated with an `OFFSET` and/or `FILLER` specification.

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><td><code>OFFSET n</code></td><td><code>OFFSET 0</code> indicates the first byte of the record.</td></tr>
<tr><td><code>FILLER nX</code></td><td>Indicates that n bytes are to be skipped in the input record.</td></tr>
</table>
Natural will assign the selected values to the individual fields and check that numeric fields as selected from the record actually contain valid numeric data according to their definition. Because checking of selected fields is performed by Natural, this option results in more overhead for the processing of a sequential file.

If a record does not fill all fields specified in the <code>SELECT</code> option, the following applies:

■ For a field which is only partially filled, the section which has not been filled is reset to blanks or zeros.
■ Fields which are not filled at all still have the contents they had before.

If the file type CSV is read, the <code>OFFSET</code> option is ignored.
</td></tr>
<tr><td><code>operand4 AND ADJUST OCCURRENCES</code></td><td>ADJUST Clause:

Specify a one-dimensional X-array with complete range (*). The X-array is expanded or reduced to the number of occurrences needed to receive all data read. See Handling of X-Arrays below.

Note: This feature is not supported by Entire Connection.</td></tr>
<tr><td><code>GIVING LENGTH operand3</code></td><td>GIVING LENGTH Clause

This clause can be used to retrieve the actual length of the record being read. The length (number of bytes) is returned in <code>operand3</code>.

<code>operand3</code> must be defined with format/length I4.

If the work file is defined as <code>TYPE UNFORMATTED</code>, the length returned indicates the number of bytes read from the byte-stream, including bytes skipped using the <code>FILLER</code> operand.

If the <code>GIVING LENGTH</code> clause is used with work file type <code>CSV</code>, the operand specified with <code>GIVING LENGTH</code> returns the number of fields in the record (not the length of the record).</td></tr>
<tr><td><code>AT END OF FILE</code></td><td>AT END OF FILE Clause

This clause can only be used in conjunction with the <code>ONCE</code> option. If the <code>ONCE</code> option is used, this clause is specified to indicate the action to be taken when an end-of-file condition is detected.

If the <code>ONCE</code> option is not used, an end-of-file condition is handled like a normal processing loop termination.</td></tr>
<tr><td><code>END-WORK</code></td><td>End of READ WORK FILE Statement:</td></tr>
</table>

Syntax Element	Description
	In structured mode with processing loop, the Natural reserved word `END-WORK` must be used to end the `READ WORK FILE` statement.
`LOOP`	**End of READ WORK FILE Statement:** In reporting mode with processing loop, the Natural statement `LOOP` must be used to end the `READ WORK FILE` statement.

Overview of `RECORD` Option Usage

`RECORD` option is used with	rejected at compile time	rejected at runtime	`RECORD` option is ignored, processing switches to `SELECT` mode
work file type `ENTIRECONNECTION`		x	
dynamic variables	x		
work file type `CSV`			x
work file type `PORTABLE`			x
work file types `ASCII`, `ASCII-COMPRESSED`, `CSV`, `UNFORMATTED`, code page is specified in Configuration Utility (conversion is necessary) or at least one Unicode field is specified (operand of format U, conversion is necessary)			x

Field Lengths

The field lengths in the ***Operand Definition Table*** are determined as follows:

Format	Length
A, B, I, F	The number of bytes in the input record is the same as the internal length definition.
N	The number of bytes in the input record is the sum of internal positions before and after the decimal point. The decimal point and sign do not occupy a byte position in the input record.
P, D, T	The number of bytes in the input record is the sum of positions before and after the decimal point plus 1 for the sign, divided by 2 rounded upwards.
L	1 byte is used. For C format fields, 2 bytes are used.

Examples of Field Lengths:

Field Definition	Input Record
`#FIELD1 (A10)`	10 bytes
`#FIELD2 (B15)`	15 bytes
`#FIELD3 (N1.3)`	4 bytes
`#FIELD4 (N0.7)`	7 bytes
`#FIELD5 (P1.2)`	2 bytes
`#FIELD6 (P6.0)`	4 bytes

See also *Format and Length of User-Defined Variables* in the *Programming Guide*.

Variable Index Range

When reading an array from a work file, you can specify a variable index range for the array. For example:

```
READ WORK FILE work-file-number #ARRAY (I:J)
```

Handling of Large and Dynamic Variables

Work File Type	Handling
`ASCII` `ASCII-COMPRESSED`	The work file types `ASCII`, `ASCII-COMPRESSED` can handle dynamic and large variables with a maximum field/record length of 32766 bytes. Reading a dynamic variable from an `ASCII` or `ASCII-COMPRESSED` work file puts the rest of the work file record into the variable. Thus, for work files with these types, the dynamic variable is resized in each execution of the `READ WORK FILE` statement to match the exact length of the remaining part of the record.
`SAG` (binary)	The work file type `SAG` (binary) cannot handle dynamic variables and will produce an error. It can, however, handle large variables with a maximum field/record length of 32766 bytes.
`ENTIRECONNECTION`	The work file type `ENTIRECONNECTION` cannot handle dynamic variables. It can, however, handle large variables with a maximum field/record length of 107341824 bytes.
`PORTABLE` `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.

Work File Type	Handling
	Reading a dynamic variable from a PORTABLE work file leads to resizing to the stored length.
CSV	The maximum field/record length is 32766 bytes for dynamic and large variables. Dynamic variables are supported. X-arrays are not allowed and will result in an error message.

Handling of X-Arrays

When the ADJUST clause is *not* used, X-arrays are treated the same as regular arrays; that is, their existing occurrences are filled.

When the ADJUST clause is used, a one-dimensional X-array specified with complete range (*) is processed as shown in the table:

Work File Type	Handling
ASCII ASCII-COMPRESSED SAG (binary) CSV	A one-dimensional X-array specified with complete range (*) is expanded to receive all data from the rest of the record.
UNFORMATTED	A one-dimensional X-array specified with complete range (*) is expanded to receive all data from the rest of the file.
PORTABLE ENTIRECONNECTION	X-arrays are not supported.

Examples

- Example 1 - READ WORK FILE
- Example 2 - READ WORK FILE ASCII with Dynamic Variable
- Example 3 - READ WORK FILE Unformatted with Dynamic Variable
- Example 4 - READ WORK FILE ASCII with X-array and ADJUST its Occurrences
- Example 5 - READ WORK FILE Unformatted with X-array and ADJUST its Occurrences

- Example 6 - READ WORK FILE with Numeric CONST Variable as Work File Number

Example 1 - READ WORK FILE

```
** Example 'RWFEX1': READ WORK FILE
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
*
1 #RECORD
  2 #PERS-ID (A8)
  2 #NAME    (A20)
END-DEFINE
*
FIND EMPLOY-VIEW WITH CITY = 'STUTTGART'
  WRITE WORK FILE 1
        PERSONNEL-ID NAME
END-FIND
*
* ...
*
READ WORK FILE 1 RECORD #RECORD
  DISPLAY NOTITLE #PERS-ID #NAME
END-WORK
*
END
```

Output of Program RWFEX1:

```
#PERS-ID        #NAME
-------- --------------------

11100328 BERGHAUS
11100329 BARTHEL
11300313 AECKERLE
11300316 KANTE
11500304 KLUGE
11500308 DIETRICH
11500318 GASSNER
11500343 ROEHM
11600303 BERGER
11600320 BLAETTEL
11500336 JASPER
11100330 BUSH
11500328 EGGERT
```

Example 2 - READ WORK FILE ASCII with Dynamic Variable

```
** Example 'RWFEX2': READ WORK FILE - ASCII with dynamic variable
********************************************************************
DEFINE DATA LOCAL
  1 #DYNA (A) DYNAMIC
END-DEFINE
*
DEFINE WORK FILE 1 TYPE 'ASCII'
*
WRITE WORK FILE 1 VARIABLE 'text1 text2 text3 '
WRITE WORK FILE 1 VARIABLE 'text4 text5'
*
READ WORK FILE 1 AND SELECT #DYNA
  DISPLAY *LENGTH(#DYNA) #DYNA (AL=40)
  /*
  /* Length: 18  Dyn.Var: 'text1 text2 text3'
  /* Length: 11  Dyn.Var: 'text4 text5'
END-WORK
*
END
```

Output of Program RWFEX2:

```
Page      1                                                   11-07-15  09:21:09

  LENGTH                    #DYNA
----------- ----------------------------------------

         18 text1 text2 text3
         11 text4 text5
```

Example 3 - READ WORK FILE Unformatted with Dynamic Variable

```
** Example 'RWFEX3': READ WORK FILE - Unformatted with dynamic variable
************************************************************************
DEFINE DATA LOCAL
  1 #DYNA (A) DYNAMIC
END-DEFINE
*
DEFINE WORK FILE 1 TYPE 'UNFORMATTED'
*
WRITE WORK FILE 1 VARIABLE 'text1 text2 text3 '
WRITE WORK FILE 1 VARIABLE 'text4 text5'
*
DEFINE WORK FILE 1 TYPE 'UNFORMATTED'
*
READ WORK FILE 1 AND SELECT #DYNA
  DISPLAY *LENGTH(#DYNA) #DYNA (AL=40)
  /*
```

```
  /* Length: 29  Dyn.Var: 'text1 text2 text3 text4 text5'
END-WORK
*
END
```

Output of Program RWFEX3:

```
Page      1                                                  11-07-15  09:31:04

  LENGTH                    #DYNA
----------- ----------------------------------------

         29 text1 text2 text3 text4 text5
```

Example 4 - READ WORK FILE ASCII with X-array and ADJUST its Occurrences

```
** Example 'RWFEX4': READ WORK FILE - ASCII with X-array
**                                     and ADJUST its occurrences
********************************************************************
DEFINE DATA LOCAL
  1 #ARR (A6/1:*)
  1 #OCC (I4)
END-DEFINE
*
DEFINE WORK FILE 1 TYPE 'ASCII'
*
WRITE WORK FILE 1 VARIABLE 'text1 text2 text3 '
WRITE WORK FILE 1 VARIABLE 'text4 text5'
*
READ WORK FILE 1 AND SELECT #ARR(*) AND ADJUST OCCURRENCES
  #OCC := *OCCURRENCE(#ARR)
  DISPLAY #OCC #ARR(1:#OCC)
  /*
  /* Occurrences: 3 Array(*): 'text1', 'text2', 'text3'
  /* Occurrences: 2 Array(*): 'text4', 'text5'
END-WORK
*
END
```

Output of Program RWFEX4:

```
Page      1                                                  11-07-15  09:36:13

   #OCC      #ARR
----------- ------

          3 text1
            text2
            text3
```

```
          2 text4
            text5
```

Example 5 - READ WORK FILE Unformatted with X-array and ADJUST its Occurrences

```
** Example 'RWFEX5': READ WORK FILE - Unformatted with X-array
**                                      and ADJUST its occurrences
************************************************************************
DEFINE DATA LOCAL
  1 #ARR (A6/1:*)
  1 #OCC (I4)
END-DEFINE
*
DEFINE WORK FILE 1 TYPE 'UNFORMATTED'
*
WRITE WORK FILE 1 VARIABLE 'text1 text2 text3 '
WRITE WORK FILE 1 VARIABLE 'text4 text5'
*
DEFINE WORK FILE 1 TYPE 'UNFORMATTED'
*
READ WORK FILE 1 AND SELECT #ARR(*) AND ADJUST OCCURRENCES
  #OCC := *OCCURRENCE(#ARR)
  DISPLAY #OCC #ARR(1:#OCC)
  /*
  /*Occurrences: 5 Array(*): 'text1', 'text2', 'text3', 'text4', 'text5'
END-WORK
*
END
```

Output of Program RWFEX5:

```
Page      1                                                  11-07-15  09:41:25

   #OCC      #ARR
----------- ------

          5 text1
            text2
            text3
            text4
            text5
```

Example 6 - READ WORK FILE with Numeric CONST Variable as Work File Number

```
** Example 'RWFEX6': READ WORK FILE - with numeric CONST variable as
**                                    work file number
** see similar example RWFEX5 with numeric constant
************************************************************************
DEFINE DATA LOCAL
  1 #ARR    (A6/1:*)
  1 #OCC    (I4)
  1 #WF-1   (N4) CONST<1>
END-DEFINE
*
DEFINE WORK FILE #WF-1 TYPE 'UNFORMATTED'
*
WRITE WORK FILE  #WF-1 VARIABLE 'text1 text2 text3 '
WRITE WORK FILE  #WF-1 VARIABLE 'text4 text5'
*
DEFINE WORK FILE #WF-1 TYPE 'UNFORMATTED'
*
READ WORK FILE #WF-1 AND SELECT #ARR(*) AND ADJUST OCCURRENCES
  #OCC := *OCCURRENCE(#ARR)
  DISPLAY #OCC #ARR(1:#OCC)
  /*
  /*Occurrences: 5 Array(*): 'text1', 'text2', 'text3', 'text4', 'text5'
END-WORK
*
END
```

Output of Program RWFEX6:

```
Page      1                                                   21-12-20  17:42:43

   #OCC      #ARR
----------- ------

          5 text1
            text2
            text3
            text4
            text5
```

111 READLOB

```
READLOB  [ { ALL        } ]  [IN] [FILE] view-name
           { (operand1) }
 [PASSWORD=operand2]
 [CIPHER=operand3]
 [[WITH] ISN [=] operand4]
 [[STARTING] [AT] OFFSET [=] operand5]
 statement ...
END-READLOB (structured mode only)
LOOP        (reporting mode only)
```

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

Related Statements: READ | FIND | GET | UPDATELOB

Belongs to Function Group: ***Database Access and Update***

Function

The READLOB statement is used on a single record, where the defined LOB field (Large OBject field) is read in fixed length segments during the loop processing. It is only applicable to read this LOB field.

At loop beginning, the offset inside the LOB field is set from where to get the first data. On the next loop iteration, the segment following the last segment is returned. If the LOB data end is reached, the loop terminates.

This statement causes a processing loop to be initiated. See also *Loop Processing* in the *Programming Guide*.

Restrictions

The READLOB statement can only be used for access to Adabas databases.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S						N	P	I	B *								yes	no
operand2	C	S				A													yes	no
operand3	C	S						N											yes	no
operand4	C	S						N	P	I	B *								yes	no
operand5	C	S						N	P	I	B *								yes	no

* Format B of *operand1*, *operand4* and *operand5* may be used with a length of less than or equal to 4.

Syntax Element Description:

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td>operand1</td><td>Number of LOB Segments to Be Read:

The number of loop executions to be performed may be limited by specifying operand1 (enclosed in parentheses, immediately after the keyword <code>READLOB</code>) - either as a numeric constant (<code>0</code> - <code>4294967295</code>) or as the name of a numeric variable.

Example:

<pre>READLOB (5) IN FILE VIEW01 ...</pre><pre>#CNT := 10
READLOB (#CNT) IN FILE VIEW01 ...</pre>For this statement, the specified limit has priority over a limit set with a <code>LIMIT</code> statement. If a smaller limit is set with the profile or session parameter <code>LT</code>, the <code>LT</code> limit applies.

Note: operand1 is evaluated when the <code>READLOB</code> is started. If the value of operand1 is modified within the <code>READLOB</code> loop, this does not affect the number of loop iterations.</td></tr>
<tr><td><code>ALL</code></td><td>ALL Option:

To emphasize that the LOB data is to be read until its end, you can optionally specify the keyword <code>ALL</code>.

The <code>ALL</code> option is used by default if operand1 and <code>ALL</code> are omitted.</td></tr>
</table>

Syntax Element	Description
view-name	**View Name:** As *view-name*, you specify the name of a view, which must have been defined either within a DEFINE DATA statement or outside the program in a global or local data area. - The view has to contain just a single-valued LOB field, additional fields are not allowed. - If the LOB is a MU or PE field, a unique occurrence must be specified; a range notation is not allowed. - The LOB field must be defined in the DDM with a fixed (non-dynamic) length (which represents the segment length of the LOB field).
PASSWORD=*operand2* CIPHER=*operand3*	**PASSWORD and CIPHER Clauses:** The PASSWORD clause is used to provide a password when retrieving data from a file which is password-protected. The CIPHER clause is used to provide a cipher key when retrieving data from a file which is enciphered. See the statements FIND and PASSW for further information.
WITH ISN=*operand4*	**WITH ISN Option:** This option is used to specify the ISN of the record which is accessed by the READLOB statement. During the complete loop execution, only this record is fetched. *operand4* must be provided either in the form of a numeric constant or as a user-defined variable, or via the Natural system variable *ISN. The field is not modified by the READLOB execution. **Note:** *operand4* is evaluated when the READLOB is started. If the value of *operand4* is modified within the READLOB loop, this does not affect the record being fetched. If this option is omitted, the *ISN field of the last active database statement is used by default.
STARTING AT OFFSET=*operand5*	**STARTING AT OFFSET Clause:** Provides the start offset within the LOB field, where the first segment read begins. The first byte of the LOB field is offset zero (0). *operand5* must be provided either in the form of a numeric constant or as a user-defined variable, without precision digits. The field is not modified by the READLOB execution. If this clause is omitted, start offset (0) is assumed, which causes the LOB field to be read from the beginning.

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td></td><td>See also *NUMBER during the processing described in System Variables Available with READLOB below.</td></tr>
<tr><td>END-READLOB</td><td rowspan="2">End of READLOB Statement:

In structured mode, the Natural reserved keyword END-READLOB must be used to end the READLOB statement.

In reporting mode, the Natural statement LOOP is used to end the READLOB statement.</td></tr>
<tr><td>LOOP</td></tr>
</table>

System Variables Available with READLOB

The Natural system variables *ISN, *COUNTER and *NUMBER are provided with the READLOB statement.

The format/length of these system variables is P10. This format/length cannot be changed.

The purpose of the Natural system variables, when used with the READLOB statement, is explained below:

<table>
<tr><th>System Variable</th><th colspan="2">Explanation</th></tr>
<tr><td>*ISN</td><td colspan="2">Contains the Adabas ISN of the record currently being processed. Since a READLOB statement always makes access to the same record, the *ISN value returned is the same over all loop iterations.</td></tr>
<tr><td>*COUNTER</td><td colspan="2">Contains the number of times the processing loop has been passed through.</td></tr>
<tr><td rowspan="3">*NUMBER</td><td>Before the call:</td><td>It specifies the byte offset in the LOB field from which the segment is to be read. Value zero (0) represents the leftmost byte in the LOB field.

This does not apply for the first loop iteration. In this case the read offset is determined by the STARTING AT OFFSET clause.</td></tr>
<tr><td>After the call:</td><td>If data was found (that is, the offset was less than the LOB field length), it receives the offset plus the segment length. This may lead to a *NUMBER value which is higher than the length of the entire LOB field.

If no data was found (that is, the offset was higher or equal to the LOB field length), the value of *NUMBER remains unchanged.</td></tr>
<tr><td colspan="2">If a consecutive read over a LOB field is requested, the *NUMBER value must not be modified within the READLOB, as it contains the offset to exactly continue with the next segment in the subsequent loop iteration. However, if a continuation at another place within the LOB field is desired (re-position), you may change the *NUMBER value to this offset. If *NUMBER is reset, this</td></tr>
</table>

System Variable	Explanation
	leads to the next segment coming from the LOB start. If *NUMBER is incremented by (*n*), this number of bytes will be skipped in the LOB field processing.

Functional Considerations

- The READLOB statement always reads the record without hold. In order to guarantee stability on the LOB data (that is, to prevent an update by other users) while the READLOB browses over the LOB field, the record can be set into hold with the database statement providing the ISN; either
  - into an *exclusive hold*, due to an UPDATE or DELETE referring to an outer READ or FIND statement or
  - into a *shared hold* with an explicit IN SHARED HOLD option applied in a READ or FIND statement. If the additional subparameter MODE=Q is used, the record is automatically released from hold if the next record is fetched in the read sequence.
- Since the READLOB statement always reads the record without hold, an UPDATE, DELETE or GET SAME statement must not refer to a READLOB statement.

Examples

Example 1 - Get Record Number from READ Loop

```
DEFINE DATA LOCAL
1 VIEW01 VIEW OF ..
  2 NAME
  2 L@LOBFIELD
1 VIEW02 VIEW OF ..
  2 LOBFIELD_SEGMENT               /* LOB field defined in DDM with (A1000).
END-DEFINE
*
READ VIEW01 BY NAME = 'SMITH'      /* Outer statement reads all demanded record
                                   /* fields, except the LOB field.
     IN SHARED HOLD MODE=Q         /* Set record into shared hold to enforce LOB
                                   /* data stability during READLOB.
  DISPLAY NAME 'Total-length LOB-field' L@LOBFIELD
  READLOB VIEW02                   /* Record number used from active record of
                                   /* READ statement.
```

```
                                    /* LOB is read in segments with length 1000.
      STARTING AT OFFSET = 2000     /* Start to read the LOB field at byte 2000.
    WRITE 'Loop counter:' *COUNTER  10X ' Next offset:' *NUMBER
    PRINT VIEW02.LOBFIELD_SEGMENT
  END-READLOB
END-READ
END
```

Example 2 - Get Record Number by User-defined Value

```
DEFINE DATA LOCAL
1 #ISN  (I4)
1 #CNT  (I4)
1 #OFF  (I4)
1 VIEW02 VIEW OF ..
  2 LOBFIELD_SEGMENT            /* LOB field defined in DDM with (A1000).
END-DEFINE
*
INPUT (AD=T)
  / ' Read record (ISN):' #ISN
  / 'Number of segments:' #CNT
  / '   Start at offset:' #OFF
*
READLOB (#CNT) VIEW02           /* Read max. (#CNT) segments with length 1000.
    WITH ISN = #ISN             /* Record number provided by user.
                                /* Record is not in hold.
    STARTING AT OFFSET = #OFF   /* Start to read the LOB field at byte (#OFF).
  WRITE 'Loop counter:' *COUNTER  10X ' Next offset:' *NUMBER
  PRINT VIEW02.LOBFIELD_SEGMENT
END-READLOB
END
```

Example 3 - Get Record Number from READ Loop (with Exclusive Hold)

```
DEFINE DATA LOCAL
1 VIEW01 VIEW OF ..
  2 NAME
1 VIEW02 VIEW OF ..
  2 LOBFIELD_SEGMENT           /* LOB field defined in DDM with (A1000).
END-DEFINE
*
R1. READ VIEW01 BY NAME = 'SMITH'/* Outer statement reads all demanded
                                 /* record fields, except the LOB field.
  DISPLAY NAME
  READLOB VIEW02             /* Record number from active record of READ.
                             /* LOB is read in segments with length 1000.
      STARTING AT OFFSET = 2000 /* Start to read LOB field at byte 2000.
    WRITE 'Loop counter:' *COUNTER 10X ' Next offset:' *NUMBER
    PRINT VIEW02.LOBFIELD_SEGMENT
  END-READLOB
```

```
  ...
  UPDATE (R1.)            /* Set record into exclusive hold that
                          /* enforces LOB data stability during READLOB.
  END OF TRANSACTION
END-READ
END
```

112 REDEFINE

```
REDEFINE { operand1 ( { nX       } ... ) } ...
                      { operand2 }
```

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

Belongs to Function Group: **Reporting Mode Statements**

Function

The `REDEFINE` statement is used to redefine a field. The resulting definition may consist of one or more user-defined variables.

With one `REDEFINE` statement, several fields may be redefined.

Restriction

The `REDEFINE` statement is only valid in reporting mode. To redefine a field in structured mode, use the `REDEFINE` clause of the `DEFINE DATA` statement.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1		S	A	G		A	U	N	P	I	F	B	D	T	L	C			yes	no
operand2		S	A	G		A		N	P	I	F	B	D	T	L	C			yes	yes

Syntax Element Description:

Syntax Element	Description
`REDEFINE` *operand1* *operand2*	**Method of Redefinition:** The byte positions of *operand1* are redefined from left to right regardless of format. The format of *operand2* can be different from the format of *operand1*. However, the data at the byte positions of *operand2* should match the format specification of *operand2* to avoid strange results in the output report. For example, if an alphanumeric field is redefined as numeric and does not contain numeric data

Syntax Element	Description
	according to the format specification, an abnormal termination may result when it is used. **Further Redefinition:** Fields defined using a `REDEFINE` statement may be subsequently redefined with another `REDEFINE` statement.
*n*X	**Filler Notation:** The *n*X notation is used to denote filler bytes within the field/variable being redefined. Any trailing *n*X notation is optional.

Examples

- Example 1
- Example 2
- Example 3
- Example 4

Example 1

The user-defined variable `#A` (format/length A10) contains the value `123ABCDEFG`.

```
REDEFINE #A (#A1(N3) #A2(A7))
```

The value in `#A1` is `123`. The value in `#A2` is `ABCDEFG`.

Example 2

The user-defined variable `#B` (format/length A10) contains the value (shown in hexadecimal format) `12345CC1C2C3C4C5C6C7`.

```
REDEFINE #B (#B1(P4) #B2(A7))
```

The value in `#B1` is `12345C` (in hexadecimal format).

The value in `#B2` is `C1C2C3C4C5C6C7` (in hexadecimal format).

```
REDEFINE #B (#BB1(B2)8X)) or REDEFINE #B(#BB1(B2))
```

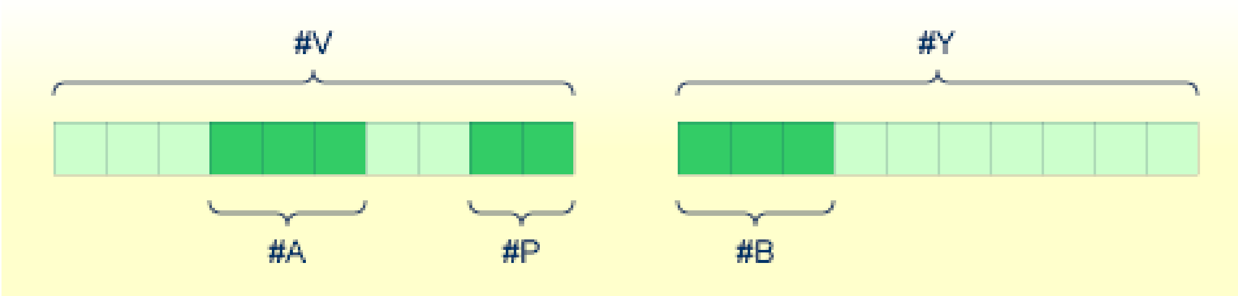
The value in `#BB1` is `1234` (in hexadecimal format).

Note: For packed data (Format P), the number of decimal positions required must be specified. The following formula can be used to determine the number of bytes that the packed number occupies:

`Number of bytes = (number of decimal positions + 1) / 2`, rounded upwards to full bytes.

Example 3

```
COMPUTE #V (N8.2) = #Y (N10) = ...
REDEFINE #V (3X  #A(N3)  2X  #P (N2))  #Y (#B(N3)  7X)
```



Example 4

This example redefines the value of the system variable `*DATN`, which is in the form `YYYYMMDD`, and displays the result as three separate fields in the order day/month/year:

```
MOVE *DATN TO #DATINT (N8)
REDEFINE #DATINT (#YEAR (N4) #MONTH (N2) #DAY (N2))
DISPLAY NOTITLE #DATINT #DAY #MONTH #YEAR
END
```

Output:

```
 #DATINT  #DAY #MONTH #YEAR
--------- ---- ------ -----


 20140326  26    3     2014
```

113 REDUCE

```
REDUCE  { dynamic-clause }  [GIVING operand5]
        { array-clause   }
```

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

Related statements: `EXPAND` | `RESIZE`

Belongs to Function Group: ***Memory Management Control for Dynamic Variables or X-Arrays***.

Function

The `REDUCE` statement is used to reduce:

- the allocated length of a dynamic variable (*`dynamic-clause`*), or
- the number of occurrences of X-arrays (*`array-clause`*).

For further information, see also the following sections in the *Programming Guide*:

- *Using Dynamic Variables*
- *Allocating/Freeing Memory Space for a Dynamic Variable*
- *X-Arrays*
- *Storage Management of X-Group Arrays*

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1		S	A			A	U					B							no	no
operand2	C	S								I									no	no
operand3			A	G		A	U	N	P	I	F	B	D	T	L	C	G	O	yes	no
operand4	C	S						N	P	I									no	no
operand5		S								I4									no	yes

Syntax Element Description:

Syntax Element	Description
dynamic-clause	**Dynamic Clause:** The `REDUCE DYNAMIC VARIABLE` statement reduces the allocated length of a dynamic variable (*operand1*) to the length specified (*operand2*). For further information, see ***Dynamic Clause*** below.
operand1	**Dynamic Variable:** *operand1* is the dynamic variable for which the length is to be reduced.
operand2	**Target Length of Dynamic Variable:** *operand2* is used to specify the length to which the dynamic variable is to be reduced. The value specified must be a non-negative integer constant or a variable of type Integer 4 (I4).
array-clause	**Array Clause:** The `REDUCE ARRAY` statement reduces the number of occurrences of the X-array (*operand3*) to the upper and lower bound specified with `(dim[,dim[,dim]])`. For further information, see ***Array Clause*** below.
operand3	**X-Array:** *operand3* is the X-array. The occurrences of the X-array can be reduced. The index notation of the array is optional. As index notation only the complete range notation * is allowed for each dimension.
dim *operand4*	**Dimension:** The lower and upper bound notation (*operand4* or asterisk) to which the X-array should be reduced is specified here. If the current value of the upper or lower bound should be used, an asterisk (*) must be specified instead of *operand4*. For further information, see ***Dimension*** below.
GIVING *operand5*	**GIVING Clause:** If the `GIVING` clause is not specified, Natural runtime error processing is triggered if an error occurs. If the `GIVING` clause is specified, *operand5* contains the Natural message number if an error occurred, or zero upon success.

Dynamic Clause

```
[SIZE OF] DYNAMIC [VARIABLE] operand1 TO operand2
```

The `REDUCE DYNAMIC VARIABLE` statement reduces the allocated length of a dynamic variable (*operand1*) to the length specified (*operand2*). The allocated memory of the dynamic variable which is beyond the given length is released immediately, i.e., when the statement is executed.

If the currently allocated length (`*LENGTH`) of the dynamic variable is greater than the given length, `*LENGTH` is set to the given length and the content of the variable is truncated (but not modified). If the given length is larger than the currently allocated length of the dynamic variable, the statement will be ignored.

Array Clause

$$[\texttt{OCCURRENCES OF}]\ \texttt{ARRAY}\ \textit{operand3}\ \texttt{TO} \left\{ \begin{array}{l} \texttt{0} \\ (\textit{dim}[,\textit{dim}[,\textit{dim}]]) \end{array} \right\}$$

If `REDUCE TO 0` (zero) is specified, all occurrences of the X-array are released. In other words, the whole array is reduced.

The `REDUCE ARRAY` statement reduces the number of occurrences of the X-array (*operand3*) to the upper and lower bound specified with `TO (dim[,dim[,dim]])`.

An upper or lower bound used in a `REDUCE` statement must be exactly the same as the corresponding upper or lower bound defined for the array.

Example:

```
DEFINE DATA LOCAL
1 #a(I4/1:*)
1 #g(1:*)
  2 #ga(I4/1:*)

1 #i(i4)
END-DEFINE
...


*/ reducing #a (1:10)

REDUCE ARRAY #a TO (1:10)       /* #a is reduced
REDUCE ARRAY #a TO (*:10)       /* to 10 occurrences.


*/ reducing #ga (1:10,1:20)
```

```
REDUCE ARRAY #g TO (1:10)       /* 1st dimension is set to (1:10)
REDUCE ARRAY #ga TO (*:*,1:20)  /* 1st dimension is dependent and
                                /* therefore kept with (*:*)
                                /* 2nd dimension is set to (1:20)

REDUCE ARRAY #a TO (5:10)       /* This is rejected because the lower index
                                /* must be 1 or *
REDUCE ARRAY #a TO (#i:10)      /* This is rejected because the lower index
                                /* must be 1 or *

REDUCE ARRAY #ga TO (1:10,1:20) /* (1:10) for the 1st dimension is rejected
                                /* because the dimension is dependent and
                                /* must be specified with (*:*).
```

For further information, see

- *Storage Management of X-Arrays*
- *Storage Management of X-Group Arrays*

Dimension

Each of the dimensions (`dim`) specified in the ***Array Clause*** is defined using the following syntax:

$$\left\{\begin{array}{l} * \\ \left\{\begin{array}{l} * \\ \mathit{operand4} \end{array}\right\} : \left\{\begin{array}{l} * \\ \mathit{operand4} \end{array}\right\} \end{array}\right\}$$

The lower and upper bound notation (`operand4` or asterisk) to which the X-array should be reduced is specified here. If the current value of the upper or lower bound should be used, an asterisk (*) may be specified in place of `operand4`. In place of `*:*`, you may also specify a single asterisk.

The number of dimensions (`dim`) must exactly match the defined number of dimensions of the X-array (1, 2 or 3).

When using the `REDUCE` statement, it is only possible to decrease the number of occurrences. If the requested number is larger than the currently allocated number of occurrences, it will simply be ignored.

114 REINPUT

```
REINPUT   [FULL] [(statement-parameters)]  { USING HELP       }
                                            { WITH-TEXT-option }
          [MARK-option]
          [ALARM-option]
```

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

Related Statements: DEFINE WINDOW | INPUT | SET WINDOW

Belongs to Function Group: ***Screen Generation for Interactive Processing***

Function

The REINPUT statement is used to return to and re-execute an INPUT statement. It is generally used to display a message indicating that the data input as a result of the previous INPUT statement were invalid. See ***Example 1***.

No WRITE or DISPLAY statements may be executed between an INPUT statement and its corresponding REINPUT statement. The REINPUT statement is not valid in batch mode.

The REINPUT statement, when executed, repositions the program status regarding subroutine, special condition and loop processing as it existed when the INPUT statement was executed (as long as the status of the INPUT statement is still active). If the loop was initiated after the execution of the INPUT statement and the REINPUT statement is within this loop, the loop will be discontinued and then restarted after the INPUT statement has been reprocessed as a result of REINPUT.

If a hierarchy of subroutines was invoked after the execution of the INPUT statement, and the REINPUT is performed within a subroutine, Natural will trace back all subroutines automatically and reposition the program status to that of the INPUT statement.

It is not possible, however, to have an INPUT statement positioned within a loop, a subroutine or a special condition block, and then execute the REINPUT statement when the status under which the INPUT statement was executed has already been terminated. An error message will be produced and program execution terminated when this error condition is detected.

Note: The execution of a REINPUT statement (without FULL option) does not reset the MODIFIED status of an attribute control variable used in the corresponding INPUT statement. To check if an attribute control variable has been assigned the status MODIFIED, use the MODIFIED option.

Syntax Description

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td><code>REINPUT FULL</code></td><td>
FULL Option:

If you specify the <code>FULL</code> option in a <code>REINPUT</code> statement, the corresponding <code>INPUT</code> statement will be re-executed fully:

With an ordinary <code>REINPUT</code> statement (without <code>FULL</code> option), the contents of variables that were changed between the <code>INPUT</code> and <code>REINPUT</code> statement will not be displayed; that is, all variables on the screen will show the contents they had when the <code>INPUT</code> statement was originally executed.
With a <code>REINPUT FULL</code> statement, all changes that have been made after the initial execution of the <code>INPUT</code> statement will be applied to the <code>INPUT</code> statement when it is re-executed; that is, all variables on the screen contain the values they had when the <code>REINPUT</code> statement was executed.

Note: The contents of input-only fields (<code>AD=A</code>) will be deleted again by <code>REINPUT FULL</code>.

Another characteristic of the <code>REINPUT FULL</code> statement is that the status of attribute control variables is reset to <code>NOT MODIFIED</code>. This is not done with the ordinary <code>REINPUT</code> statement. To check if an attribute control variable has been assigned the status <code>MODIFIED</code>, use the MODIFIED option.

See also Example 3 - REINPUT FULL WITH MARK POSITION.
</td></tr>
<tr><td>statement-parameters</td><td>
Parameters:

Parameters specified in a <code>REINPUT</code> statement will be applied to all fields specified in the statement.

Any parameter specified at element (field) level (see MARK Option) will override any corresponding parameter at statement level.
<table>
<tr><td>Parameters that can be specified with the <code>REINPUT</code> statement:</td><td>Specification (S = at statement level, E = at element level)</td></tr>
<tr><td><code>AD</code> - Attribute Definition *</td><td>SE</td></tr>
<tr><td><code>CD</code> - Color Definition</td><td>S</td></tr>
<tr><td colspan="2">* If <code>AD=P</code> is specified at statement level, all fields - except those used in the <code>MARK</code> option - are protected.</td></tr>
</table>
The individual session parameters are described in the Parameter Reference.
</td></tr>
<tr><td><code>USING HELP</code></td><td>
USING HELP Option:

This option causes the helproutine defined for the <code>INPUT</code> map to be invoked.
</td></tr>
</table>

Syntax Element	Description
	USING HELP used in combination with the MARK option causes the helproutine defined for the first field specified in the MARK option to be invoked. If no helproutine is defined for that field, the helproutine for the map will be invoked. Example: `REINPUT USING HELP MARK 3` As a result, the helproutine defined for the third field in the INPUT map will be invoked.
WITH-TEXT-option	**WITH TEXT Option:** The WITH TEXT option is used to provide text which is to be displayed in the message line. See ***WITH TEXT Option*** below.
MARK-option	**MARK Option** With the MARK option, you can mark a specific field, that is, specify a field in which the cursor is to be placed when the REINPUT statement is executed. See ***MARK Option*** below.
ALARM-option	**ALARM Option:** This option causes the sound alarm feature of the terminal to be activated when the REINPUT statement is executed. See ***ALARM Option*** below.

WITH TEXT Option

WITH TEXT is used to provide text which is to be displayed in the message line. This is usually a message indicating what action should be taken to process the screen or to correct an error.

```
[WITH] [TEXT] { *operand1 } [(attributes)] [,operand3]...7
              {  operand2 }
```

Operand Definition Table:

Operand	Possible Structure				Possible Formats	Referencing Permitted	Dynamic Definition
operand1	C	S			N P I B*	yes	no
operand2	C	S			A U	yes	no
operand3	C	S			A U N P I F B D T L	yes	no

[*] Format B of *operand1* may be used only with a length of less than or equal to 4.

Syntax Element Description:

Syntax Element	Description
operand1	**Message Text from Natural Message File:** *operand1* represents the number of a message text that is to be retrieved from a Natural message file. You can retrieve either user-defined messages or Natural system messages: ■ If you specify a positive value of up to four digits (for example: `954`), you will retrieve user-defined messages. ■ If you specify a negative value of up to four digits (for example: `-954`), you will retrieve Natural system messages. See also ***Example 4 - WITH TEXT Options***. Natural message files are created and maintained with the `SYSERR` utility as described in the relevant documentation.
operand2	**Message Text:** *operand2* represents the message to be placed in the message line. See also ***Example 4 - WITH TEXT Options***.
attributes	**Output Attributes:** It is possible to assign various output attributes for *operand1*/*operand2*. These attributes and the syntax that may be used are described in the section ***Output Attributes*** below.
operand3	**Dynamic Replacement of Message Text:** *operand3* represents a numeric or text constant or the name of a variable. The values provided are used to replace parts of a message text that are either specified with *operand1* or *operand2*. The notation `:`*n*`:` is used within the message text as a reference to *operand3* contents, where *n* represents the occurrence (1 - 7) of *operand3*. See also ***Example 4 - WITH TEXT Options***. **Note:** Multiple specifications of *operand3* must be separated from each other by a comma. If the comma is used as a decimal character (as defined with the session parameter `DC`) and numeric constants are specified as *operand3*, put blanks before and after the comma so that it cannot be misinterpreted as a decimal character. Alternatively, multiple specifications of *operand3* can be separated by the input delimiter character (as defined with the session parameter `ID`); however, this is not possible in the case of `ID=/` (slash). Leading zeros or trailing blanks will be removed from the field value before it is displayed in a message.

Output Attributes

`attributes` indicates the output attributes to be used for text display. Attributes may be:

```
{ AD=ad-value ... }
{ CD=cd-value ... }
{ CV=variable     } ...
```

For the possible session parameter values, refer to the corresponding sections in the *Parameter Reference* documentation:

- *AD - Attribute Definition*, section *Field Representation*
- *CD - Color Definition*
- *CV - Attribute Control Variable*

Note: The compiler actually accepts more than one attribute value for an output field. For example, you may specify: `AD=BDI`. In such a case, however, only the last value applies. In the given example, only the value `I` will become effective and the output field will be displayed intensified.

MARK Option

With the `MARK` option, you can mark a specific field, that is, specify a field in which the cursor is to be placed when the `REINPUT` statement is executed. You can also mark a specific position within a field. Moreover, you can make fields input-protected, and change their display and color attributes.

```
MARK [POSITION operand4 [IN]] [FIELD] { { operand5   } [(attributes)] } ...
                                      { { *fieldname }                }
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand4	C	S						N	P	I									yes	no
operand5	C	S	A					N	P	I									yes	no

Syntax Element Description:

Syntax Element	Description

operand5
**fieldname*
	Field to be Marked:
	
	All `AD=A` or `AD=M` (that is, non-protected) fields specified in an `INPUT` statement are sequentially numbered (beginning with 1) by Natural. *operand5* represents the number of the field in which the cursor is to be positioned.
	
	The **fieldname* notation is used to position to a field (as used in the `INPUT` statement) using the name of the field as a reference.
	
	If the corresponding `INPUT` field is an array, a unique index or an index range may be used to reference one or more occurrences of the array.
	
	`INPUT #ARRAY (A1/1:5)`
	`...`
	`REINPUT (AD=P) 'TEXT' MARK *#ARRAY (2:3)`
	
	If *operand5* is also an array, the values in *operand5* are used as field numbers for the `INPUT` array.
	
	`RESET #X(N2/1:2)`
	`INPUT #ARRAY ...`
	`...`
	`REINPUT (AD=P) 'TEXT' MARK #X (1:2)`

`MARK POSITION`	**MARK POSITION Option:** With this option, you can have the cursor placed at a specific position - as specified with *operand4* - within a field. See also ***Example 3 - REINPUT FULL WITH MARK POSITION***.
operand4	**Cursor Position:** *operand4* specifies the cursor position. *operand4* must not contain decimal digits.
attributes	**Attribute Assignments:** See ***Attribute Assignments*** below.

Attribute Assignments

With explicit attributes, you can define the display presentation and color of the `WITH TEXT` message and also the layout of the `MARK` field (which is positioned by the `REINPUT` statement).

```
[ AD=[P] [ B ] ]   [ CD= { BL } ]   [CV=operand6]
         [ C ]            { GR }
         [ D ]            { NE }
         [ I ]            { PI }
         [ N ]            { RE }
         [ U ]            { TU }
         [ V ]            { YE }
```

Operand Definition Table:

Operand	Possible Structure	Possible Formats	Referencing Permitted	Dynamic Definition
operand6	S	C	no	no

With the attribute `AD=P`, you can make an input field (`AD=A` or `AD=M`) input-protected.

Note: You cannot use an attribute to make output-only fields (`AD=O`) available for input.

For information on the attributes `AD`, `CD` and `CV`, refer to the *Parameter Reference*.

The attributes for the `WITH TEXT` and `MARK` fields need not be specified in a fixed manner, but can also be assigned dynamically by means of a control variable, which is referenced in a `(CV=`*operand6*`)` clause. See ***Example 5 - REINPUT with Attribute Assignment Using a Control Variable***.

If both an `AD` and a `CV` option are specified for the same field, the attributes from the `AD` option are completely ignored, except `(AD=P)` which remains in effect.

If a `CD` and a `CV` option are specified for the same field, the color from the `CV` option is used. If the `CV` variable contains no color specification, the color from the `CD` option is applied to that field.

If `AD=P` is specified at statement level, all fields except those specified in the `MARK` option are input-protected. See also ***Example 2 - REINPUT with Attribute Assignment***.

ALARM Option

```
[AND] [SOUND] ALARM
```

This option causes the sound alarm feature of the terminal to be activated when the `REINPUT` statement is executed. The appropriate hardware must be available to be able to use this feature.

Examples

Example 1 - REINPUT Statement

```
** Example 'REIEX1': REINPUT
************************************************************************
DEFINE DATA LOCAL
1 #FUNCTION (A1)
1 #PARM     (A1)
END-DEFINE
*
INPUT #FUNCTION #PARM
*
DECIDE FOR FIRST CONDITION
  WHEN #FUNCTION = 'A' AND #PARM = 'X'
    REINPUT 'Function A with parameter X selected.'
            MARK *#PARM
  WHEN #FUNCTION = 'C' THRU 'D'
    REINPUT 'Function C or D selected.'
  WHEN #FUNCTION = 'X'
    STOP
  WHEN NONE
    REINPUT 'Please enter a valid function.'
            MARK *#FUNCTION
END-DECIDE
*
END
```

Output of Program REIEX1:

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

And after pressing ENTER:

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

Example 2 - REINPUT with Attribute Assignment

```
** Example 'REIEX2': REINPUT  (with attributes)
************************************************************************
DEFINE DATA LOCAL
1 #A (A20)
1 #B (N7.2)
1 #C (A5)
1 #D (N3)
END-DEFINE
*
INPUT (AD=A) #A  #B  #C  #D
*
IF #A = ' ' OR #B = 0
  REINPUT (AD=P) 'RETYPE VALUES'
                 MARK *#A (AD=I CD=RE)    /* put cursor on first field
                      *#B (AD=U CD=PI)    /* and change colors
END-IF
*
END
```

Example 3 - REINPUT FULL with MARK POSITION

```
** Example 'REIEX3': REINPUT  (with FULL and POSITION option)
************************************************************************
DEFINE DATA LOCAL
1 #A (A20)
1 #B (N7.2)
1 #C (A5)
1 #D (N3)
END-DEFINE
*
INPUT (AD=M) #A  #B  #C  #D
*
IF #A = ' '
  COMPUTE #B = #B + #D
  RESET #D
END-IF
*
IF #A = SCAN 'TEST' OR = ' '
REINPUT FULL 'RETYPE VALUES' MARK POSITION 5 IN *#A
END-IF
*
END
```

Output of Program REIEX3:

```
RETYPE VALUES
#A                      #B         0.00 #C        #D     0
```

Example 4 - WITH TEXT Options

```
** Example 'REIEX4': REINPUT (with TEXT option)
************************************************************************
DEFINE DATA LOCAL
01 #NAME   (A8)
01 #TEXT   (A20)
END-DEFINE
*
*
INPUT WITH TEXT 'Enter a program name.' 'Program name:' #NAME
*
IF #NAME = ' '
  REINPUT WITH TEXT 'Input missing. Enter a name.'
END-IF
*
IF #NAME NE MASK (A)
  MOVE 'Invalid input.' TO #TEXT
  REINPUT WITH TEXT ':1: Name must start with a letter.',#TEXT
ELSE
  /* Using Natural error message 7600 for demonstration
  COMPRESS *INIT-USER 'on' *DAT4I INTO #TEXT
  INPUT WITH TEXT *-7600,#NAME,#TEXT 'Input accepted.'
END-IF
END
```

Example 5 - REINPUT with Attribute Assignment Using a Control Variable

```
DEFINE DATA LOCAL
1 #HELLO (A5)  INIT <'HELO'>
1 #VAR   (A20) INIT <'Enter "HELLO"'>
1 #CV (C)
END-DEFINE
*
INPUT (IP=OFF) #HELLO (AD=M)
*
IF #HELLO NE 'HELLO' THEN
  MOVE (AD=U CD=RE) TO #CV
  REINPUT FULL WITH TEXT #VAR (CD=YE)
               MARK *#HELLO   (CV=#CV)
END-IF
END
```

115 REJECT

For more information about this statement, see the statement `ACCEPT/REJECT`.

116 RELEASE

```
RELEASE { STACK                }
        { SETS [set-name...]   }
        { VARIABLES            }
```

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

Related Statements: `STACK` | `FIND` **with** `RETAIN` **option** | `DEFINE DATA GLOBAL`

Function

The `RELEASE` statement is used to:

- delete the entire contents of the Natural stack;
- release sets of ISNs retained via a `FIND` statement that contained a `RETAIN` clause (applicable to Adabas databases only);
- reset global and application-independent variables.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
set-name	C	S				A												no	no

Syntax Element Description:

Syntax Element	Description
`RELEASE STACK`	**RELEASE STACK Option:** Causes all data/commands currently in the Natural stack to be deleted.
`RELEASE SETS`	**RELEASE SETS Option:** Is applicable to Adabas databases only. If only `RELEASE SETS`, without a *set-name*, is specified, all ISN sets retained with a `FIND` statement with a `RETAIN` clause will be released.
`RELEASE SETS` *set-name*	Causes a specific single ISN set to be released.

Syntax Element	Description

	`RELEASE SET 'CITY-SET'`
	
	`MOVE 'CITY-SET' TO #SET(A32)`
	`RELEASE SET #SET`

`RELEASE VARIABLES`	**RELEASE VARIABLES Option:** Causes all variables defined in the current global data area (GDA) to be reset to their initial values. Also, it eliminates all application-independent variables (AIVs), thus making them no longer available. The `RELEASE VARIABLES` statement does not perform the reset/eliminate operations directly after execution. Instead, a signal is set first which triggers these operations when all Natural objects currently running have finished processing.

Example

```
** Example 'RELEX1': FIND (with RETAIN clause and RELEASE statement)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 BIRTH
  2 NAME
*
1 #BIRTH (D)
END-DEFINE
*
MOVE EDITED '19400101' TO #BIRTH (EM=YYYYMMDD)
*
FIND NUMBER EMPLOY-VIEW WITH BIRTH GT #BIRTH
     RETAIN AS 'AGESET1'
IF *NUMBER = 0
  STOP
END-IF
*
FIND EMPLOY-VIEW WITH 'AGESET1' AND CITY = 'NEW YORK'
  DISPLAY NOTITLE NAME CITY BIRTH (EM=YYYY-MM-DD)
END-FIND
*
RELEASE SET 'AGESET1'
*
END
```

Output of Program RELEX1:

```
NAME                 CITY            DATE
                                         OF
                                       BIRTH
-------------------- -------------------- ----------

RUBIN                NEW YORK             1945-10-27
WALLACE              NEW YORK             1945-08-04
```

117 REPEAT

Related Statements: FOR | ESCAPE

Belongs to Function Group: ***Loop Execution***

Function

The REPEAT statement is used to initiate a processing loop.

See also *Loop Processing* in the *Programming Guide*.

Syntax Description

Two different structures are possible for this statement.

- **Syntax 1** - Statements are executed one or more times
- **Syntax 2** - Statements are executed zero or more times

The placement of the logical condition (either at the beginning or at the end of the loop) determines when it is to be evaluated.

For further information on logical conditions, see the section *Logical Condition Criteria* in the *Programming Guide*.

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

Syntax 1:

```
REPEAT
   statement ...    [ { UNTIL } logical-condition ]
                      { WHILE }
END-REPEAT          (structured mode only)
LOOP                (reporting mode only)
```

Syntax 2:

```
REPEAT
    [ { UNTIL }  logical-condition ]    statement...
        { WHILE }
END-REPEAT          (structured mode only)
LOOP                (reporting mode only)
```

Syntax Element Description:

Syntax Element	Description
UNTIL	**UNTIL Option:** The processing loop will be continued until the logical condition becomes true.
WHILE	**WHILE Option:** The processing loop will be continued as long as the logical condition is true.
logical-condition	**Logical Condition:** If a logical condition is specified, the condition determines when the execution of the loop is to be terminated. If no logical condition is specified, the loop must be exited by an ESCAPE, STOP or TERMINATE statement specified within the loop. The syntax for a logical condition is described in the section *Logical Condition Criteria* in the Programming Guide.
END-REPEAT LOOP	**End of REPEAT Statement:** In structured mode, the Natural reserved word END-REPEAT must be used to end the REPEAT statement. In reporting mode, the Natural statement LOOP is used to end the REPEAT statement.

Examples

- Example 1 - REPEAT

- Example 2 - Using WHILE and UNTIL Options

Example 1 - REPEAT

```
** Example 'RPTEX1S': REPEAT (structured mode)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
*
1 #PERS-NR (A8)
END-DEFINE
*
REPEAT
  INPUT 'ENTER A PERSONNEL NUMBER:' #PERS-NR
  IF #PERS-NR = ' '
    ESCAPE BOTTOM
  END-IF
  /*
  FIND EMPLOY-VIEW WITH PERSONNEL-ID = #PERS-NR
    IF NO RECORD FOUND
      REINPUT 'NO RECORD FOUND'
    END-NOREC
    DISPLAY NOTITLE NAME
  END-FIND
END-REPEAT
*
END
```

Output of Program RPTEX1S:

```
ENTER A PERSONNEL NUMBER: 11500304
```

After entering and confirming personnel number:

```
        NAME
--------------------

KLUGE
```

Equivalent reporting-mode example: **RPTEX1R**.

Example 2 - Using WHILE and UNTIL Options

```
** Example 'RPTEX2S': REPEAT (with WHILE and UNTIL option)
************************************************************************
DEFINE DATA LOCAL
1 #X (I1) INIT <0>
1 #Y (I1) INIT <0>
END-DEFINE
*
REPEAT WHILE #X <= 5
  ADD 1 TO #X
  WRITE NOTITLE '=' #X
END-REPEAT
*
SKIP 3
REPEAT
  ADD 1 TO #Y
  WRITE '=' #Y
  UNTIL #Y = 6
END-REPEAT
*
END
```

Output of Program RPTEX2S:

```
#X:    1
#X:    2
#X:    3
#X:    4
#X:    5
#X:    6



#Y:    1
#Y:    2
#Y:    3
#Y:    4
#Y:    5
#Y:    6
```

Equivalent reporting-mode example: **RPTEX2R**.

118 REQUEST DOCUMENT

```
REQUEST DOCUMENT  FROM url
[        WITH [with-clause]       ]
[        RETURN [return-clause]   ]
RESPONSE http-response-code
[GIVING natural-error-number]
```

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

Related Statement: `PARSE XML`

Belongs to Function Group: ***Internet and XML***

Function

The `REQUEST DOCUMENT` statement is used to retrieve and upload documents on the internet as described in *REQUEST DOCUMENT* in *Statements for Internet and XML Access* in the *Programming Guide*.

For information on Unicode support, see *REQUEST DOCUMENT* in the *Unicode and Code Page Support* documentation.

See also the description of the Natural profile and session parameter `RQTOUT`, which specifies the timeouts used for HTTP requests issued internally by the `REQUEST DOCUMENT` statement.

Restrictions for Cookies

Under the HTTP Protocol, a server uses cookies to maintain state information on the client workstation.

`REQUEST DOCUMENT` is implemented using internet option settings. This means that, depending on the security settings, cookies will be used.

If the internet option setting `Disabled` is set, no cookies will be sent, even if a cookie header (*`header-name-out/header-value-out`*) is sent.

For server environments, do not use the internet option setting `Prompt`. This setting leads to a "hanging" server, because no client will be able to answer the prompt.

The following profile parameters have to be considered: `NOPROX`, `PROXPORT`, `PROX`, `SSLPRX`, `SSLPRXPT`, `NOSSLPRX`. For information on these profile parameters, refer to the *Parameter Reference*.

For HTTPS, OpenSSL must be installed.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
url	C	S				A												yes	no
http-response-code		S								I4								yes	yes
natural-error-number		S								I4								yes	no

Syntax Element Description:

Syntax Element	Description
url	**Location of Document:** *url* is the URL to access a document.
with-clause	**WITH Clause:** See *with-clause*.
return-clause	**RETURN Clause:** See *return-clause*.
http-response-code	**RESPONSE:** *http-response-code* is the HTTP response status code returned for the request, for example: 200 (request completed). See also ***HTTP Responses Redirected and Denied***. For a list of possible HTTP status codes, refer to the RFC 2616 memorandum published by the World Wide Web Consortium (W3C).
natural-error-number	**GIVING Option:** *natural-error-number* contains the 4-digit Natural error number if the request could not be performed.

with-clause

```
[USER user-id]
[PASSWORD user-password]
[HEADER {[NAME] header-name-out [VALUE] header-value-out} ...]
[DATA {ALL outbound-document [ENCODED [[IN] CODEPAGE code-page-out]]
    |{[NAME] variable-name-out [VALUE] variable-value-out} ...}]
```

The *with-clause* is used to specify optional user/password, header and data details for the request.

An empty *with-clause* (that is, no value specified after WITH) is ignored.

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
user-id	C	S				A													yes	no
user-password	C	S				A													yes	no
header-name-out	C	S				A													yes	no
header-value-out	C	S				A		N	P	I	F		D	T	L				yes	no
outbound-document	C	S				A	U	N	P	I	F	B	D	T	L				yes	no
code-page-out	C	S				A													yes	no
variable-name-out	C	S				A													yes	no
variable-value-out	C	S				A		N	P	I	F		D	T	L				yes	no

Syntax Element Description:

Syntax Element	Description
user-id	**USER:** *user-id* is the ID of the user that will be used for the request.
user-password	**PASSWORD:** *user-password* is the password of the user that will be used for the request.
header-name-out *header-value-out*	**HEADER NAME/VALUE Option:** *header-name-out* and *header-value-out* can only be used in conjunction with each other: ■ *header-name-out* is the name of a header variable sent with this request. ■ *header-value-out* is the value of a header variable sent with this request. *header-name-out*:

Syntax Element	Description
	Header names must not contain a carriage return (CR), a line feed (LF) or a colon (:). This will not be checked by the `REQUEST DOCUMENT` statement. For valid header names, see the HTTP specifications. For compatibility with the web interface, header names can be written with underscore (_) instead of a dash (-). (Internally, the underscore is replaced by a dash). *`header-value-out`*: Header values are not allowed to contain CR/LF. This will not be checked by the `REQUEST DOCUMENT` statement. For valid header values and formats, see the HTTP specifications. See also ***Automatically Generated Headers***.
`outbound-document`	**DATA ALL Option:** *`outbound-document`* is a complete document that is to be sent. This value is needed for the HTTP REQUEST-METHOD `PUT` (see ***Automatically Generated Headers***).
`code-page-out`	**DATA ALL ENCODED Option:** Data transfer with the `REQUEST DOCUMENT` statement normally does not involve any code page conversion. If you want to encode outgoing data in a specific code page, use the `CODEPAGE` option: *`outbound-document`* will be encoded from the default code page (value of the system variable `*CODEPAGE`) to the code page given in *`code-page-out`*. **Encoding and Charset Attributes:** If the outbound document contains an encoding (XML) or a charset (HTML) attribute, we recommend that the value of the `ENCODED` option maps the attribute value of the document. Example: If an outbound XML document contains the attribute encoding=`"UTF-8"`, code the `REQUEST DOCUMENT` statement with the option `DATA ALL #DOCUMENT ENCODED CODEPAGE 'UTF-8'`.
`variable-name-out` *`variable-value-out`*	**DATA NAME/VALUE Option:** *`variable-name-out`* and *`variable-value-out`* request only specific `DATA` variable information instead of the complete document. They can only be used in conjunction with each other: ■ *`variable-name-out`* is the name of a `DATA` variable to be sent with this request. ■ *`variable-value-out`* is the value of a `DATA` variable to be sent with this request. This value is needed for the HTTP `REQUEST-METHOD POST` (URL encoding necessary, especially ampersand (&), equal sign (=), percent sign (%) characters). **Restriction:**

Syntax Element	Description
	If *variable-name-out* and *variable-value-out* are given and the communication is `http://` or `https://`, by default, the `REQUEST-METHOD POST` (see ***Automatically Generated Headers***) with content type `application/x-www-form-urlencoded` is used. During the request, *variable-name-out* and *variable-value-out* will be separated by equal sign (=) and ampersand (&) characters. Therefore, the operands are not allowed to contain equal sign (=), ampersand (&) and, because of URL encoding, percent sign (%) characters. These characters are considered reserved and need to be encoded as indicated in ***Reserved Characters***.

return-clause

[HEADER [ALL *header-all-in*] [[NAME] *header-name-in* [VALUE] *header-value-in* ...]]
[PAGE *inbound-document* [ENCODED [[FOR <u>TYPE</u>S *mime-type* ...] [IN] CODEPAGE *code-page-in*]]]

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
header-all-in		S				A													yes	yes
header-name-in	C	S				A													yes	no
header-value-in		S	A *			A		N	P	I	F	B	D	T	L				yes	yes
inbound-document		S				A	U					B							yes	yes
mime-type	C	S				A													yes	no
code-page-in	C	S				A													yes	no

This clause can be used to specify return information for the headers and/or the document.

Syntax Element Description:

Syntax Element	Description
header-all-in	**HEADER ALL Option:** *header-all-in* contains all header data delivered with the HTTP response. The first line contains the status information and all following lines contain the headers as pairs of name and value. The names always end in a colon (:) and the values end in a line feed (LF). Internally, all carriage returns/line feeds (CR/LF) are transformed into line feeds (LF).
header-name-in *header-value-in*	**HEADER NAME/VALUE Option:** *header-name-in* and *header-value-in* return only specific header information. They can only be used in conjunction with each other:

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td></td><td>
<code>header-name-in</code> is the name for the header information returned by the HTTP request.

For compatibility with the web interface, header names can be written with underscore (_) instead of dash (-) characters.

Internally, the underscore is replaced by a dash. If <code>header-name-in</code> is a blank string, the status information is returned, for example:

<pre>HTTP/1.0 200 OK</pre>
<code>header-value-in</code> is the scalar or array value required to receive the header data returned by the HTTP request.

An array definition is required if more than one occurrence of the same header is expected, for example, multiple Set-Cookie headers.

Only one dimension of a multi-dimensional array may contain an index range (see Example 9).

An X-array must be materialized before you can use it.

If the number of array occurrences exceeds the number of headers, the unused occurrences are reset. If the number of headers exceeds the number of array occurrences, the remaining header values are ignored.

For an example of an array definition, see Example 9 - RETURN HEADER NAME VALUE with Array Definition.
</td></tr>
<tr><td><code>inbound-document</code></td><td>PAGE Option:

<code>inbound-document</code> is the document returned for this request. No encoding at all of the returned page will be done; that is, the page will remain encoded as delivered from the HTTP server.</td></tr>
<tr><td><code>code-page-in</code></td><td>PAGE ENCODED Option:

Data transfer with the <code>REQUEST DOCUMENT</code> statement normally does not involve any code page conversion. If you want to encode incoming data in a specific code page, use the <code>ENCODED</code> option:

If necessary, <code>inbound-document</code> will be encoded in the default code page (value of system variable <code>*CODEPAGE</code>) of the Natural session.

If the value of <code>code-page-in</code> is blank, no conversion occurs. <code>inbound-document</code> is then encoded in the default code page (profile parameter <code>CP</code> in the Configuration Utility).

Note: “Returned MIME type contains an encoding” means that the HTTP server returns a content-type header with a <code>charset=</code> clause, for example: <code>charset=ISO-8859-1</code>.</td></tr>
</table>

Syntax Element	Description
mime-type	**PAGE ENCODED FOR TYPES Option:** As a response of an HTTP/HTTPS request, incoming data may contain binary data (for example, image/gif) or character data (for example, text/html). Together with the response, the REQUEST DOCUMENT statement receives a parameter which specifies the type of content of the requested document (MIME type, also known as internet media type). This parameter may contain information about the code page in which the document is encoded. *mime-type* is the list of MIME types for which an encoding of the returned document in *inbound-document* will be performed. If the returned MIME type contains an encoding, *inbound-document* will be encoded from this code page to the default code page (A/B) or (U). If the returned MIME type does not contain an encoding, then *inbound-document* will be encoded from the code page defined with *code-page-in* to the default code page (value of the system variable *CODEPAGE) (A/B) or (U). If the returned MIME type does not contain an encoding, then an additional check is performed if the returned MIME type matches one of the types given with *mime-type*. If a match is found, *inbound-document* will be encoded from the code page defined with *code-page-in* to the default code page (A/B) or (U).

Automatically Generated Headers

For an HTTP request, some headers are required, for example: REQUEST-METHOD or content type. These headers will be automatically generated depending on the parameters given with the REQUEST DOCUMENT statement.

Note: It is possible to overwrite the automatically generated headers. Natural will not check them for errors. Unexpected errors may occur.

- HTTP REQUEST-METHOD
- Content Type

HTTP REQUEST-METHOD

The REQUEST DOCUMENT statement supports the following HTTP REQUEST-METHODs: HEAD, POST, GET and PUT.

The following table shows the HTTP REQUEST-METHOD generated depending on the given operands:

	WITH HEADER	WITH DATA	RETURN HEADER	RETURN PAGE
HEAD	o	-	x	-
POST	o	x	o	x
GET	o	-	o	x
PUT	o	DATA ALL *	o	o

In addition to the standard `REQUEST-METHOD`s mentioned above, the methods `DELETE`, `PATCH`, `OPTIONS` and `TRACE` can be specified in a `REQUEST-METHOD` header.

Explanation:

o Optional. Operand can be optionally specified.

- Operand cannot be specified.

x Operand is always specified.

* Only applies to `DATA ALL` and not `DATA NAME VALUE`.

Content Type

The `REQUEST-METHOD POST` requires a content-type header for the HTTP request. If no content type is explicitly specified, Natural inserts the following default content-type header into the request:

```
application/x-www-form-urlencoded
```

URL Encoding for Special Characters

When sending `POST` data with the content type `application/x-www-form-urlencoded`, certain characters must be represented by means of URL encoding, which means substituting the character with *`%hexadecimal-character-code`*. Some basic details are given here:

- Non-ASCII Characters
- Unsafe Characters
- Reserved Characters

For full details of when and why URL encoding is necessary, refer to the memorandums RFC 1630, RFC 1738 and RFC 1808 published by the World Wide Web Consortium (W3C).

Non-ASCII Characters

All non-ASCII characters (that is, valid ISO 8859/1 characters that are not also ASCII characters) must be URL encoded, for example, the file `köln.html` would appear in an URL as `k%F6ln.html`.

Unsafe Characters

URL encode the following unsafe characters when you request web pages to avoid server failures:

<table>
<tr><th>Unsafe Character</th><th>URL Encoding</th></tr>
<tr><td>the tab character</td><td>%09</td></tr>
<tr><td>the space character</td><td>%20</td></tr>
<tr><td>[</td><td>%5B</td></tr>
<tr><td>\</td><td>%5C</td></tr>
<tr><td>]</td><td>%5D</td></tr>
<tr><td>^</td><td>%5E</td></tr>
<tr><td>`</td><td>%60</td></tr>
<tr><td>{</td><td>%7B</td></tr>
<tr><td>|</td><td>%7C</td></tr>
<tr><td>}</td><td>%7D</td></tr>
<tr><td>~</td><td>%7E</td></tr>
</table>

Reserved Characters

Some characters have special meanings in URLs, such as the colon (:) that separates the URL scheme from the rest of the URL, the double slash (//) that indicates that the URL conforms to the Common Internet Scheme syntax and the percent sign (%). Generally, when these characters appear as parts of file names, they must be URL encoded to distinguish them from their special meaning in URLs (this is a simplification, refer to the RFCs mentioned earlier for full details).

Reserved characters are:

Reserved Character	URL Encoding
"	%22
#	%23
%	%25
&	%26
+	%2B
,	%2C
/	%2F

Reserved Character	URL Encoding
:	%3A
<	%3C
=	%3D
>	%3E
?	%3F
@	%40

HTTP Responses Redirected and Denied

For a list of HTTP status codes, refer to the RFC 2616 memorandum published by the World Wide Web Consortium (W3C).

The following special considerations apply to the HTTP responses for redirected and denied requests:

- Response 301 - 303 (Redirected)
- Response 401 (Denied/Unauthorized)

Response 301 - 303 (Redirected)

The HTTP response codes 301, 302 and 303 mean that the URL where the requested document resides has changed and that the request was therefore redirected to another URL. As a response, the return header with the name `LOCATION` will be displayed. This header contains the URL where the requested page has moved to. A new `REQUEST DOCUMENT` request can be used to retrieve the page moved.

HTTP browsers redirect automatically to the new URL, but the `REQUEST DOCUMENT` statement does not handle redirection automatically.

Response 401 (Denied/Unauthorized)

The HTTP response code 401 means that the requested page can only be accessed if a valid user ID and password are provided with the request. As a response, the return header with the name `WWW-AUTHENTICATE` will be delivered with the REALM needed for this request.

HTTP browsers normally display a dialog with user ID and password, but with the `REQUEST DOCUMENT` statement, no dialog is displayed.

Examples

Note: There is an example dialog `V5-RDOC` for this statement in the example library `SYSEXV`.

Example 1 - General Request

```
REQUEST DOCUMENT FROM "http://bolsap1:5555/invoke/sap.demo/handle_RFC_XML_POST"
  WITH
    USER #User PASSWORD #Password
    DATA
    NAME 'XMLData'       VALUE #Queryxml
    NAME 'repServerName' VALUE 'NT2'
  RETURN
    PAGE #Resultxml
RESPONSE #rc
```

Example 2 - Simple GET Request (no data)

```
REQUEST DOCUMENT FROM "http://pcnatweb:8080"
  RETURN
    PAGE #Resultxml
RESPONSE #rc
```

Example 3 - Simple HEAD Request (no return page)

```
REQUEST DOCUMENT FROM "http://pcnatweb"
RESPONSE #rc
```

Example 4 - Simple POST Request (default REQUEST-METHOD)

```
REQUEST DOCUMENT FROM "http://pcnatweb/cgi-bin/nwwcgi.exe/sysweb/nat-env"
  WITH
    DATA
    NAME 'XMLData'       VALUE #Queryxml
    NAME 'repServerName' VALUE 'NT2'
  RETURN
    PAGE #Resultxml
RESPONSE #rc
```

Example 5 - Simple PUT Request (with DATA ALL)

```
REQUEST DOCUMENT FROM "http://pcnatweb/test.txt"
  WITH
    DATA ALL     #document
  RETURN
    PAGE #Resultxml
RESPONSE #rc
```

Example 9 - RETURN HEADER NAME VALUE with Array Definition

```
DEFINE DATA
LOCAL
1 #FROM      (A) DYNAMIC
1 #HEADER    (A) DYNAMIC
1 #PAGE      (A) DYNAMIC
1 #COOKIES (A20/1:3,1:4,2:5)
1 #RC      (I4)
END-DEFINE
ASSIGN #FROM = 'http://www.myserver.com'
REQUEST DOCUMENT FROM #FROM
   RETURN
        HEADER NAME 'Set-Cookie' VALUE #COOKIES(1,2:3,3)
        PAGE   #PAGE
        RESPONSE #RC
PRINT #COOKIES(*,*,*)
END
```

In the example program above, *invalid* array definitions (with multiple dimensions) would be:

```
RETURN HEADER NAME 'Set-Cookie' VALUE #COOKIES(1:3,2:3,3)
RETURN HEADER NAME 'Set-Cookie' VALUE #COOKIES(*,2,*)
```

119 RESET

```
RESET [INITIAL] operand1 ...
```

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

Related Statements: ADD | COMPRESS | COMPUTE | DIVIDE | EXAMINE | MOVE | MOVE ALL | MULTIPLY | SEPARATE | SUBTRACT

Belongs to Function Group: ***Arithmetic and Data Movement Operations***

Function

The RESET statement is used to reset the value of a field:

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

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.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1		S	A	G	M	A	U	N	P	I	F	B	D	T	L	C	G	O	yes	yes

Syntax Element Description:

Syntax Element	Description
`RESET` *operand1*	**Reset to Null Value:** `RESET` (without `INITIAL`) sets the content of each specified field (*operand1*) to its **default initial value**. If *operand1* is a dynamic variable, it will be reset to a null value with the length the variable currently has at the time the `RESET` statement is executed. The current length of a dynamic variable can be ascertained by using the system variable `*LENGTH`. For general information on dynamic variables, see the section *Using Dynamic and Large Variables*.
`RESET INITIAL` *operand1*	**Reset to Initial Value:** `RESET INITIAL` sets each specified field (*operand1*) to the initial value as defined for the field in the `DEFINE DATA` statement. ■ If you specify no `INIT` value in the `DEFINE DATA` statement, a field will be initialized with a **default initial value** depending on its format. ■ If a dynamic variable is used, `*LENGTH` is set to zero if no initial value is defined. ■ If you apply `RESET INITIAL` to an array, it must be applied to the entire array (as defined in the `DEFINE DATA` statement); a `RESET INITIAL` of individual array occurrences is not possible. ■ If an X-array is used, `*OCCURRENCE` is set to zero. ■ `RESET INITIAL` of fields resulting from a redefinition is not possible either. ■ `RESET INITIAL` is applied to a dynamic variable. ■ `RESET INITIAL` cannot be applied to database fields.

Example

```
** Example 'RSTEX1': RESET (with/without INITIAL)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
1 #BINARY  (B4) INIT <1>
1 #INTEGER (I4) INIT <5>
1 #NUMERIC (N2) INIT <25>
END-DEFINE
*
LIMIT 1
READ EMPLOY-VIEW
  /*
  WRITE NOTITLE 'VALUES BEFORE RESET STATEMENT:'
```

```
  WRITE / '=' NAME '=' #BINARY '=' #INTEGER '=' #NUMERIC
  /*
  RESET NAME #BINARY #INTEGER #NUMERIC
  /*
  WRITE /// 'VALUES AFTER  RESET STATEMENT:'
  WRITE / '=' NAME '=' #BINARY '=' #INTEGER '=' #NUMERIC
  /*
  RESET INITIAL #BINARY #INTEGER #NUMERIC
  /*
  WRITE /// 'VALUES AFTER  RESET INITIAL STATEMENT:'
  WRITE / '=' NAME '=' #BINARY '=' #INTEGER '=' #NUMERIC
  /*
END-READ
END
```

Output of Program RSTEX1:

```
VALUES BEFORE RESET STATEMENT:

NAME: ADAM                 #BINARY: 00000001 #INTEGER:           5 #NUMERIC:
 25



VALUES AFTER  RESET STATEMENT:

NAME:                      #BINARY: 00000000 #INTEGER:           0 #NUMERIC:
  0



VALUES AFTER  RESET INITIAL STATEMENT:

NAME:                      #BINARY: 00000001 #INTEGER:           5 #NUMERIC:
 25
```

120 RESIZE

```
RESIZE { dynamic-clause } [GIVING operand5]
       { array-clause   }
```

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

Related Statements: `EXPAND` | `REDUCE`

Belongs to Function Group: ***Memory Management Control for Dynamic Variables or X-Arrays***.

Function

The `RESIZE` statement is used to adjust:

- the size of a dynamic variable (*`dynamic-clause`*), or
- the number of occurrences of X-arrays (*`array-clause`*).

For further information, see also the following sections in the *Programming Guide*:

- *Using Dynamic Variables*
- *Allocating/Freeing Memory Space for a Dynamic Variable*
- *X-Arrays*
- *Storage Management of X-Group Arrays*

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1		S	A			A	U					B							no	no
operand2	C	S								I									no	no
operand3			A	G		A	U	N	P	I	F	B	D	T	L	C	G	O	yes	no
operand4	C	S						N	P	I									no	no
operand5		S								I4									no	yes

Syntax Element Description:

Syntax Element	Description
dynamic-clause	**DYNAMIC Clause:** The `RESIZE DYNAMIC` statement adjusts the allocated length of the currently allocated storage of a dynamic variable (*operand1*) to the value specified with *operand2*. For more information, see ***Dynamic Clause*** below.
operand1	**Dynamic Variable to be Adjusted:** *operand1* is the dynamic variable for which the length is to be adjusted.
operand2	**New Length Specification:** *operand2* is used to specify the new length of the dynamic variable. The value specified must be a non-negative numeric integer constant or a variable of type Integer 4 (I4).
array-clause	**ARRAY Clause:** The `RESIZE ARRAY` statement adjusts the number of occurrences of the X-array (*operand3*) to the upper and lower bound specified with `(dim[,dim[,dim]])`. For more information, see ***Array Clause*** below.
operand3	**Name of X-array:** *operand3* is the X-array. The occurrences of the X-array can be expanded or reduced. The index notation of the array is optional. As index notation only the complete range notation `*` is allowed for each dimension.
`dim` *operand4*	**X-array Lower and Upper Bound:** The lower and upper bound notation (*operand4* or asterisk) to which the X-array should be expanded is specified here. If the current value of the upper or lower bound should be used, an asterisk (*) must be specified in place of *operand4*. For further information, see ***Dimension*** below.
`GIVING` *operand5*	**GIVING Clause:** If the `GIVING` clause is not specified, Natural runtime error processing is triggered if an error occurs. If the `GIVING` clause is specified, *operand5* contains the Natural message number if an error occurred, or zero upon success.

Dynamic Clause

```
[SIZE OF] DYNAMIC [VARIABLE] operand1 TO operand2
```

The `RESIZE DYNAMIC` statement adjusts the allocated length of a dynamic variable (*operand1*) to the value specified with *operand2*.

When the `RESIZE` statement is used, the currently allocated storage size will be adjusted to the requested values, regardless whether it must be increased or decreased.

Array Clause

```
[AND RESET] [OCCURRENCES OF] ARRAY operand3 TO (dim[,dim[,dim]])
```

The `RESIZE ARRAY` statement adjusts the number of occurrences of the X-array (*operand3*) to the upper and lower bound specified with `(dim[,dim[,dim]])`.

The `RESET` option resets all occurrences of the resized X-array to its default zero value. By default (no `RESET` option), the actual values are kept and the resized (new) occurrences are reset.

An upper or lower bound used in an `RESIZE` statement must be exactly the same as the corresponding upper or lower bound defined for the array.

Example:

```
DEFINE DATA LOCAL
1 #a(I4/1:*)
1 #g(1:*)
  2 #ga(I4/1:*)

1 #i(i4)
END-DEFINE
...

*/ resizing #a (1:10)
RESIZE ARRAY #a TO (1:10)       /* #a is resized to
RESIZE ARRAY #a TO (*:10)       /* 10 occurrences.


/* resizing #ga (1:10,1:20)
RESIZE ARRAY #g TO (1:10)       /* 1st dimension is set to (1:10)
RESIZE ARRAY #ga TO (*:*,1:20)  /* 1st dimension is dependent and
                                /* therefore kept with (*:*)
                                /* 2nd dimension is set to (1:20)

RESIZE ARRAY #a TO (5:10)       /* This is rejected because the lower index
                                /* must be 1 or *
RESIZE ARRAY #a TO (#i:10)      /* This is rejected because the lower index
```

```
                                /* must be 1 or *

RESIZE ARRAY #ga TO (1:10,1:20) /* (1:10) for the 1st dimension is rejected
                                /* because the dimension is dependent and
                                /* must be specified with (*:*).
```

For further information, see the following sections in the *Programming Guide*:

- *Storage Management of X-Arrays*
- *Storage Management of X-Group Arrays*

Dimension

Each of the dimensions (`dim`) specified in the ***Array Clause*** is defined using the following syntax:

$$\left\{\begin{array}{l} * \\ \left\{\begin{array}{l} * \\ \mathit{operand4}\end{array}\right\} : \left\{\begin{array}{l} * \\ \mathit{operand4}\end{array}\right\}\end{array}\right\}$$

The lower and upper bound notation (*`operand4`* or asterisk) to which the X-array should be expanded is specified here. If the current value of the upper or lower bound should be used, an asterisk (*) may be specified in place of *`operand4`*. In place of `*:*`, you may also specify a single asterisk.

The number of dimensions (`dim`) must exactly match the defined number of dimensions of the X-array (1, 2 or 3).

121 ROLLBACK (SQL)

```
ROLLBACK
```

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

Belongs to Function Group: ***Database Access and Update***

Function

The SQL statement `ROLLBACK` corresponds to the Natural statement `BACKOUT TRANSACTION`. It undoes all database modifications made since the beginning of the last recovery unit. A recovery unit may start either after the beginning of a session or after the last `SYNCPOINT`, `COMMIT`, `END TRANSACTION` or `BACKOUT TRANSACTION` statement. This statement also releases all records held during the transaction.

If a program tries to backout updates which have already been committed by a terminal I/O, a corresponding Natural error message (NAT3711) is returned.

Caution: As all cursors are closed when a logical unit of work ends, a `ROLLBACK` statement must not be placed within a database modification loop; instead, it has to be placed outside such a loop or after the outermost loop of nested loops.

Consideration for Non-Natural Programs

If an external program written in another standard programming language is called from a Natural program, this external program should not contain its own `ROLLBACK` statement if the Natural program issues database calls, too. The calling Natural program should issue the `ROLLBACK` statement on behalf of the external program.

Example

```
...
DELETE FROM SQL-PERSONNEL WHERE NAME = 'SMITH'
ROLLBACK
...
```

122 RETRY

```
RETRY
```

Related Statements: ACCEPT/REJECT | AT BREAK | AT START OF DATA | AT END OF DATA | BACKOUT TRANSACTION | BEFORE BREAK PROCESSING | DELETE | END TRANSACTION | FIND | GET | GET SAME | GET TRANSACTION DATA | HISTOGRAM | LIMIT | PASSW | PERFORM BREAK PROCESSING | READ | STORE | UPDATE

Belongs to Function Group: ***Database Access and Update***

Function

The RETRY statement is used within an ON ERROR statement block (see ON ERROR statement). It is used to reattempt to obtain a record which is in hold status for another user.

When a record to be held is already in hold status for another user, Natural issues Error Message 3145. See also the session parameter WH (Wait for Record in Hold Status).

The RETRY statement must be placed in the object that causes the Error 3145.

For details on record hold logic, see the section *Record Hold Logic* in the *Programming Guide*.

Restriction

This statement can only be used to access Adabas databases.

Example

```
** Example 'RTYEX1': RETRY
**
** CAUTION: Executing this example will modify the database records!
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
*
1 #RETRY (A1) INIT <' '>
END-DEFINE
*
FIND EMPLOY-VIEW WITH NAME = 'ALDEN'
  /*
  DELETE
  END TRANSACTION
```

```
  /*
  ON ERROR
    IF *ERROR-NR = 3145
      INPUT NO ERASE 10/1
            'RECORD IS IN HOLD' /
            'DO YOU WISH TO RETRY?' /
            #RETRY '(Y)ES OR (N)O?'
      IF #RETRY = 'Y'
        RETRY
      ELSE
        STOP
      END-IF
    END-IF
  END-ERROR
  /*
  AT END OF DATA
    WRITE NOTITLE *NUMBER 'RECORDS DELETED'
  END-ENDDATA
END-FIND
*
END
```

123 RUN

```
RUN [REPEAT] operand1 [operand2 [(parameter)]] ... 40
```

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

Belongs to Function Group: ***Invoking Programs and Routines***

Function

The RUN statement is used to read a Natural source program from the Natural system file and then execute it.

For Natural RPC: See *Notes on Natural Statements on the Server* in the *Natural RPC (Remote Procedure Call)* documentation.

Syntax Description

Operand Definition Table:

Operand	Possible Structure				Possible Formats	Referencing Permitted	Dynamic Definition
operand1	C	S			A	yes	no
operand2	C	S	A	G	A U N P I F B D T L G	yes	no

Syntax Element Description:

Syntax Element	Description
REPEAT	**REPEAT Option:** RUN REPEAT causes the program not to prompt the user for input until the program has finished executing even if multiple output screens (produced by INPUT statements) are produced. This feature may be used if the program is to display multiple screens of information without having the user respond to each screen.
operand1	**Program Name:** As *operand1* the name of the program can be specified as an alphanumeric constant or as the content of an alphanumeric variable. If a variable is used, it must be 8 characters in length. The program may be stored in the current library or in a concatenated library (default steplib is SYSTEM). If the program is not found, an error message is issued.

Syntax Element	Description
	The program is read into the source program work area and overlays any current source program.
operand2	**Parameters:** The RUN statement may also be used to pass parameters to the program to be run. A parameter may be defined with any format. The parameters are converted to a format suitable for a corresponding INPUT field. All parameters are placed on the top of the Natural stack. The parameters can be read using an INPUT statement. The first INPUT statement issued will result in the insertion of all parameters into the fields specified in the INPUT statement. The INPUT statement must have the sign specification (session parameter SG=ON) for parameter fields defined with numeric format. If more parameters are passed than are read by the next INPUT statement, the extra parameters are ignored. The number of parameters may be obtained with the system variable *DATA. **Note:** If *operand2* is a time variable (format T), only the time component of the variable content is passed, but not the date component.
parameter	**Date Format:** If *operand2* is a date variable, you can specify the session parameter DF (described in the *Parameter Reference*) as *parameter* for this variable.

Dynamic Source Text Creation/Execution

The RUN statement may be used to dynamically compile and execute a program for which the source or parts thereof are created dynamically.

Dynamic source text creation is performed by placing source text into global variables and then referring to these variables by using an ampersand (&) instead of a plus sign (+) as the first character of the variable name in the source text. The content of the global variable will be interpreted as source text when the program is invoked using the RUN statement.

A global variable with index must not be used within a program that is invoked via a RUN statement.

It is not allowed to place a comment or an INCLUDE statement in a global variable.

Example

Program containing RUN statement:

```
** Example 'RUNEX1': RUN (with dynamic source program creation)
************************************************************************
DEFINE DATA
GLOBAL
  USING RUNEXGDA
LOCAL
1 #NAME (A20)
1 #CITY (A20)
END-DEFINE
*
INPUT 'Please specify the search values:' //
      'Name:' #NAME /
      'City:' #CITY
*
RESET +CRITERIA     /* defined in GDA 'RUNEXGDA'
*
IF #NAME = ' ' AND #CITY = ' '
  REINPUT 'Enter at least 1 value'
END-IF
*
IF #NAME NE ' '
  COMPRESS 'NAME' ' =''' #NAME '''' INTO +CRITERIA LEAVING NO
END-IF
IF #CITY NE ' '
  IF +CRITERIA NE ' '
    COMPRESS +CRITERIA 'AND' INTO +CRITERIA
  END-IF
  COMPRESS +CRITERIA ' CITY =''' #CITY '''' INTO +CRITERIA LEAVING NO
END-IF
*
RUN 'RUNEXFND'
*
END
```

Program RUNEXFND executed by RUN statement:

```
** Example 'RUNEXFND': RUN (program executed with RUN in RUNEX1)
************************************************************************
DEFINE DATA
GLOBAL
  USING RUNEXGDA
LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
```

```
END-DEFINE
*
* &CRITERIA filled with  "NAME = 'xxxxx' AND CITY = 'xxxx'"
*
FIND NUMBER EMPLOY-VIEW WITH &CRITERIA
     RETAIN AS 'EMP-SET'
DISPLAY *NUMBER
*
END
```

Global Data Area RUNEXGDA:

```
Global    RUNEXGDA  Library SYSEXSYN                      DBID    10 FNR    32
Command                                                                     > +
I T L  Name                             F Length     Miscellaneous
All -- -------------------------------- - ---------- ------------------------->
     1 +CRITERIA                        A         80
```

XIV

124 SELECT (SQL)

For explanations of the symbols used in the syntax diagrams, see *Syntax Symbols*.

Belongs to Function Group: *Database Access and Update*

Function

The `SELECT` statement supports both the **cursor-oriented selection** that is used to retrieve an arbitrary number of rows and the **non-cursor selection** (singleton `SELECT`) that retrieves at most one single row. With the `SELECT ... END-SELECT` construction, Natural uses the same database loop processing as with the `FIND` statement.

Two different structures are possible.

Syntax 1 - Cursor-Oriented Selection

Like the Natural `FIND` statement, the cursor-oriented `SELECT` statement is used to select a set of rows (records) from one or more Db2 tables, based on a search criterion. Since a database loop is initiated, the loop must be closed by a `LOOP` (reporting mode) or `END-SELECT` statement. With this construction, Natural uses the same loop processing as with the `FIND` statement.

In addition, no cursor management is required from the application program; it is automatically handled by Natural.

- Syntax 1 - Common Set
- Syntax 1 - Extended Set

Syntax 1 - Common Set

```
SELECT selection into-clause table-expression

[ { UNION     } [ ALL ] [ (SELECT selection        ] ]
[ { EXCEPT    }         [ table-expression)        ] ]
[ { INTERSECT }         [ SELECT selection         ] ]
                          table-expression             ...
[ORDER BY criteria]
statement ...
{ END-SELECT }
{ LOOP       }
```

Syntax 1 - Extended Set

```
SELECT selection into-clause table-expression

[ { UNION     } [ ALL ] [ (SELECT selection          ] ]
[ { EXCEPT    }         [ table-expression)          ] ]
[ { INTERSECT }         [ SELECT selection           ] ]
                        [ table-expression           ] ] ...

[ORDER BY criteria]
[OPTIMIZE FOR integer ROWS]
[WITH isolation-level]
[FETCH FIRST row-limit]
[WITH scroll-mode]
[IF NO RECORDS FOUND instruction]
statement ...
{ END-SELECT }
{ LOOP       }
```

Syntax 2 - Non-Cursor Selection

The `SELECT SINGLE` statement supports the functionality of a non-cursor selection (singleton `SELECT`); that is, a **select expression** that retrieves at most one row without using a cursor. It cannot be referenced by a **positioned** `UPDATE` or a **positioned** `DELETE` statement.

- Syntax 2 - Common Set
- Syntax 2 - Extended Set

Syntax 2 - Common Set

```
SELECT SINGLE
selection into-clause table-expression
[IF NO RECORDS FOUND instruction]
statement ...
{ END-SELECT }
{ LOOP       }
```

Syntax 2 - Extended Set

```
SELECT SINGLE
selection into-clause table-expression
[WITH isolation-level]
[FETCH FIRST row-limit]
[IF NO RECORDS FOUND instruction]
statement ...
{ END-SELECT }
{ LOOP       }
```

Syntax Element Description

This section alphabetically lists and explains the syntax items contained in the syntax diagrams of *Syntax 1 - Cursor-Oriented Selection* and *Syntax 2 - Non-Cursor Selection*:

- END-SELECT | LOOP
- FETCH FIRST row-limit
- IF NO RECORDS FOUND instruction
- into-clause
- OPTIMIZE FOR integer ROWS
- ORDER BY criteria
- selection
- statement
- table-expression
- UNION | EXCEPT | INTERSECT Clause
- WITH isolation-level
- WITH scroll-mode

END-SELECT | LOOP

In structured mode, the Natural reserved keyword `END-SELECT` must be used to end the `SELECT` statement.

In reporting mode, the `LOOP` statement must be used to end the `SELECT` statement.

FETCH FIRST row-limit

```
FETCH FIRST [ 1       ] { ROW  } ONLY
            [ integer ] { ROWS }
```

The FETCH FIRST clause limits the number of rows to be fetched. It improves the performance of queries with potentially large result sets if only a limited number of rows is needed.

This clause is only valid against Db2 databases. When used against other databases, it will cause runtime errors.

IF NO RECORDS FOUND instruction

Note: This clause actually does not belong to Natural SQL; it represents Natural functionality which has been made available to SQL loop processing.

Structured Mode Syntax

```
IF NO [RECORDS] [FOUND]
{ ENTER         }
{ statement ... }
END-NOREC
```

Reporting Mode Syntax

```
IF NO [RECORDS] [FOUND]
{ ENTER                     }
{ statement                 }
{ DO statement ... DOEND    }
```

The IF NO RECORDS FOUND clause is used to initiate a processing loop if no records meet the selection criteria specified in the preceding SELECT statement.

If no records meet the specified selection criteria, the IF NO RECORDS FOUND clause causes the processing loop to be executed once with an "empty" record. If this is not desired, specify the statement ESCAPE BOTTOM within the IF NO RECORDS FOUND clause.

If one or more statements are specified with the IF NO RECORDS FOUND clause, the statements are executed immediately before the processing loop is entered. If no statements are to be executed before entering the loop, the keyword ENTER must be used.

Note: If the result set of the SELECT statement consists of a single row of NULL values, the IF NO RECORDS FOUND clause is not executed. This could occur if the selection list consists solely of one of the aggregate functions SUM, AVG, MIN or MAX on columns, and the set on which these aggregate functions operate is empty. When you use these aggregate functions in the above-mentioned way, you should therefore check the values of the corresponding null-indicator fields instead of using an `IF NO RECORDS FOUND` clause.

Database Values
Unless other value assignments are made in the statements accompanying an `IF NO RECORDS FOUND` clause, Natural resets to empty all database fields which reference the file specified in the current loop.

Evaluation of System Functions
Natural system functions are evaluated once for the empty record that is created for processing as a result of the `IF NO RECORDS FOUND` clause.

into-clause

```
INTO { parameter,...
     { VIEW {view-name [correlation-name ]},... }
```

The `INTO` keyword introduces an `INTO` clause. This clause is used to specify the target fields in the program which are to be filled with the result of the selection.

The `INTO` clause can specify either single *parameters* or one or more views as defined in the `DEFINE DATA` statement.

All target field values can come either from a single table or from more than one table as a result of a join operation (see also ***Join Queries***).

Note: In standard SQL syntax, an `INTO` clause is only used in non-cursor select operations (singleton `SELECT`) and can be specified only if a single row is to be selected. In Natural, however, the `INTO` clause is used for both cursor-oriented and non-cursor select operations.

The *selection* can also merely consist of an asterisk (*). In a standard **select expression**, this is a shorthand for a list of all column names in the table(s) specified in the `FROM` clause. In the Natural `SELECT` statement, however, the same syntactical item `SELECT *` has a different semantic meaning: all the items listed in the `INTO` clause are also used in the selection. Their names must correspond to names of existing database columns.

Syntax Element Description:

Syntax Element	Description
parameter	If single parameters are specified as target fields, their number and formats must correspond to the number and formats of the *columns* and/or *scalar-expressions* specified in the corresponding selection as described above (for details, see ***Scalar Expressions***). See ***Example 5***.
view-name	The name a Natural view as defined in the `DEFINE DATA` statement. If one or more views are referenced in the `INTO` clause, the number of items specified in the *selection* must correspond to the number of fields defined in the view(s) (not counting group fields, redefining fields and indicator fields). **Note:** Both the Natural target fields and the table columns must be defined in a Natural DDM. Their names, however, can be different, since assignment is made according to their sequence. See ***Example 5***.
correlation-name	If the `VIEW` clause is used within a `SELECT *` construction where multiple tables are to be joined, *correlation-names* are required if the specified view contains fields that reference columns which exist in more than one of these tables. In order to know which column to select, all these columns are qualified by the specified *correlation-name* at generation of the selection list. The *correlation-name* assigned to a view must correspond to one of the *correlation-names* used to qualify the tables to be joined. See also the section ***Join Queries*** and ***Example 6***.

Examples

Example 1:

```
DEFINE DATA LOCAL
01 PERS VIEW OF SQL-PERSONNEL
  02 NAME
  02 AGE
END-DEFINE
...
SELECT *
  INTO NAME, AGE
```

Example 2:

```
...
SELECT *
  INTO VIEW PERS
```

These examples are equivalent to the following ones:

Example 3:

```
...
SELECT NAME, AGE
  INTO NAME, AGE
```

Example 4:

```
...
SELECT NAME, AGE
  INTO VIEW PERS
```

Example 5:

```
DEFINE DATA LOCAL
01 PERS VIEW OF SQL-PERSONNEL
  02 NAME
  02 AGE
END-DEFINE
...
SELECT FIRSTNAME, AGE
  INTO VIEW PERS
  FROM SQL-PERSONNEL
...
```

The target fields `NAME` and `AGE`, which are part of a Natural view, receive the contents of the table columns `FIRSTNAME` and `AGE`.

Example 6:

```
DEFINE DATA LOCAL
01 PERS VIEW OF SQL-PERSONNEL
  02 NAME
  02 FIRST-NAME
  02 AGE
END-DEFINE
...
SELECT *
  INTO VIEW PERS A
  FROM SQL-PERSONNEL A, SQL-PERSONNEL B
...
```

OPTIMIZE FOR integer ROWS

```
OPTIMIZE FOR integer ROWS
```

This clause is only valid against Db2 databases. When used against other databases, it will cause runtime errors.

The `OPTIMIZE FOR integer ROWS` clause is used to inform Db2 in advance of the number (*integer*) of rows to be retrieved from the result table. Without this clause, Db2 assumes that all rows of the result table are to be retrieved and optimizes accordingly.

This optional clause is useful if you know how many rows are likely to be selected, because optimizing for *integer* rows can improve performance if the number of rows actually selected does not exceed the *integer* value (which can be in the range from `0` to `2147483647`).

Example

```
SELECT name INTO
#name FROM table WHERE AGE = 2 OPTIMIZE FOR 100 ROWS
```

ORDER BY criteria

$$\texttt{ORDER BY} \left\{ \left\{ \begin{matrix} \textit{column-reference} \\ \textit{integer} \end{matrix} \right\} \left[\begin{matrix} \texttt{ASC} \\ \texttt{DESC} \end{matrix} \right] \right\}$$

The `ORDER BY` clause arranges the result of a `SELECT` statement in a particular sequence.

Syntax Element Description:

Syntax Element	Description
column-reference	Each `ORDER BY` clause must specify a column of the result table. In most `ORDER BY` clauses a column can be identified either by *column-reference* (that is, by an optionally qualified column name) or by column number. In a query involving `UNION`, a column must be identified by column number. See also ***Column Reference***.
integer	In a query involving `UNION`, a column must be identified by column number. The column number is the ordinal left-to-right position of a column within the selection, which means it is an integer value. This feature makes it possible to order a result on the basis of a computed column which does not have a name.
`ASC|DESC`	Specifies the sort order: ascending (`ASC`) or descending (`DESC`). `ASC` is the default. See ***Example 2***.

Examples

Example 1:

```
DEFINE DATA LOCAL
1 #NAME          (A20)
1 #YEARS-TO-WORK (I2)
END-DEFINE
...
SELECT NAME , 65 - AGE
  INTO #NAME, #YEARS-TO-WORK
  FROM SQL-PERSONNEL
  ORDER BY 2
  ...
```

Example 2:

```
DEFINE DATA LOCAL
1 PERS VIEW OF SQL-PERSONNEL
1 NAME
1 AGE
1 ADDRESS   (1:6)
END-DEFINE
...
SELECT NAME, AGE, ADDRESS
  INTO VIEW PERS
  FROM SQL-PERSONNEL
  WHERE AGE = 55
  ORDER BY NAME DESC
  ...
```

selection

See ***Selection*** in *Select Expressions*.

statement

The Natural statement(s) to be executed in the processing loop.

table-expression

See `table-expression` in *Select Expressions*.

UNION | EXCEPT | INTERSECT Clause

```
{ UNION     }  [ DISTINCT ]  [ (SELECT selection table-expression) ]
{ EXCEPT    }  [ ALL      ]  [ SELECT selection table-expression   ] ...
{ INTERSECT }
```

`UNION`, `EXCEPT` and `INTERSECT` introduce a query that involves set operations.

Set operations combine the results of two or more *`select-expressions`*. The columns specified in the individual *`select-expressions`* must match in number, type and format.

The `INTO` clause must be specified with the first *`select-expression`* only.

Syntax Element Description:

Syntax Element	Description
`UNION`	Combines the results of two or more *`select-expressions`*.
`EXCEPT`	Specifies the difference set of the result sets of two *`select-expressions`*.
`INTERSECT`	Specifies the intersection of two result sets.
`DISTINCT`	Specifies that the result set does not contain redundant (duplicate) rows. `DISTINCT` is the default setting.
`ALL`	Specifies that the result set contains redundant (duplicate) rows. Redundant duplicate rows are eliminated from the result of a set operation unless the set operation explicitly includes the `ALL` qualifier.

Example

```
DEFINE DATA LOCAL
01 PERS VIEW OF SQL-PERSONNEL
  02 NAME
  02 AGE
  02 ADDRESS (1:6)
END-DEFINE
...
SELECT NAME, AGE, ADDRESS
  INTO VIEW PERS
  FROM SQL-PERSONNEL
  WHERE AGE > 55
UNION ALL
SELECT NAME, AGE, ADDRESS
  FROM SQL-EMPLOYEES
  WHERE PERSNR < 100
ORDER BY NAME
...
```

```
END-SELECT
...
```

WITH isolation-level

```
     ⎧ CS                     ⎫
     ⎪ RR                     ⎪
WITH ⎨ RR KEEP UPDATE LOCK    ⎬
     ⎪ RS                     ⎪
     ⎪ RS KEEP UPDATE LOCKS   ⎪
     ⎩ UR                     ⎭
```

This clause allows you to specify an explicit isolation level with which the statement is to be executed.

This clause is only valid against Db2 databases. When used against other databases, it will cause runtime errors.

The following options are provided:

Option	Meaning
CS	Cursor Stability
RR	Repeatable Read
RR KEEP UPDATE LOCKS	Only applies to ***Syntax 1 - Extended Set*** and only if a **positioned** UPDATE or a **positioned** DELETE statement is processed with the SELECT statement. Repeatable Read and retaining update locks.
RS	Read Stability
RS KEEP UPDATE LOCKS	Only applies to ***Syntax 1 - Extended Set*** and only if a **positioned** UPDATE or a **positioned** DELETE statement is processed with the SELECT statement. Read Stability and retaining update locks.
UR	Uncommitted Read UR can only be specified within a SELECT statement and when the table is read-only. The default isolation level is determined by the isolation of the package or plan into which the statement is bound. The default isolation level also depends on whether the result table is read-only or not. To find out the default isolation level, refer to the IBM literature.

WITH scroll-mode

```
WITH { ASENSITIVE SCROLL          } [:] scroll_hv [GIVING [:] sqlcode]
     { INSENSITIVE SCROLL         }
     { SENSITIVE STATIC SCROLL    }
     { SENSITIVE DYNAMIC SCROLL   }
```

Natural supports SQL scrollable cursors by using the clauses `WITH ASENSITIVE SCROLL`, `WITH SENSITIVE STATIC SCROLL`, and `SENSITIVE DYNAMIC SCROLL`. Scrollable cursors allow Natural applications to position randomly any row in a result set. With non-scrollable cursors, the data can only be read sequentially, from top to bottom.

RDBMS scrollable cursors are enabled with this clause. Scrollable cursors can be `ASENSITIVE`, `INSENSITIVE`, `SENSITIVE STATIC`, or `SENSITIVE DYNAMIC`.

Scrollable cursors allow the application to position any row in the cursor at any time as long as the cursor is open. Scrollable cursors are not supported for Sybase databases at all. Scrollable cursors are not supported for the MS SQL Server DBLIB interface, but only for the MS SQL Server ODBC interface.

The positioning is performed depending on the content of the *scroll_hv*. The content is evaluated each time a `FETCH` against the database is executed.

Note: Not all SQL database systems support all options.

Syntax Element Description:

Syntax Element	Description
`ASENSITIVE SCROLL`	Specifies that the cursor is either `INSENSITIVE` or `SENSITIVE DYNAMIC`. This is determined by the database at open time of the cursor, depending on the read-only property of the cursor: If the cursor is read-only, the cursor will become `INSENSITIVE`. If the cursor is not read-only, the cursor will become `SENSITIVE DYNAMIC`. This is supported for Db2 databases.
`INSENSITIVE SCROLL`	Specifies that the cursor is insensitive for updates, deletes and inserts executed against the base table, after the cursor has been updated.`INSENSITIVE SCROLL` refers to a cursor that cannot be used in **Positioned** `UPDATE` or **Positioned** `DELETE` operations. This is supported for Oracle, Adabas D, MS SQL Server ODBC, MySQL, MariaDB, PostgreSQL, and Db2 databases. In addition, once opened, an `INSENSITIVE SCROLL` cursor does not reflect `UPDATE`, `DELETE` or `INSERT` operations against the base table after the cursor was opened. See also **Note**.
`SENSITIVE STATIC SCROLL`	Specifies that the cursor is sensitive for updates and deletes against the base table, but not against inserts, after the cursor has been opened.`SENSITIVE STATIC SCROLL`

Syntax Element	Description
	refers to a cursor that can be used for Positioned `UPDATE` or Positioned `DELETE` operations. This is supported for Adabas D, MS SQL Server ODBC and Db2 databases.In addition, a `SENSITIVE STATIC SCROLL` cursor reflects `UPDATE` and `DELETE` operations of base table rows. The cursor does not reflect `INSERT` operations. See also **Note**.
`SENSITIVE DYNAMIC SCROLL`	`SENSITIVE DYNAMIC` specifies that the cursor is sensitive for updates, deletes and inserts against the base table, after the cursor has been opened. `SENSITIVE DYNAMIC` scrollable cursors reflect `UPDATE` and `DELETE` operations against the base table while the cursor is open. This is supported for Adabas D, MS SQL Server ODBC and Db2 databases.

Note: `INSENSITIVE` and `SENSITIVE STATIC` scrollable cursors use temporary result tables and require a `TEMP` database in Db2 (see the relevant Db2 literature by IBM).

scroll_hv

The variable *scroll_hv* must be alphanumeric.

The variable *scroll_hv* specifies which row of the result table will be fetched during one execution of the database processing loop. The content of *scroll_hv* is evaluated each time the database processing loop cycle is executed.

```
                    { CURRENT                                  }
                    { FIRST                                    }
                    { LAST                                     }
                    { PRIOR                                    }
[ INSENSITIVE ]     { NEXT                                     }
[ SENSITVE    ]     {                                          }
                    { { ABSOLUTE } [ + ]                       }
                    { { RELATIVE } [ - ]  integer              }
```

scroll_hv Options

Option	Explanation
`CURRENT`	Fetches the current row (again).
`FIRST`	Fetches the first row.
`LAST`	Fetches the last row.
`NEXT`	Fetches the row after the current one. This is the default value.
`PRIOR`	Fetch the row before the current one.
`+|-` *integer*	Only applies in connection with `ABSOLUTE` or `RELATIVE`. Specifies the position of the row to be fetched `ABSOLUTE` or `RELATIVE`.

Option	Explanation
	Enter a plus (+) or minus (-) sign followed by an integer. The default value is a plus (+).
`ABSOLUTE`	Only applies in connection with `+`|`-` *`integer`*. Uses *`integer`* as the absolute position within the result set from where the row is fetched.
`RELATIVE`	Only applies in connection with `+`|`-` *`integer`*. Uses *`integer`* as the relative position to the current position within the result set from where the row is fetched.

There are some restrictions for special RDBMS systems:

- Db2 does not support the keyword `CURRENT`.
- In a `SELECT FOR UPDATE` loop Db2 only supports `NEXT` as scrolling option.
- MS SQL Server (ODBC interface) does not support the keyword `CURRENT`.
- Adabas D does not support `RELATIVE` scrolling.

GIVING [:] *`sqlcode`*

The specification of `GIVING [:]` *`sqlcode`* is optional. If specified, the Natural variable `[:]` *`sqlcode`* must be of format I4. The values for this variable are returned from the Db2 `SQLCODE` of the underlying `FETCH` operation. This allows the application to react to different statuses encountered while the scrollable cursor is open. The most important status codes indicated by `SQLCODE` are listed in the following table:

SQLCODE	Explanation
`0`	`FETCH` operation successful, data returned except for `FETCH` with option `BEFORE` or `AFTER`.
`+100`	Row not found, cursor still open, no data returned.
`-1`	General error while trying to `FETCH` a row

If you specify `GIVING [:]` *`sqlcode`*, the application must react to the different statuses. If an `SQLCODE +100` is entered five times successively and without terminal I/O, the Natural for Db2 runtime will issue Natural error NAT3296 in order to avoid application looping. The application can terminate the processing loop by executing an `ESCAPE` statement.

If you do not specify `GIVING [:]` *`sqlcode`*, except for `SQLCODE 0` and `SQLCODE +100`, each `SQLCODE` will generate Natural error NAT3700 and the processing loop will be terminated. `SQLCODE +100` (row not found) will terminate the processing loop.

See also the example program `DEM2SCRL` supplied in the Natural system library `SYSDB2`.

Join Queries

A join is a query in which data is retrieved from more than one table. All the tables involved must be specified in the `FROM` clause.

A join always forms the Cartesian product of the tables listed in the `FROM` clause and later eliminates from this Cartesian product table all the rows that do not satisfy the join condition specified in the `WHERE` clause.

Correlation names can be used to save writing if table names are rather long. Correlation names must be used when a column specified in the selection list exists in more than one of the tables to be joined in order to know which of the identically named columns to select.

Example

```
DEFINE DATA LOCAL
1 #NAME     (A20)
1 #MONEY    (I4)
END-DEFINE
...
SELECT NAME, ACCOUNT
  INTO #NAME, #MONEY
  FROM  SQL-PERSONNEL P, SQL-FINANCE F
  WHERE P.PERSNR = F.PERSNR
    AND F.ACCOUNT > 10000
    ...
```

125 SEND METHOD

```
SEND [METHOD] operand1 TO [OBJECT] operand2

  [      {  operand3  [ (AD= { M } ) ]  }    ]
  [ WITH {                   { O }      }... ]
  [      {  nX               { A }      }    ]

[RETURN operand4]
[GIVING operand5]
```

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

Related Statements: `CREATE OBJECT` | `DEFINE CLASS` | `INTERFACE` | `METHOD` | `PROPERTY`

Belongs to Function Group: ***Component Based Programming***

Function

The `SEND METHOD` statement is used to invoke a particular method of an object.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S				A													yes	no
operand2		S																O	no	no
operand3	C	S	A	G		A	U	N	P	I	F	B	D	T	L	C	G	O	yes	no
operand4		S	A			A	U	N	P	I	F	B	D	T	L	C	G	O	yes	no
operand5		S			N					I									yes	no

The formats C and G can only be passed to methods of local classes.

Syntax Element Description:

Syntax Element	Description

operand1	**Method-Name:**
	
	operand1 is the name of a method which is supported by the object specified in *operand2*.
	
	Since the method names can be identical in different interfaces of a class, the method name in *operand1* can also be qualified with the interface name to avoid ambiguity.
	
	In the following example, the object `#O3` has an interface `Iterate` with the method `Start`. The following statements apply:
	
	`* Specifying only the method name.`
	`SEND 'Start' TO #O3`
	`* Qualifying the method name with the interface name.`
	`SEND 'Iterate.Start' TO #O3`
	
	If no interface name is specified, Natural searches the method name in all the interfaces of the class. If the method name is found in more than one interface, a runtime error occurs.

operand2	**Object Handle:** The handle of the object to which the method call is to be sent. *operand2* must be defined as an object handle (`HANDLE OF OBJECT`). The object must already exist. To invoke a method of the current object inside a method, use the system variable `*THIS-OBJECT`.
operand3	**Parameter(s) Specific to the Method:** As *operand3* you can specify parameters specific to the method. In the following example, the object `#O3` has the method `PositionTo` with the parameter `Pos`. The method is called in the following way: `SEND 'PositionTo' TO #O3 WITH Pos` Methods can have optional parameters. Optional parameters need not to be specified when the method is called. To omit an optional parameter, use the placeholder 1X. To omit *n* optional parameters, use the placeholder *n*X. In the following example, the method `SetAddress` of the object `#O4` has the parameters `FirstName`, `MiddleInitial`, `LastName`, `Street` and `City`, where `MiddleInitial`, `Street` and `City` are optional. The following statements apply:

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td></td><td><pre>* Specifying all parameters.
SEND 'SetAddress' TO #O4 WITH FirstName MiddleInitial
LastName Street City
* Omitting one optional parameter.
SEND 'SetAddress' TO #O4 WITH FirstName 1X LastName Street City
* Omitting all optional parameters.
SEND 'SetAddress' TO #O4 WITH FirstName 1X LastName 2X</pre>
Omitting a non-optional (mandatory) parameter results in a runtime error.</td></tr>
<tr><td><code>AD=</code></td><td>Attribute Definition:
If operand3 is a variable, you can mark it in one of the following ways:
<table>
<tr><td><code>AD=O</code></td><td>Non-modifiable, see session parameter <code>AD=O</code>.</td></tr>
<tr><td><code>AD=M</code></td><td>Modifiable, see session parameter <code>AD=M</code>.

This is the default setting.</td></tr>
<tr><td><code>AD=A</code></td><td>Input only, see session parameter <code>AD=A</code>.</td></tr>
</table>
If operand3 is a constant, <code>AD</code> cannot be explicitly specified. For constants <code>AD=O</code> always applies.</td></tr>
<tr><td>n<code>X</code></td><td>Parameter(s) to be Skipped:

With the notation n<code>X</code> you can specify that the next n parameters are to be skipped (for example, <code>1X</code> to skip the next parameter, or <code>3X</code> to skip the next three parameters). This means that for the next n parameters no values are passed to the method.

For a method implemented in Natural, a parameter that is to be skipped must be defined with the keyword <code>OPTIONAL</code> in the dialog's <code>DEFINE DATA PARAMETER</code> statement. <code>OPTIONAL</code> means that a value can - but need not - be passed from the invoking object to such a parameter.</td></tr>
<tr><td><code>RETURN</code> operand4</td><td>RETURN Clause:

If the <code>RETURN</code> clause is omitted and the method has a return value, the return value is discarded.

If the <code>RETURN</code> clause is specified, operand4 contains the return value of the method. If the method execution fails, operand4 is reset to its initial value.

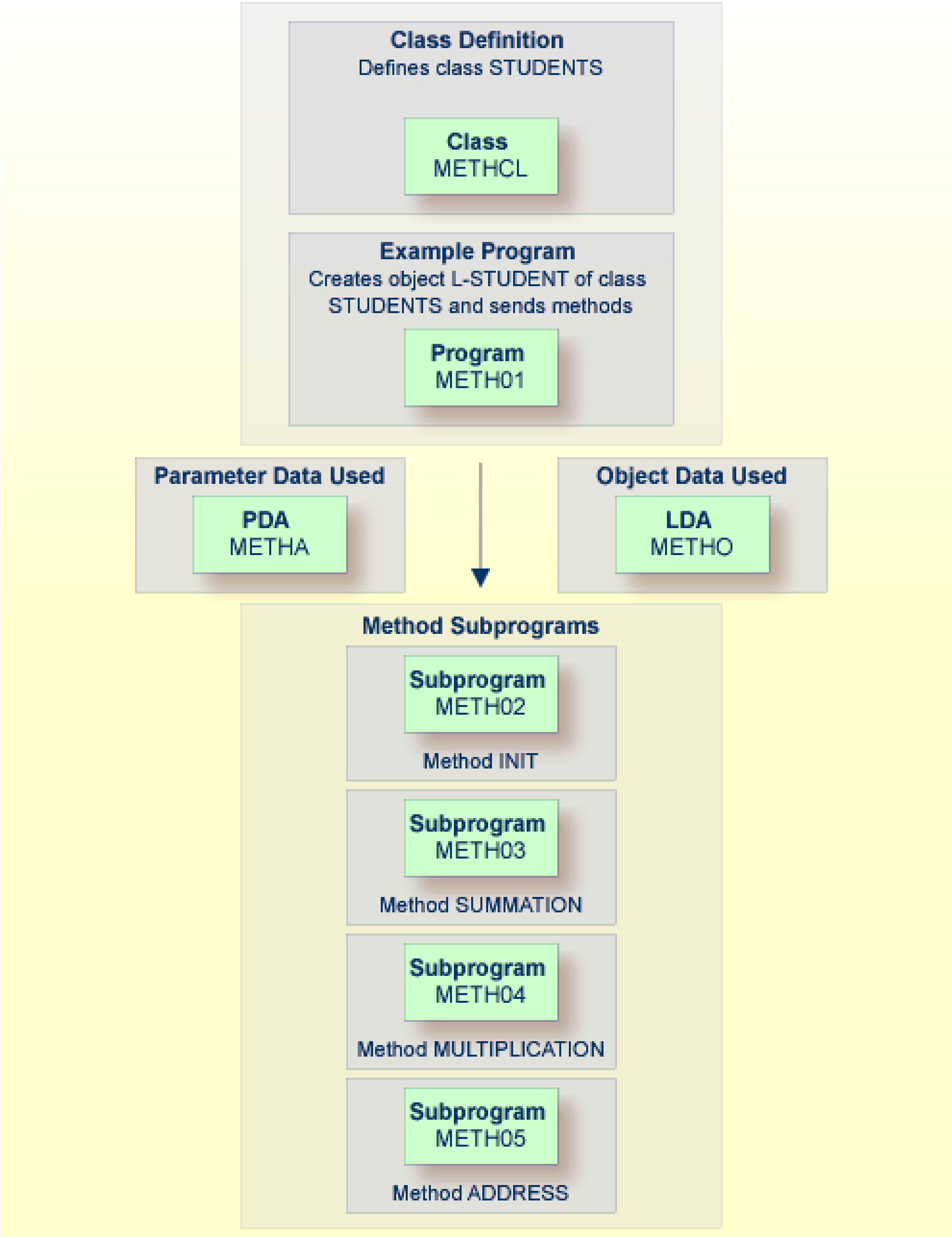
Note: For classes written in Natural, the return value of a method is defined by entering one additional parameter in the parameter data area of the method and by marking it with <code>BY VALUE RESULT</code>. For more information, see the <code>PARAMETER</code> clause in the <code>INTERFACE</code> statement description. Therefore the parameter data area of a method that is written in Natural and that has a return value always contains one additional field next to the method parameters. This is to be considered when you call a method of a Natural written class and want to use the parameter data area of the method in the <code>SEND</code> statement.</td></tr>
<tr><td><code>GIVING</code> operand5</td><td>GIVING Clause:

If the <code>GIVING</code> clause is not specified, the Natural run time error processing is triggered if an error occurs.</td></tr>
</table>

Syntax Element	Description
	If the `GIVING` clause is specified, *`operand5`* contains the Natural message number if an error occurred, or zero on success.

Example

The following diagram gives an overview of the Natural objects that are used in this example. The corresponding source code and the program output are shown below.


Class Definition
Defines class STUDENTS
Class
METHCL
Example Program
Creates object L-STUDENT of class STUDENTS and sends methods
Program
METH01
Parameter Data Used
PDA
METHA
Object Data Used
LDA
METHO
Method Subprograms
Subprogram
METH02
Method INIT
Subprogram
METH03
Method SUMMATION
Subprogram
METH04
Method MULTIPLICATION
Subprogram
METH05
Method ADDRESS

Program METH01 - CREATE OBJECT and SEND METHOD Using a Class and Several Methods:

```
** Example 'METH01':  CREATE OBJECT and SEND METHOD
**                    using a class and several methods (see METH*)
************************************************************************
DEFINE DATA
LOCAL
  USING METHA
LOCAL
1 L-STUDENT HANDLE OF OBJECT
1 #NAME     (A20)
1 #STREET   (A20)
1 #CITY     (A20)
1 #SUM      (I4)
1 #MULTI    (I4)
END-DEFINE
*
CREATE OBJECT L-STUDENT OF CLASS 'STUDENTS' /* see METHCL for class
*
L-STUDENT.<> := 'John Smith'
*
SEND METHOD 'INIT' TO L-STUDENT             /* see METHCL
     WITH #VAR1 #VAR2 #VAR3 #VAR4
*
SEND METHOD 'SUMMATION' TO L-STUDENT        /* see METHCL
     WITH #VAR1 #VAR2 #VAR3 #VAR4
*
SEND METHOD 'MULTIPLICATION' TO L-STUDENT   /* see METHCL
     WITH #VAR1 #VAR2 #VAR3 #VAR4
*
#NAME  := L-STUDENT.<>
#SUM   := L-STUDENT.<>
#MULTI := L-STUDENT.<>
*
SEND METHOD 'ADDRESS' TO L-STUDENT          /* see METHCL
*
#STREET := L-STUDENT.<>
#CITY   := L-STUDENT.<>
*
*
WRITE 'Name  :' #NAME
WRITE 'Street:' #STREET
WRITE 'City  :' #CITY
WRITE ' '
WRITE 'The summation of     ' #VAR1 #VAR2 #VAR3 #VAR4
WRITE 'is' #SUM
WRITE 'The multiplication of' #VAR1 #VAR2 #VAR3 #VAR4
WRITE 'is' #MULTI
*
END
```

Class Definition METHCL Used by METH01:

```
** Example 'METHCL': DEFINE CLASS (used by METH01)
************************************************************************
* Defining class STUDENTS for METH01
*
DEFINE CLASS STUDENTS
  OBJECT
    USING METHO                  /* Object data for class STUDENTS
  /*
  INTERFACE STUDENT-ARITHMETICS
    PROPERTY FULL-NAME
      IS NAME
    END-PROPERTY
    PROPERTY SUM
    END-PROPERTY
    PROPERTY MULTI
    END-PROPERTY
*
    METHOD INIT
      IS METH02
      PARAMETER USING METHA
    END-METHOD
    METHOD SUMMATION
      IS METH03
      PARAMETER USING METHA
    END-METHOD
    METHOD MULTIPLICATION
      IS METH04
      PARAMETER USING METHA
    END-METHOD
  END-INTERFACE
*
  INTERFACE STUDENT-ADDRESS
    PROPERTY STUDENT-NAME
      IS NAME
    END-PROPERTY
    PROPERTY STREET
    END-PROPERTY
    PROPERTY CITY
    END-PROPERTY
*
    METHOD ADDRESS
      IS METH05
    END-METHOD
  END-INTERFACE
END-CLASS
END
```

Local Data Area METHO (object data) Used by Class METHCL and Subprograms METH02, METH03, METH04 and METH05:

```
Local     METHO     Library SYSEXSYN                       DBID    10 FNR    32
Command                                                                   > +
I T L  Name                             F Length     Miscellaneous
All -- -------------------------------- - ---------- ------------------------->
     1 NAME                             A         20
     1 STREET                           A         30
     1 CITY                             A         20
     1 SUM                              I          4
     1 MULTI                            I          4
```

Parameter Data Area METHA Used by Program METH01, Class METHCL and Subprograms METH02, METH03 and METH04:

```
Parameter METHA     Library SYSEXSYN                       DBID    10 FNR    32
Command                                                                   > +
I T L  Name                             F Length     Miscellaneous
All -- -------------------------------- - ---------- ------------------------->
     1 #VAR1                            I          4
     1 #VAR2                            I          4
     1 #VAR3                            I          4
     1 #VAR4                            I          4
```

Subprogram METH02 - Method INIT Used by Program METH01:

```
** Example 'METH02': Method INIT (used by METH01)
************************************************************************
DEFINE DATA
PARAMETER
  USING METHA
OBJECT
  USING METHO
END-DEFINE
*
* Method INIT of class STUDENTS
*
#VAR1 := 1
#VAR2 := 2
#VAR3 := 3
#VAR4 := 4
*
END
```

Subprogram METH03 - Method SUMMATION Used by Program METH01:

```
** Example 'METH03': Method SUMMATION (used by METH01)
************************************************************************
DEFINE DATA
PARAMETER
  USING METHA
OBJECT
  USING METHO
END-DEFINE
*
* Method SUMMATION of class STUDENTS
*
COMPUTE SUM = #VAR1 + #VAR2 + #VAR3 + #VAR4
END
```

Subprogram METH04 - Method MULTIPLICATION Used by Program METH01:

```
** Example 'METH04': Method MULTIPLICATION (used by METH01)
************************************************************************
DEFINE DATA
PARAMETER
  USING METHA
OBJECT
  USING METHO
END-DEFINE
*
* Method MULTIPLICATION of class STUDENTS
*
COMPUTE MULTI = #VAR1 * #VAR2 * #VAR3 * #VAR4
END
```

Subprogram METH05 - Method ADDRESS Used by Program METH01:

```
** Example 'METH05': Method ADDRESS (used by METH01)
************************************************************************
DEFINE DATA
  OBJECT USING METHO
END-DEFINE
*
* Method ADDRESS of class STUDENTS
*
IF NAME = 'John Smith'
  STREET := 'Oxford street'
  CITY   := 'London'
END-IF
END
```

Output of Program METH01:

```
Page      1                                                05-01-17  15:59:04

Name  : John Smith
Street: Oxford street
City  : London

The summation of               1           2           3           4
is          10
The multiplication of          1           2           3           4
is          24
```

126 SEPARATE

```
SEPARATE { operand1                                     }
         { SUBSTRING (operand1,operand2,operand3)        }
         [[STARTING] FROM [POSITION] operand8]
         [LEFT [JUSTIFIED]] INTO operand4...
         [ IGNORE                     ]
         [ REMAINDER operand5         ]
         [ REMAINDER POSITION operand9 ]
         [ WITH [RETAINED] { [ANY] DELIMITERS           } ]
         [                 { INPUT DELIMITERS           } ]
         [                 { DELIMITERS operand6        } ]
         [[GIVING] NUMBER [IN] operand7]
```

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

Related Statements: COMPRESS | COMPUTE | EXAMINE | MOVE | MOVE ALL | RESET

Belongs to Function Group: ***Arithmetic and Data Movement Operations***

Function

The SEPARATE statement is used to separate the content of an alphanumeric or binary operand into two or more alphanumeric or binary operands (or into multiple occurrences of an alphanumeric or binary array).

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A			A	U					B							yes	no
operand2	C	S						N	P	I		B*							yes	no
operand3	C	S						N	P	I		B*							yes	no
operand4		S	A	G		A	U					B							yes	yes
operand5		S				A	U					B							yes	yes
operand6	C	S				A	U					B							yes	no
operand7		S						N	P	I									yes	yes

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand8	C	S						N	P	I									yes	no
operand9		S						N	P	I									yes	yes

* Format B of *operand2* and *operand3* may be used only with a length of less than or equal to 4.

Syntax Element Description:

Syntax Element	Description
operand1	**Source Operand:** *operand1* is the alphanumeric/binary constant or variable whose content is to be separated. Trailing blanks in *operand1* are removed before the value is processed (even if the blank is used as a delimiter character; see also the `DELIMITERS` option).
`SUBSTRING`	**SUBSTRING Option:** Normally, the whole content of a field is separated, starting from the beginning of the field. The `SUBSTRING` option allows you to separate only a certain part of the field. After the field name (*operand1*) in the `SUBSTRING` clause you specify first the starting position (*operand2*) and then the length (*operand3*) of the field portion to be separated. For example, if a field `#A` contained `CONTRAPTION`, `SUBSTRING(#A,5,3)` would contain `RAP`. **Note:** If you omit *operand2*, the starting position is assumed to be `1`. If you omit *operand3*, the length is assumed to be from the starting position to the end of the field.
`STARTING FROM POSITION` *operand8*	**STARTING FROM POSITION Option:** This option determines the starting position for the source operand (*operand1*) to be separated. For details, see ***Defining Ranges for STARTING POSITION***.
`LEFT JUSTIFIED`	**LEFT JUSTIFIED Option:** This option causes leading blanks which may occur between the delimiter character and the next non-blank character to be removed from the target operand.
operand4	**Target Operand:** *operand4* represents the target operands. If an array is specified as target operand, it is filled occurrence by occurrence with the separated values. The number of target operands corresponds to the number of delimiter characters (including trailing delimiter characters) in *operand1*, plus 1.

Syntax Element	Description
	If *operand4* is a dynamic variable, its length may be modified by the SEPARATE operation. The current length of a dynamic variable can be ascertained by using the system variable *LENGTH. For general information on dynamic variables, see the section *Using Dynamic and Large Variables*.
IGNORE / REMAINDER *operand5*	**IGNORE / REMAINDER Options:** If you do not specify enough target fields for the source value to be separated into, you will receive an appropriate error message. To avoid this, you have two options: ■ **IGNORE Option:** If you specify IGNORE, Natural will ignore it if there are not enough target operands to receive the source value. ■ **REMAINDER Option:** If you specify REMAINDER *operand5*, that section of the source value which could not be placed into target operands will be placed into *operand5*. You may then use the content of *operand5* for further processing, for example in a subsequent SEPARATE statement. REMAINDER can only be used for single-value source operands. For array source operands, use the REMAINDER POSITION option. See also ***Rules and Operational Considerations*** and ***Example 3***.
REMAINDER POSITION *operand9*	**REMAINDER POSITION Option:** The value returned by the REMAINDER POSITION clause corresponds to the position from which a REMAINDER data field is filled. For details, see ***Rules and Operational Considerations***.
DELIMITERS	**DELIMITERS Option:** See ***DELIMITERS Option*** below.
RETAINED	**RETAINED Option:** Normally, the delimiter characters themselves are not moved into the target operands. When you specify RETAINED, however, each delimiter (that is, either default delimiters and blanks, or the delimiter specified with *operand6*) will also be placed into a target operand. Example: The following SEPARATE statement would place 150 into #B, + into #C, and 30 into #D:

Syntax Element	Description

	`...`
	`MOVE '150+30' TO #A`
	`SEPARATE #A INTO #B #C #D WITH RETAINED DELIMITER '+'`
	`...`
	
	See also *Example 3*.

`GIVING NUMBER` *operand7*	**GIVING NUMBER Option:** This option causes the number of filled target operands (including those filled with blanks) to be returned in *operand7*. The number actually obtained is the number of delimiters plus 1. If you use the ***IGNORE Option***, the maximum possible number returned in *operand7* will be the number of target operands (*operand4*). If you use the ***REMAINDER Option***, the maximum possible number returned in *operand7* will be the number of target operands (*operand4*) plus 1 (for *operand5*).

DELIMITERS Option:

Delimiter characters within *operand1* indicate the positions at which the value is to be separated.

Syntax Element Description:

Syntax Element	Description
`WITH [ANY] DELIMITERS`	If you omit the `DELIMITERS` option or specify `WITH ANY DELIMITERS`, a blank and any character which is neither a letter nor a numeric character will be treated as delimiter character.
`WITH INPUT DELIMITERS`	Indicates that the blank and the default input delimiter character (as specified with the session parameter `ID`) are to be used as delimiter character.
`WITH DELIMITERS` *operand6*	Indicates that each of the characters specified in *operand6* is to be treated as delimiter character. If *operand6* contains trailing blanks, these will be ignored.

Rules and Operational Considerations

- Processing of Source and Target Operands
- Defining Ranges for STARTING FROM POSITION
- Values Returned by REMAINDER POSITION
- Overlapping Fields: REMAINDER and REMAINDER POSITION

- Delimiters in SEPARATE

Processing of Source and Target Operands

Trailing blanks are ignored in source operands (in single values and array occurrences as well) when the separation process starts. Trailing blanks only count when the REMAINDER POSITION value is calculated: see also *Values Returned by REMAINDER POSITION*.

If the source operand (*operand1*) is an empty dynamic field (*LENGTH=0) or an X-array that is not expanded, the SEPARATE statement stops executing after resetting the following fields:

- all target operands (*operand4*);
- the field (*operand7*) returning the number of filled target operands;
- the REMAINDER data field (*operand5*);
- the REMAINDER POSITION field (*operand9*)

The same applies if the source operand contains only blanks.

Defining Ranges for STARTING FROM POSITION

The value range allowed for the STARTING FROM POSITION clause *operand8* is 1:*n* where *n* is the last byte of the source field.

If the source operand (*operand1*) is an array, all occurrences are counted, including trailing blanks. For a dynamic array, the length of each individual field is counted, up to the specified position.

Examples of *operand8*:

Position 63 in #A (A100)	is the 63rd char in #A.
Position 63 in #B (A20/1:10)	is the 3rd char in #B(4).
Position 63 in #C (A10/1:3,1:4)	is the 3rd char in #C(2,3).
Position 63 in #D (1:5) DYNAMIC with *LENGTH(#D(*)) = (15,25,0,33,61)	is the 23rd char in #D(4).

If you specify an invalid range (a negative or zero value, or a value greater than the actual field length), the return fields listed in *Processing of Source and Target Operands* are reset, but no runtime error occurs. Since the STARTING FROM value denotes a position (and not an offset), *operand8* requires a minimum value of 1 for the first execution.

Values Returned by REMAINDER POSITION

The value returned by the REMAINDER POSITION clause corresponds to the position from which a REMAINDER data field is filled.

Example:

```
...
SEPARATE 'AB  CD' INTO #A REMAINDER #R
...
```

The above statement returns #A= 'AB' and #R= ' CD' as the REMAINDER starts after the separator character (here: a blank), right after AB. With the REMAINDER POSITION option used instead, a value of 4 would be returned.

Although trailing blanks are ignored during the separation process, they are taken into account for the calculation of the REMAINDER POSITION value in occurrences of a source array.

If all source segments are processed and the end of the source field is reached, REMAINDER POSITION returns a value of zero indicating "no more data".

See also ***Example 6 - Using a Source Array with STARTING FROM and REMAINDER POSITION***.

Overlapping Fields: REMAINDER and REMAINDER POSITION

When the SEPARATE statement is executed, the source data (*operand1*) is usually copied and processed from a work field. Therefore, the REMAINDER result is independent of possibly overlapping source and result fields.

Such field backup copies are not produced if a REMAINDER POSITION clause is used. The complete separation process operates on the original source operand, regardless of whether you separate the source and target operands. Overlapping operands are neither rejected during compilation nor execution but can cause undesired results.

Delimiters in SEPARATE

When you separate a single-value field, the field border always delimits the last word. The same applies to each occurrence of an array field.

If the RETAINED DELIMITERS option is used, delimiters are also placed into the target field. This only applies to delimiter characters within an array occurrence, and not to consecutive array occurrences that are automatically delimited (without delimiter character) when an occurrence ends.

See also ***Example 4 - Using a Source Array of a Redefined String*** and ***Example 5 - Using a Source Array with RETAINED Delimiters***.

Examples

Example 1 - Various Samples

```
** Example 'SEPEX1': SEPARATE
************************************************************************
DEFINE DATA LOCAL
1 #TEXT1   (A6) INIT <'AAABBB'>
1 #TEXT2   (A7) INIT <'AAA BBB'>
1 #TEXT3   (A7) INIT <'AAA-BBB'>
1 #TEXT4   (A7) INIT <'A.B/C,D'>
1 #FIELD1A (A6)
1 #FIELD1B (A6)
1 #FIELD2A (A3)
1 #FIELD2B (A3)
1 #FIELD3A (A3)
1 #FIELD3B (A3)
1 #FIELD4A (A3)
1 #FIELD4B (A3)
1 #FIELD4C (A3)
1 #FIELD4D (A3)
1 #NBT     (N1)
1 #DEL     (A5)
END-DEFINE
*
WRITE NOTITLE 'EXAMPLE A (SOURCE HAS NO BLANKS)'
SEPARATE #TEXT1 INTO #FIELD1A #FIELD1B GIVING NUMBER #NBT
WRITE     / '=' #TEXT1 5X '=' #FIELD1A 4X '=' #FIELD1B 4X '=' #NBT
*
WRITE NOTITLE /// 'EXAMPLE B (SOURCE HAS EMBEDDED BLANK)'
SEPARATE #TEXT2 INTO #FIELD2A #FIELD2B GIVING NUMBER #NBT
WRITE     / '=' #TEXT2 4X '=' #FIELD2A 7X '=' #FIELD2B 7X '=' #NBT
*
WRITE NOTITLE /// 'EXAMPLE C (USING DELIMITER ''-'')'
SEPARATE #TEXT3 INTO #FIELD3A #FIELD3B WITH DELIMITER '-'
WRITE     /    '=' #TEXT3 4X '=' #FIELD3A 7X '=' #FIELD3B
*
MOVE ',/' TO #DEL
WRITE NOTITLE /// 'EXAMPLE D USING DELIMITER' '=' #DEL
*
SEPARATE #TEXT4 INTO #FIELD4A #FIELD4B
```

```
         #FIELD4C #FIELD4D WITH DELIMITER #DEL
WRITE     /    '=' #TEXT4 4X '=' #FIELD4A 7X '=' #FIELD4B
          /              19X '=' #FIELD4C 7X '=' #FIELD4D
*
END
```

Output of Program SEPEX1:

```
EXAMPLE A (SOURCE HAS NO BLANKS)

#TEXT1: AAABBB     #FIELD1A: AAABBB    #FIELD1B:           #NBT:  1



EXAMPLE B (SOURCE HAS EMBEDDED BLANK)

#TEXT2: AAA BBB    #FIELD2A: AAA       #FIELD2B: BBB       #NBT:  2



EXAMPLE C (USING DELIMITER '-')

#TEXT3: AAA-BBB    #FIELD3A: AAA       #FIELD3B: BBB



EXAMPLE D USING DELIMITER #DEL: ,/

#TEXT4: A.B/C,D    #FIELD4A: A.B       #FIELD4B: C
                   #FIELD4C: D         #FIELD4D:
```

Example 2 - Using an Array

```
** Example 'SEPEX2': SEPARATE (using array variable)
************************************************************************
DEFINE DATA LOCAL
1 #INPUT-LINE (A60) INIT <'VALUE1,   VALUE2,VALUE3'>
1 #FIELD      (A20/1:5)
1 #NUMBER     (N2)
END-DEFINE
*
SEPARATE #INPUT-LINE LEFT JUSTIFIED INTO #FIELD (1:5)
                     GIVING NUMBER IN #NUMBER
*
WRITE NOTITLE #INPUT-LINE //
              #FIELD (1)  /
              #FIELD (2)  /
              #FIELD (3)  /
              #FIELD (4)  /
              #FIELD (5)  /
              #NUMBER
```

```
*
END
```

Output of Program SEPEX2:

```
VALUE1,   VALUE2,VALUE3

VALUE1
VALUE2
VALUE3


  3
```

Example 3 - Using REMAINDER/RETAINED Options

```
** Example 'SEPEX3': SEPARATE (with REMAINDER, RETAIN option)
************************************************************************
DEFINE DATA LOCAL
1 #INPUT-LINE (A60) INIT <'VAL1,   VAL2, VAL3,VAL4'>
1 #FIELD      (A10/1:4)
1 #REM        (A30)
END-DEFINE
*
WRITE TITLE LEFT 'INP:' #INPUT-LINE /
            '#FIELD (1)' 13T '#FIELD (2)' 25T '#FIELD (3)'
        37T '#FIELD (4)' 49T 'REMAINDER'
       /    '----------' 13T '----------' 25T '----------'
        37T '----------' 49T '------------------------------'
*
SEPARATE #INPUT-LINE INTO #FIELD (1:2)
         REMAINDER #REM WITH DELIMITERS ','
WRITE #FIELD(1) 13T #FIELD(2) 25T #FIELD(3) 37T #FIELD(4) 49T #REM
*
RESET #FIELD(*) #REM
SEPARATE #INPUT-LINE INTO #FIELD (1:2)
         IGNORE WITH DELIMITERS ','
WRITE #FIELD(1) 13T #FIELD(2) 25T #FIELD(3) 37T #FIELD(4) 49T #REM
*
RESET #FIELD(*) #REM
SEPARATE #INPUT-LINE INTO #FIELD (1:4) IGNORE
        WITH RETAINED DELIMITERS ','
WRITE #FIELD(1) 13T #FIELD(2) 25T #FIELD(3) 37T #FIELD(4) 49T #REM
*
RESET #FIELD(*) #REM
*
SEPARATE SUBSTRING(#INPUT-LINE,1,50) INTO #FIELD (1:4)
         IGNORE WITH DELIMITERS ','
WRITE #FIELD(1) 13T #FIELD(2) 25T #FIELD(3) 37T #FIELD(4) 49T #REM
*
END
```

Output of Program SEPEX3:

```
INP: VAL1,   VAL2, VAL3,VAL4
#FIELD (1)  #FIELD (2)  #FIELD (3)  #FIELD (4)  REMAINDER
----------  ----------  ----------  ----------  ------------------------------
VAL1           VAL2                              VAL3,VAL4
VAL1           VAL2
VAL1         ,              VAL2      ,
VAL1           VAL2      VAL3       VAL4
```

Example 4 - Using a Source Array of a Redefined String

```
**  Example 'SEPEX4': SEPARATE with source array
************************************************************************
* This example shows different results when separating a scalar string
* or a string array redefining the scalar string.
*
*
************************************************************************
*
*
DEFINE DATA LOCAL
1 #TEXT (A24) INIT <'VAL1 VAL2 VAL3 VAL4 VAL5'>
1 REDEFINE #TEXT
  2 #TEXTARRAY (A12/2)
1 #WORD1(A5/6)
1 #WORD2(A5/6)
END-DEFINE
*
SEPARATE #TEXT INTO #WORD1(*)
/* Redefinition may split original words into two parts
SEPARATE #TEXTARRAY(*) INTO #WORD2(*)
*
DISPLAY #TEXT #WORD1(*) #TEXTARRAY(*) #WORD2(*)
END
```

Output of Program SEPEX4:

```
         #TEXT           #WORD1  #TEXTARRAY  #WORD2
------------------------ ------ ----------- ------

VAL1 VAL2 VAL3 VAL4 VAL5 VAL1   VAL1 VAL2 VA VAL1
                         VAL2   L3 VAL4 VAL5 VAL2
                         VAL3                VA
                         VAL4                L3
                         VAL5                VAL4
                                             VAL5
```

Example 5 - Using a Source Array with RETAINED Delimiters

```
**  Example 'SEPEX5': SEPARATE with and without RETAINED DELIMITERS
************************************************************************
* This example shows different results with a source array
* when using the option RETAINED DELIMITERS or not.
*
*
************************************************************************
*
*
DEFINE DATA LOCAL
1 #TEXT(A20)        INIT <'VAL1,VAL2,VAL3,VAL4'>
1 #TEXTARRAY(A10/3) INIT <'VAL1,VAL2',
                          'VAL3',
                          'VAL4'>
1 #WORD1(A5/7)
1 #WORD2(A5/7)
END-DEFINE
*
SEPARATE #TEXT         INTO #WORD1(*)
SEPARATE #TEXTARRAY(*) INTO #WORD2(*)
DISPLAY #TEXT #WORD1(*) #TEXTARRAY(*) #WORD2(*)
*
SEPARATE #TEXT         INTO #WORD1(*) WITH RETAINED DELIMITERS
SEPARATE #TEXTARRAY(*) INTO #WORD2(*) WITH RETAINED DELIMITERS
DISPLAY #TEXT #WORD1(*) #TEXTARRAY(*) #WORD2(*)
*
END
```

Output of Program SEPEX5:

```
       #TEXT         #WORD1 #TEXTARRAY #WORD2
-------------------- ------ ---------- ------

VAL1,VAL2,VAL3,VAL4  VAL1   VAL1,VAL2  VAL1
                     VAL2   VAL3       VAL2
                     VAL3   VAL4       VAL3
                     VAL4              VAL4



VAL1,VAL2,VAL3,VAL4  VAL1   VAL1,VAL2  VAL1
                     ,      VAL3       ,
                     VAL2   VAL4       VAL2
                     ,                 VAL3
                     VAL3              VAL4
                     ,
                     VAL4
```

Example 6 - Using a Source Array with STARTING FROM and REMAINDER POSITION

```
**  Example 'SEPEX6': SEPARATE with STARTING FROM and REMAINDER POSITION
************************************************************************
* This example shows how the options STARTING FROM POSITION and
* REMAINDER POSITION work together in a processing loop when
* separating a source array.
*
************************************************************************
*
*
DEFINE DATA LOCAL
1 #TEXT (A15/1:3) INIT <'VAL1 VAL2',
                        'VAL3',
                        'VAL4 VAL5 VAL6'>
1 #WORD (A5/1:4)
1 #POS  (I1) INIT <1>
END-DEFINE
*
WRITE '#TEXT(A15/1:3):      (1)             (2)             (3)'
  /  16T #TEXT(*)
  /  16T '----+----1----+ ----2----+----3 ----+----4----+'
  // '#WORD (A5/1:4): (1)   (2)   (3)   (4)      :  #POS'
     '(within #TEXT(*))'
*
REPEAT
  SEPARATE #TEXT(*) STARTING FROM POSITION #POS
      INTO #WORD(*) REMAINDER POSITION #POS
  WRITE 16T #WORD(*) 44T ': ' #POS
  UNTIL #POS = 0
END-REPEAT
END
```

Output of Program SEPEX6

```
#TEXT(A15/1:3):      (1)             (2)             (3)
               VAL1 VAL2       VAL3            VAL4 VAL5 VAL6
               ----+----1----+ ----2----+----3 ----+----4----+

#WORD (A5/1:4): (1)   (2)   (3)   (4)      :  #POS (within #TEXT(*))
               VAL1  VAL2  VAL3  VAL4      :    36
               VAL5  VAL6                  :     0
```

127 SET CONTROL

```
SET CONTROL operand1 ...
```

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

Function

The `SET CONTROL` statement is used to perform terminal commands from within a program.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S				A													yes	no

Syntax Element Description:

Syntax Element	Description
operand1	**Terminal Commands to be Performed:** The terminal commands are specified as *operand1* without the control character `%` (by default). They can be specified as a text constant or as the content of an alphanumeric variable. For further information on terminal commands, see the *Terminal Commands* documentation.

Examples

- Example 1 - Switching to Lower Case

- Example 2 - Activating Hardcopy Output Destination

Example 1 - Switching to Lower Case

```
...
SET CONTROL 'L'
...
```

Switches to lower case (equivalent to the terminal command `%L`).

Example 2 - Activating Hardcopy Output Destination

```
...
SET CONTROL 'HDEST'...
```

Activates hardcopy output to destination `DEST` (equivalent to the terminal command `%H`*`destination`*).

128 SET GLOBALS

```
SET GLOBALS {parameter=value} ...
```

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

Function

The `SET GLOBALS` statement is used to set values for session parameters.

The parameters are evaluated either when the program that contains the `SET GLOBALS` statement is compiled, or when it is executed; this depends on the individual parameters.

The parameter settings specified with `SET GLOBALS` remain in effect until the end of the Natural session, unless they are overridden with a subsequent `SET GLOBALS` statement or `GLOBALS` system command. The statement `SET GLOBALS` and the system command `GLOBALS` offer the same parameters for modification. They can both be used in the same Natural session. Parameter values specified with a `GLOBALS` command remain in effect until they are overridden by a new `GLOBALS` command or `SET GLOBALS` statement, the session is terminated, or you log on to another library.

Syntax Description

Syntax Element	Description
parameter=value	**Parameter Specification(s):** In place of *parameter*, specify the name of the parameter to be set. For a list of possible parameters, see ***Parameters*** below. If you specify multiple parameters, you have to separate them from one another by one or more blanks. The parameters can be specified in any order; see also ***Example***. In place of *value*, specify a valid parameter value. For information on valid parameter values, see the descriptions of the individual parameters listed below.

Parameters

Parameters that can be specified with the SET GLOBALS statement		Evaluation (R = at runtime, C = at compilation)
CC	Conditional Program Execution	R
CF	Character for Terminal Commands	R
CO	Compiler Output	R
CPCVERR	Code Page Conversion Error	R
DC	Character for Decimal Point Notation	R
DFOUT	Date Format for Output	R
DFSTACK	Date Format for Stack	R
DFTITLE	Date Format in Default Page Title	R
DO	Display Order of Output Data	R
DU	Dump Generation	R
EJ	Page Eject	R
FCDP	Filler Character for Dynamically Protected Fields	R
FS	Format Specification	R
IA	INPUT Assign Character	R
ID	INPUT Delimiter Character	R
IM	INPUT Mode	R
LE	Limit Error Processing	C
LS	Line Size	RC
LT	Limit of Records Read	R
OPF	Overwriting of Protected Fields by Helproutines	R
PD	NATPAGE Page Data Set	R
PM	Print Mode	C
PS	Page Size	RC
REINP	Internal REINPUT for Invalid Data	R
SA	Sound Terminal Alarm	R
SF	Spacing Factor	C
WH	Wait for Record in Hold Status	R
ZD	Zero Division Check	R
ZP	Zero Printing	RC

The individual session parameters are described in the *Parameter Reference*.

Example

In the example below, the `SET GLOBALS` statement is used to set the maximum number of characters permitted per line to 74 and to limit the number of database records that can be read in processing loops within a Natural program to 5000.

```
SET GLOBALS LS=74 LT=5000
...
```

129 SET KEY

Function

The SET KEY statement is used to assign functions to the following types of keys:

- video terminal PA (program attention) keys,
- PF (program function) keys,
- CLEAR key.

When a SET KEY statement is executed, Natural receives control of the keys during program execution and uses the values assigned to the keys.

The Natural system variable *PF-KEY identifies which key was pressed last.

Note: If a user presses a key to which no function is assigned, either a warning message will be issued prompting the user to press a valid key, or the value ENTR will be placed into the Natural system variable *PF-KEY; that is, Natural will react as if the ENTER key had been pressed (this depends on the Natural profile parameter IKEY as set by the Natural administrator).

Syntax Description

Several structures are possible for this statement.

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

Syntax 1 - Affecting All Keys:

```
         { ALL                      }
         { ON                       }
         { OFF                      }
SET KEY  {                          }
         { COMMAND  { ON  }         }
         {          { OFF }         }
         {                          }
         { NAMED OFF                }
```

Syntax 2 - Affecting Individual Keys:

```
          ⎧ ⎧                    ⎫ ⎡     ⎧ ON                         ⎫ ⎤ ⎫
          ⎪ ⎪ PAn                ⎪ ⎢     ⎪ OFF                        ⎪ ⎥ ⎪
          ⎪ ⎪ PFn                ⎪ ⎢     ⎪ DISABLED                   ⎪ ⎥ ⎪
SET KEY   ⎨ ⎨ CLR                ⎬ ⎢  =  ⎨                            ⎬ ⎥ ⎬
          ⎪ ⎪ DYNAMIC operand1   ⎪ ⎢     ⎪ COMMAND  ⎧ ON  ⎫           ⎪ ⎥ ⎪
          ⎪ ⎩                    ⎭ ⎢     ⎪          ⎩ OFF ⎭           ⎪ ⎥ ⎪
          ⎩                        ⎣     ⎩                            ⎭ ⎦ ⎭ ...
```

Syntax 3 - Affecting Individual Keys:

```
     ⎧ ⎧ PAn              ⎫ ⎡    ⎧ ⎧ PGM     ⎫ ⎫ ⎤                               ⎫
     ⎪ ⎪ PFn              ⎪ ⎢    ⎪ ⎩ PROGRAM ⎭ ⎪ ⎥                               ⎪
     ⎪ ⎨ CLR              ⎬ ⎢ =  ⎨ operand2    ⎬ ⎥  [ NAMED ⎧ operand4 ⎫ ]       ⎪
SET  ⎨ ⎪ DYNAMIC          ⎪ ⎢    ⎪ HELP        ⎪ ⎥          ⎩ OFF      ⎭         ⎬
KEY  ⎪ ⎩ operand1         ⎭ ⎣    ⎩ DATA operand3 ⎭ ⎦                             ⎪
     ⎪                                                                           ⎪
     ⎩ ENTR [ NAMED                      ⎧ operand4 ⎫ ]                          ⎭ ...
                                         ⎩ OFF      ⎭
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1		S				A													yes	no
operand2	C	S				A	U												yes	no
operand3	C	S				A	U												yes	no
operand4	C	S				A	U												yes	no

Making Keys Program-Sensitive and Deactivating Keys

Making a key program-sensitive means that the key will be available for interrogation by the currently active program. If a key is made program-sensitive, pressing the key has the same effect as pressing ENTER. All data that have been entered on the screen are transferred to the program.

Note: PA keys and the CLEAR key, when made program-sensitive, do not cause any data to be transferred from the screen.

The program-sensitivity remains in effect only for the execution of the current program. See also the section *SET KEY Statements on Different Program Levels*.

Examples:

`SET KEY ALL`	This statement causes all keys to be made program-sensitive. All function assignments to any keys are overwritten.
`SET KEY PF2` `SET KEY PF2=PGM`	Each of these statements causes PF2 to be made program-sensitive.
`SET KEY OFF`	This statement de-activates all key settings. The Natural system variable `*PF-KEY` contains `ENTR` after `SET KEY OFF` has been executed.
`SET KEY ON`	This statement re-activates the functions assigned to all keys that had an assignment and re-activates the program-sensitivity of keys that were made program-sensitive before they were de-activated.
`SET KEY PF2=OFF`	This statement de-activates PF2. After execution of `SET KEY PF2=OFF`, the Natural system variable `*PF-KEY` contains `ENTR` if it contained `PF2` before.
`SET KEY PF2=ON`	This statement re-activates the function assigned to PF2 before it was de-activated or made program-sensitive. If no function had been assigned to PF2, it will be made program-sensitive again.

Key Program-Sensitivity and Contents of *PF-KEY

The following example shows the relation between the program-sensitivity of a key and the contents of the system variable `*PF-KEY`.

Assume that PF2 has been made program-sensitive by means of `SET KEY PF2=PGM` and an `INPUT` statement is executed afterwards. The table below shows how user actions and executed Natural statements influence the contents of `*PF-KEY`.

Sequence	Natural Statement Executed / User Action	Contents of *PF-KEY
1	User presses PF2.	`PF2`
2	`SET KEY OFF`	`ENTR`
3	`SET KEY ON`	`PF2`
4	`SET KEY PF2=OFF`	`ENTR`
5	`SET KEY PF2=ON`	`PF2`
6	`SET KEY PF3=OFF`	`PF2`

Assigning Commands/Programs

You can assign a command or program name to a key, and you can delete such an assignment. When the key is pressed, the current program is terminated and the command/program assigned to the key is invoked via the Natural stack. When assigning a command/program, you can also pass parameters to the command/program (see third example below).

You can also assign a terminal command to a key. When the key is pressed, the terminal command assigned to the key is executed.

When *operand2* is specified as a constant, it must be enclosed within apostrophes.

Examples:

`SET KEY PF4='SAVE'`	The command `SAVE` is assigned to PF4.
`SET KEY PF4=#XYX`	The value contained in the variable `#XYZ` is assigned to PF4.
`SET KEY PF6='LIST MAP *'`	The command `LIST`, including the `LIST` parameters `MAP` and `*`, is assigned to PF6.
`SET KEY PF2='%%'`	The terminal command `%%` is assigned to PF2.
`SET KEY PF9=' '`	The command and name previously assigned to PF9 are deleted.

The assignment remains in effect until it is overwritten by another `SET KEY` statement, until the user logs on to another application, or until the end of the Natural session. See also the section *SET KEY Statements on Different Program Levels*.

Note: Before a program invoked via a key is executed, Natural internally issues a `BACKOUT TRANSACTION` statement.

Assigning Input DATA

You can assign a data string (*operand3*) to a key. When the key is pressed, the data string is placed into the input field in which the cursor is currently positioned, and the data are transferred to the executing program (as if ENTER had been pressed).

When *operand3* is specified as a constant, it must be enclosed within apostrophes.

Example:

```
SET KEY PF12=DATA 'YES'
```

For the validity of a DATA assignment, the same applies as for a command assignment, that is, it remains in effect until it is overwritten by another SET KEY statement, until the user logs on to another application, or until the end of the Natural session. See also the section *SET KEY Statements on Different Program Levels*.

COMMAND OFF/ON

With COMMAND OFF, you can temporarily de-activate any function (command, program, or data) assigned to a key. If the key had been program-sensitive before the function was assigned, COMMAND OFF will make it program-sensitive again.

With a subsequent COMMAND ON, you can re-activate the assigned function again.

Examples:

SET KEY PF4=COMMAND OFF	The function that has been assigned to PF4 is temporarily de-activated; if PF4 had been program-sensitive before the function was assigned, it is now made program-sensitive again.
SET KEY PF4=COMMAND ON	The function assigned to PF4 is re-activated again.
SET KEY COMMAND OFF	All functions assigned to all keys are temporarily de-activated; those keys which had been program-sensitive before functions were assigned to them, are now made program-sensitive again.
SET KEY COMMAND ON	All functions assigned to all keys are re-activated again.

With SET KEY PF*nn*='' you can delete the function assigned to a key and at the same time deactivate the program sensitivity of the key.

Assigning HELP

You can assign HELP to a key. When the key is pressed, the helproutine assigned to the field in which the cursor is currently positioned will be invoked.

The effect is the same as when entering the help character in the field to invoke help. (The help character - default is a question mark (?) - is determined by the Natural profile parameter HI as set by the Natural administrator.)

Example:

```
SET KEY PF1=HELP
```

For the validity of a `HELP` assignment, the same applies as for program-sensitivity, that is, it remains in effect only for the execution of the current program. See also the section ***SET KEY Statements on Different Program Levels***.

DYNAMIC Option

Instead of specifying a specific key with the `SET KEY` statement, you can use the `DYNAMIC` option with a variable (*operand1*), and assign a value (`PF`*n*, `PA`*n*, `CLR`) to this variable in the program. This allows you to specify a function and make it dependent on the program logic which key this function is assigned to.

Note: `SET KEY` cannot be used if *operand1* is a dynamic variable.

Example:

```
...
IF ...
   MOVE 'PF4' TO #KEY
ELSE
   MOVE 'PF7' TO #KEY
END-IF
...
SET KEY DYNAMIC #KEY = 'SAVE'
...
```

DISABLED Option

Graphical user interface (GUI) elements, such as push buttons, menus, and bitmaps, are implemented as PF keys. With the `DISABLED` option, you can disable the use the of a GUI element assigned to a PF key. Push buttons and menu items will then be displayed grey.

To cancel the effect of `SET KEY PF`*nn*`=DISABLED`, you use `SET KEY PF`*nn*`=ON`.

Example:

`SET KEY PF10=DISABLED`	Disables the use of the GUI element assigned to PF10.

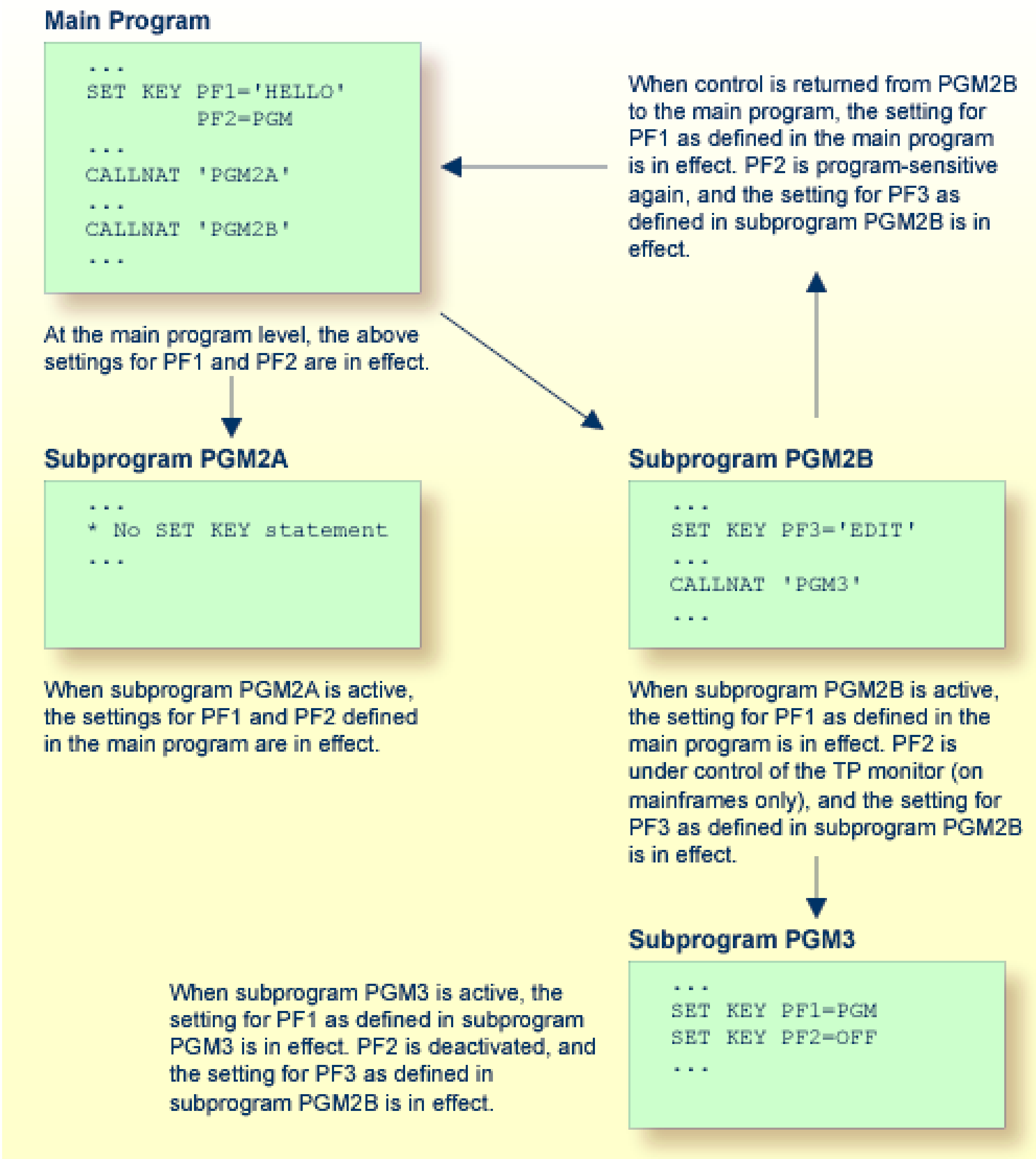
The `DISABLED` option can only be used within a processing rule.

SET KEY Statements on Different Program Levels

When an application contains `SET KEY` statements at different levels, the following applies:

- When keys are made program-sensitive, the program-sensitivity also applies to all lower level (called) programs, unless these programs contain further `SET KEY` statements. When control is returned to a higher level program, the `SET KEY` assignments made at the higher level come into effect again.
- For keys which are defined as `HELP` keys, the same applies as for keys which are program-sensitive.
- When a function (program, command, terminal command, or data string) is assigned to a key, this assignment is valid at all higher and lower levels - regardless of the level at what the assignment is made - until another function is assigned to the key or it is made program-sensitive, or until the user logs on to another application or the Natural session is terminated.

Example of SET KEY Statements on Different Program Levels



Assigning Names

With the NAMED clause, you can assign a name (*operand4*) to a key. The name will then be displayed in the PF-key lines on the screen; this allows the users to identify the functions assigned to the keys:

```
                 ?   Help
               .   Exit
              ---- --------------------------------------------------
      Code ..: ?   Library ..: *_______
                   Object ...: *________________________________________
                   DBID .....: 0__    FILENR ...: 0__


  Command ===>
 Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
       Help        Exit  Last        Flip                                  Canc
```

The display of the PF-key lines is activated with the session parameter KD (see the *Parameter Reference*). You can control the way in which the PF-key lines are displayed by using the terminal command %Y (see the *Terminal Commands* documentation).

The maximum length of a name to be assigned to a key is 10 characters. In normal tabular PF-key line format (%YN), only the first 5 characters are displayed.

When *operand4* is specified as a constant, it must be enclosed within apostrophes (see examples below).

You cannot assign a name to a key without assigning a function to it or making it program-sensitive. To the ENTER key, however, you can only assign a name, but no function.

With NAMED OFF, you delete the name assigned to a program-sensitive key.

Examples:

SET KEY ENTR NAMED 'EXEC'	The name EXEC is assigned to the ENTER key.
SET KEY PF3 NAMED 'EXIT'	PF3 is made program-sensitive, and the name EXIT is assigned to PF3.
SET KEY PF3 NAMED OFF	PF3 is made program-sensitive, and the name that has been assigned to PF3 is deleted.
SET KEY NAMED OFF	All names that have been assigned to any program-sensitive keys are deleted.
SET KEY PF4='AP1' NAMED 'APPL1'	The program AP1 and the name APPL1 are assigned to PF4.

When you use normal tabular PF-key line format (`%YN`), the following applies:

- If you omit the `NAMED` clause when assigning a command/program to a key, the command/program name will be displayed in the PF-key line; if the command/program name is longer than 5 characters, `CMND` will be displayed.
- If you omit the `NAMED` clause when assigning input data to a key, `DATA` will be displayed in the PF-key line.
- If you assign (with the NAMED clause) a name in Unicode format to a PF-key, the name might not be correctly positioned under the respective headers. This problem, however, may occur only when you are using the *Natural Web I/O Interface* and only for "wide" characters. In this case, the sequential PF-key line format (`%YS` or `%YP`) is recommended.

When you use sequential PF-key line format (`%YS` or `%YP`), only those keys to which names have been assigned will be displayed in the PF-key line; that is, if you omit the `NAMED` clause when assigning a command/program/data to a key, the key will not be displayed in the PF-key line.

Example

```
** Example 'SKYEX1': SET KEY
************************************************************************
DEFINE DATA LOCAL
1 #PF4 (A56)
END-DEFINE
*
MOVE 'LIST VIEW' TO #PF4
*
SET KEY PF1 PF2
SET KEY PF3 = 'MENU'
        PF4 = #PF4
        PF5 = 'LIST VIEW EMPLOYEES' NAMED 'Empl'
*
FORMAT KD=ON
INPUT ////
      10X 'The following function keys are assigned:' //
      10X 'PF1: Function for PF1       ' /
      10X 'PF2: Function for PF2       ' /
      10X 'PF3: Return to MENU program' /
      10X 'PF4: LIST VIEW              ' /
      10X 'PF5: LIST VIEW EMPLOYEES    ' ///
*
IF *PF-KEY = 'PF1'
  WRITE 'Function for PF1 executed.'
END-IF
IF *PF-KEY = 'PF2'
  WRITE 'Function for PF2 executed.'
END-IF
```

```
*
END
```

Output of Program SKYEX1:

```
The following function keys are assigned:

PF1: Function for PF1
PF2: Function for PF2
PF3: Return to MENU program
PF4: LIST VIEW
PF5: LIST VIEW EMPLOYEES
```

130 SET TIME

```
{ SET TIME }
{ SETTIME  }
```

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

Function

The SET TIME (or SETTIME) statement is used in conjunction with the Natural system variable *TIMD to measure the time it takes to execute a specific section of a program.

The SET TIME statement is placed at a specific position in the program, and *TIMD will contain the amount of time elapsed since the execution of the SET TIME statement.

*TIMD must always contain a reference to the SET TIME statement, either by using the source-code line number of the SET TIME statement or by assigning a label to the SET TIME statement, which can then be used as a reference.

Example

```
** Example 'STIEX1': SETTIME
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
END-DEFINE
*
ST. SETTIME
WRITE 10X 'START TIME:' *TIME
*
READ (100) EMPLOY-VIEW BY NAME
END-READ
*
WRITE NOTITLE 10X 'END TIME:  ' *TIME
WRITE         10X 'ELAPSED TIME TO READ 100 RECORDS'
                  '(HH:MM:SS.T) :' *TIMD (ST.) (EM=99:99:99'.'9)
*
END
```

Output of Program STIEX1:

```
START TIME: 16:39:07.6
END TIME:   16:39:07.7
ELAPSED TIME TO READ 100 RECORDS (HH:MM:SS.T) : 00:00:00.1
```

131 SET WINDOW

```
SET WINDOW { 'window-name' }
           { OFF           }
```

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

Related Statements: `DEFINE WINDOW` | `INPUT WINDOW='window-name'` | `REINPUT`

Belongs to Function Group: ***Screen Generation for Interactive Processing***

Function

The `SET WINDOW` statement is used to activate and de-activate a window.

Any `SET WINDOW 'window-name'` or `INPUT WINDOW='window-name'` statement de-activates the window which has currently been active and activates the window specified in the statement. This means that only one window can be active at a time.

Note: If you use `SET WINDOW` to activate a window which is defined with `SIZE AUTO`, the data on the screen *before* the window is activated determine the size of the window.

Syntax Description

Syntax Element	Description
`SET WINDOW 'window-name'`	Activates the specified window, which means that all subsequent statements refer to that window until either the window is de-activated or another window is activated. The specified window must have been defined with a `DEFINE WINDOW` statement.
`SET WINDOW OFF`	De-activates the currently active window.

Example

See `DEFINE WINDOW` statement.

132 SKIP

```
SKIP [(rep)] operand1 [LINES]
```

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

Related Statements: AT END OF PAGE | AT TOP OF PAGE | CLOSE PRINTER | DEFINE PRINTER | DISPLAY | EJECT | FORMAT | NEWPAGE | PRINT | SUSPEND IDENTICAL SUPPRESS | WRITE | WRITE TITLE | WRITE TRAILER

Belongs to Function Group: *Creation of Output Reports*

Function

The SKIP statement is used to generate one or more blank lines in an output report.

See also *Page Titles, Page Breaks, Blank Lines* in the *Programming Guide*.

Processing

If the execution of a SKIP statement would cause the page size to be exceeded, exceeding lines will be ignored (except in an AT TOP OF PAGE statement).

A SKIP statement is only executed if something has already been output on the page (output from an AT TOP OF PAGE statement is not taken into account here).

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
operand1	C	S					N	P	I									yes	no

Syntax Element Description:

Syntax Element	Description
(*rep*)	**Report Specification:** The notation (*rep*) may be used to specify the identification of the report for which the SKIP statement is applicable. A value in the range 0 - 31 or a logical name which has been assigned using the DEFINE PRINTER statement may be specified.

Syntax Element	Description
	If `(rep)` is not specified, the `SKIP` statement will apply to the first report (Report 0). For information on how to control the format of an output report created with Natural, see *Report Format and Control* in the *Programming Guide*.
operand1	**Number of Lines to be Skipped:** *operand1* represents the number (`1 - 250`) of blank lines to be generated. This number may be specified as a numeric constant or as the content of a numerical variable. If *operand1* exceeds the page size of the report, the `SKIP` statement will result in a newpage condition.

Example

```
** Example 'SKPEX1': SKIP
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 COUNTRY
  2 NAME
END-DEFINE
*
LIMIT 7
READ EMPL-VIEW BY CITY STARTING FROM 'W'
  AT BREAK OF CITY
    SKIP 2
  END-BREAK
  DISPLAY NOTITLE CITY (IS=ON) COUNTRY (IS=ON) NAME
  /*
END-READ
END
```

Output of Program SKPEX1:

```

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


WASHINGTON           USA     REINSTEDT
                             PERRY


WEITERSTADT          D       BUNGERT
                             UNGER
                             DECKER
```

```
WEST BRIDGFORD       UK      ENTWHISTLE


WEST MIFFLIN         USA     WATSON
```

133 SORT

Structured Mode Syntax

```
END-ALL
[AND]
SORT      [ THEM        ] [BY] { operand1 [ ASCENDING  ] }... 10
          [ RECORDS     ]                 [ DESCENDING ]
            USING-clause
            [GIVE-clause]
            statement ...
END-SORT
```

* If a statement label is specified, it must be placed *before* the keyword SORT, but *after* END-ALL (and AND).

Reporting Mode Syntax

```
SORT [ THEM        ] [BY] { operand1 [ ASCENDING  ] }... 10
     [ RECORDS     ]                 [ DESCENDING ]
       [USING-clause]
       [GIVE-clause]
       statement ...
LOOP
```

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

Related Statement: **FIND with SORTED BY option**

Belongs to Function Group: ***Loop Execution***

Function

The SORT statement is used to perform a sort operation, sorting the records from all processing loops that are active when the SORT statement is executed.

Note: Natural creates a temporary work file during the sort operation. If you specify the TMPSORTUNIQ profile parameter (see the *Parameter Reference* documentation), Natural generates a unique name for the temporary sort work file.

Restrictions

- The SORT statement must be contained in the same object as the processing loops whose records it sorts.
- Nested SORT statements are not allowed.
- The total length of a record to be sorted must not exceed 10240 bytes.
- The number of sort criteria must not exceed 10.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1		S				A	N	P	I	F	B	D	T						no	no

Syntax Element Description:

Syntax Element	Description
END-ALL	**Closing All Currently Active Loops:** In structured mode, the SORT statement must be preceded by END-ALL, which serves to close all active processing loops. The SORT statement itself initiates a new processing loop, which must be closed with END-SORT. **Note:** For reporting mode: The SORT statement closes all active processing loops and initiates a new processing loop.
operand1	**Sort Criteria:** *operand1* represents the fields/variables to be used as the sort criteria. 1 to 10 database fields (descriptors and non-descriptors) and/or user-defined variables may be specified. A multiple-value field or a field contained within a periodic group may be used. A group or an array is not permitted. **Note:** A field specified in the SORT criteria is used for both, to put a value into the SORT record in the **selecting phase** (1st phase), and to receive the sorted value in the **processing phase** (3rd phase). Be aware, this may cause addressing errors, when indexed array fields are used which carry a correct index value in the selecting (1st) phase, but with an out-of-range value in the processing (3rd) phase. Therefore, indexed array fields should be used with caution, and better be replaced with non-indexed fields (scalar).
ASCENDING	**Sort Sequence:**

Syntax Element	Description
`DESCENDING`	The default sort sequence is ascending. If you wish the values to be sorted in descending sequence, specify `DESCENDING`. `ASCENDING`/`DESCENDING` may be specified for each sort field.
`USING`	**USING Clause:** See ***USING Clause*** below. **Note:** The note given under the description of *operand1* also applies to the `USING` clause.
`GIVE`	**GIVE Clause:** See ***GIVE Clause*** below.
`END-SORT` `LOOP`	**End of SORT Statement:** In structured mode, the Natural reserved word `END-SORT` must be used to end the `SORT` statement. In reporting mode, the Natural statement `LOOP` is used to end the `SORT` statement.

USING Clause

The `USING` clause indicates the fields which are to be written to intermediate sort storage. It is required in structured mode and optional in reporting mode. However, it is strongly recommended to also use it in reporting mode so as to reduce memory requirements.

```
{ USING operand2 ... }
{ USING KEYS         }
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
operand2		S	A			A	N	P	I	F	B	D	T	L	C			no	no

Syntax Element Description:

Syntax Element	Description
`USING` *operand2*	**Additional Fields:** You can specify additional fields that are to be written to the intermediate sort storage - in addition to the sort key fields (as specified with *operand1*).
`USING KEYS`	**Sort Key Fields Only:** Only the sort key fields, as specified with *operand1*, will be written to intermediate sort storage.

In Reporting Mode: If you omit the `USING` clause, all database fields of processing loops initiated before the `SORT` statement, as well as all user-defined variables defined before the `SORT` statement, will be written to intermediate sort storage.

If, after sort execution, a reference is made to a field which was not written to the sort intermediate storage, the value for the field will be the last value of the field before the sort.

GIVE Clause

The `GIVE` clause is used to specify Natural system functions (such as `MAX`, `MIN`) that are to be evaluated in the first phase of the `SORT` statement. These system functions may be referenced in the third phase (see ***SORT Statement Processing***).

A reference to a system function after the `SORT` statement must be preceded by an asterisk, for example, `*AVER(SALARY)`.

Note: In place of the keyword `GIVE`, the keyword `GIVING` may be used.

```
            MAX
            MIN
            NMIN
            COUNT
            NCOUNT                 (operand3 ...)
GIVE        OLD       ...   [OF]                      [(NL=nn)]   ...
            AVER                   operand3 ...
            NAVER
            SUM
            TOTAL
```

Operand Definition Table:

Operand	Possible Structure	Possible Formats	Referencing Permitted	Dynamic Definition
operand3	S A	*	yes	no

* depends on function

Syntax Element Description:

Syntax Element	Description
MAX | MIN | NMIN | COUNT | NCOUNT | OLD | AVER | NAVER | SUM | TOTAL	**System Functions:** For details on the individual system functions, see the *System Functions* documentation.
operand3	**Field Name:** *operand3* is the field name.
(NL=*nn.m*)	**Preventing Arithmetic Overflows:** This option applies to the system functions AVER, NAVER, SUM and TOTAL only. It will be ignored for any other system function. See also session parameter NL in the *Parameter Reference* documentation. This option may be used to prevent an arithmetic overflow during the evaluation of system functions; it is described under *Format/Length Requirements for AVER, NAVER, SUM and TOTAL* in the *System Functions* documentation.

Three-Phase SORT Processing

A program containing a SORT statement is executed in three phases.

1st Phase - Selecting the Records to be Sorted

The statements before the SORT statement are executed. Data as described in the USING clause will be written to intermediate sort storage.

In reporting mode, any variables to be used as accumulators following the sort must not be defined before the SORT statement. In structured mode, they must not be included in the USING clause. Fields written to intermediate sort storage cannot be used as accumulators because they are read back with each individual record during the 3rd processing phase. Consequently, the accumulation function would not produce the desired result because with each record the field would be over-written with the value for that individual record.

The number of records written to intermediate storage is determined by the number of processing loops and the number of records processed per loop. One record on the internal intermediate storage is created each time the SORT statement is encountered in a processing loop. In the case of nested loops, a record is only written to intermediate storage if the inner loop is executed. If in the example below a record is to be written to intermediate storage even if no records are found for the inner (FIND) loop, the FIND statement must contain an IF NO RECORDS FOUND clause.

```
READ ...
  ...
  FIND ...
...
END-ALL
SORT ...
  DISPLAY ...
END-SORT
...
```

2nd Phase - Sorting the Records

The records are sorted.

3rd Phase - Processing the Sorted Records

The statements after the `SORT` statement are executed for all records on the intermediate storage in the specified sorting sequence. Database fields to be referenced after a `SORT` statement must be correctly referenced using the appropriate statement label or reference number.

Example

- Example 1 - SORT
- Example 2 - SORT
- Example 3 - SORT

Example 1 - SORT

```
** Example 'SRTEX1S': SORT (structured mode)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 SALARY      (1:2)
  2 PERSONNEL-ID
  2 CURR-CODE   (1:2)
*
1 #AVG          (P11)
1 #TOTAL-TOTAL  (P11)
1 #TOTAL-SALARY (P11)
1 #AVER-PERCENT (N3.2)
END-DEFINE
*
LIMIT 3
FIND EMPL-VIEW WITH CITY = 'BOSTON'
  COMPUTE #TOTAL-SALARY = SALARY (1) + SALARY (2)
  ACCEPT IF #TOTAL-SALARY GT 0
```

```
  /*
END-ALL
AND
SORT BY PERSONNEL-ID USING #TOTAL-SALARY SALARY(*) CURR-CODE(1)
     GIVE AVER(#TOTAL-SALARY)
  /*
  AT START OF DATA
    WRITE NOTITLE '*' (40)
         'AVG CUMULATIVE SALARY:' *AVER (#TOTAL-SALARY) /
    MOVE *AVER (#TOTAL-SALARY) TO #AVG
  END-START
  COMPUTE ROUNDED #AVER-PERCENT = #TOTAL-SALARY / #AVG * 100
  ADD #TOTAL-SALARY TO #TOTAL-TOTAL
  /*
  DISPLAY NOTITLE PERSONNEL-ID SALARY (1) SALARY (2)
          #TOTAL-SALARY CURR-CODE (1)
          'PERCENT/OF/AVER' #AVER-PERCENT
  AT END OF DATA
    WRITE / '*' (40) 'TOTAL SALARIES PAID: ' #TOTAL-TOTAL
  END-ENDDATA
END-SORT
*
END
```

Output of Program SRTEX1S:

```
PERSONNEL   ANNUAL     ANNUAL   #TOTAL-SALARY CURRENCY PERCENT
    ID       SALARY     SALARY                   CODE     OF
                                                         AVER
  --------- ---------- ---------- ------------- -------- -------

  **************************************** AVG CUMULATIVE SALARY:        41900

  20007000       16000      15200        31200  USD       74.00
  20019200       18000      17100        35100  USD       83.00
  20020000       30500      28900        59400  USD      141.00

  **************************************** TOTAL SALARIES PAID:       125700
```

The previous example is executed as follows:

First Phase:

- Records with `CITY=BOSTON` are selected from the `EMPLOYEES` file.
- The first 2 occurrences of `SALARY` are accumulated in the field `#TOTAL-SALARY`.
- Only records with `#TOTAL-SALARY` greater than 0 are accepted.
- The records are written to the sort intermediate storage. The database arrays `SALARY` (first 2 occurrences) and `CURR-CODE` (first occurrence), the database field `PERSONNEL-ID`, and the user-defined variable `#TOTAL-SALARY` are written to the intermediate storage.

- The average of #TOTAL-SALARY is evaluated.

Second Phase:

- The records are sorted.

Third Phase:

- The sorted intermediate storage is read.
- At the at-start-of-data condition, the average of #TOTAL-SALARY is displayed.
- #TOTAL-SALARY is added to #TOTAL-TOTAL and the fields PERSONNEL-ID, SALARY(1), SALARY(2), #AVER-PERCENT and #TOTAL-SALARY are displayed.
- At the end-of-data condition, the variable #TOTAL-TOTAL is written.

Equivalent reporting-mode example: **SRTEX1R**.

Example 2 - SORT

```
** Example 'SRTEX2': SORT
************************************************************************
DEFINE DATA LOCAL
1 VEHIC-VIEW VIEW OF VEHICLES
  2 MAKE
  2 YEAR
END-DEFINE
*
LIMIT 10
*
READ VEHIC-VIEW
END-ALL
SORT BY MAKE YEAR USING KEY
  DISPLAY NOTITLE (AL=15) MAKE (IS=ON) YEAR
  AT BREAK OF MAKE
    WRITE '-' (20)
  END-BREAK
END-SORT
END
```

Output of Program SRTEX2S:

```
     MAKE       YEAR
--------------- -----

FIAT             1980
                 1982
                 1984
--------------------
PEUGEOT          1980
```

```
                 1982
                 1985
--------------------
RENAULT          1980
                 1980
                 1982
                 1982
--------------------
```

Example 3 - SORT

```
** Example 'SRTEX3': SORT values in an array
************************************************************************
DEFINE DATA LOCAL
1 #I   (I4)
1 #J   (I4)
1 #X   (I1)
1 #TAB (I1/1:6) INIT <2,4,6,5,3,1>
END-DEFINE
WRITE   'Array before SORT:' #TAB(*) /
*
FOR #I := 1 TO 6
  #X := #TAB(#I)
  WRITE #X '<-- Put into SORT record'
END-ALL
SORT #X USING KEYS
  WRITE #X '<-- Get from SORT'
  ADD 1 TO #J
  #TAB(#J) := #X
END-SORT
*
WRITE / 'Array after  SORT:' #TAB(*)
END
```

Output of Program SRTEX3:

```
Array before SORT:    2    4    6    5    3    1

   2 <-- Put into SORT record
   4 <-- Put into SORT record
   6 <-- Put into SORT record
   5 <-- Put into SORT record
   3 <-- Put into SORT record
   1 <-- Put into SORT record
   1 <-- Get from SORT
   2 <-- Get from SORT
   3 <-- Get from SORT
   4 <-- Get from SORT
   5 <-- Get from SORT
   6 <-- Get from SORT
```

```
Array after  SORT:    1    2    3    4    5    6
```

Using External Sort Programs

In Natural, sort operations are by default processed by Natural's internal sort program, as described above. However, an external sort program can be used. This external sort program then processes the sort operations instead of Natural's internal sort program.

As external sort programs DMExpress Syncsort for Linux and IRI CoSort for Linux are supported.

Whether an external sort program is used or not, can be determined while you install Natural. For further information, see *Re-Linking a Natural Nucleus* in the *Installation* documentation.

The records that are to be sorted will be temporarily stored in the directory specified under `TMP_PATH` in the *Installation Assignments* of your *Local Configuration File*.

134 STACK

STACK [TOP] { COMMAND *operand1* [*operand2* [(*parameter*)]] ...   [DATA] [FORMATTED] {*operand2* [(*parameter*)]} ... }

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

Related Statements: INPUT | RELEASE

Function

The STACK statement is used to place any of the following into the Natural stack:

- the name of a Natural program or Natural system command to be executed;
- data to be used during the execution of an INPUT statement.

For further information on the stack, see *Further Programming Aspects*, *Stack Processing* in the *Programming Guide*.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A	G	N	A													yes	yes
operand2	C	S	A	G	N	A	U	N	P	I	F	B	D	T	L		G		yes	yes

Syntax Element Description:

Syntax Element	Description
TOP	**TOP Option:** If you specify TOP, the data/program/command will be placed at the top of the Natural stack. Otherwise, they are placed at the bottom of the stack. Example: The following statement causes the content of the variable #FIELDA to be placed as data on top of the stack:

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td></td><td><pre>STACK TOP #FIELDA</pre></td></tr>
<tr><td>DATA</td><td>DATA Option:

This option, which is also the default, causes data to be placed in the stack which are to be used as input data for an INPUT statement.

Delimiter characters or input assign characters contained within the data values will be processed as delimiters. For details on how data from the stack are processed by an INPUT statement, refer to Processing Data from the Natural Stack (in the description of the INPUT statement).

Example: The following statements cause the contents of the variables #FIELD1 and #FIELD2 to be placed in the stack:
<pre>MOVE 'ABC' TO #FIELD1
MOVE 'XYZ' TO #FIELD2
STACK #FIELD1 #FIELD2</pre>These variables will be passed as data to the next INPUT statement in the Natural program, using delimiter mode:
<pre>INPUT #FIELD1 #FIELD2</pre>Note: If operand2 is a time variable (Format T), only the time component of the variable content is placed in the stack, but not the date component.</td></tr>
<tr><td>FORMATTED</td><td>FORMATTED Option:

This option causes all data to be passed on a field-by-field basis to the next INPUT statement; no key assignments or delimiter characters will be interpreted.

Examples:

The following statements cause ABC,DEF to be placed in #FIELD1 and XYZ in #FIELD2:
<pre>MOVE 'ABC,DEF' TO #FIELD1
MOVE 'XYZ'   TO #FIELD2
STACK TOP DATA FORMATTED #FIELD1 #FIELD2
...
INPUT #FIELD1 #FIELD2</pre>Assuming the input delimiter character to be the comma (profile/session parameter ID=,), the following statements - without the keyword FORMATTED - cause ABC to be placed in #FIELD1 and DEF in #FIELD2:</td></tr>
</table>

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td></td><td>
<pre>
MOVE 'ABC,DEF' TO #FIELD1
STACK TOP DATA #FIELD1
...
INPUT #FIELD1 #FIELD2
</pre>
Note: The FORMATTED option should be used if the data to be passed contains delimiter, control or DBCS characters to avoid unintentional interpretation of these characters.
</td></tr>
<tr><td>COMMAND
operand1</td><td>
COMMAND Option:
To place a command (or program name) in the stack, you specify the keyword COMMAND followed by the command specified in operand1. Natural will execute the command instead of displaying the NEXT prompt and prompting the user for input.
Example:
The following statement causes the command RUN to be placed at the top of the stack. Natural will execute this command at the point where the NEXT prompt would normally be issued.
<pre>
STACK TOP COMMAND 'RUN'
</pre>
</td></tr>
<tr><td>COMMAND
operand1
operand2 ...</td><td>
COMMAND with Data Option:
Together with a command (operand1), you may also place data (operand2) in the stack. These data will then be processed by the next INPUT statement after the command has been executed.
Data stacked with a command are always stacked unformatted.
Note: If the data to be stacked include empty alphanumeric fields (that is, blanks), these blanks will be interpreted as delimiters between values and thus not processed correctly by the corresponding INPUT statement. Therefore, if you wish to stack empty alphanumeric fields as data with a command, you have to use two STACK statements: one STACK DATA operand2 ... to stack the data, and one STACK COMMAND operand1 to stack the command.
</td></tr>
<tr><td>parameter</td><td>
Date Format:
If operand2 is a date variable, you can specify the session parameter DF as a parameter for this variable.
</td></tr>
</table>

Example

```
** Example 'STKEX1': STACK
************************************************************************
DEFINE DATA LOCAL
1 #CODE (A1)
END-DEFINE
*
INPUT //
  10X 'PLEASE SELECT COMMAND' //
  10X 'LIST VIEW      (V)' /
  10X 'LIST PROGRAM * (P)' /
  10X 'TECH INFO      (T)' /
  10X 'STOP           (.)' //
  20X 'CODE:' #CODE
*
*
DECIDE ON FIRST #CODE
  VALUE 'V'
    STACK TOP DATA    'VIEW'
    STACK TOP COMMAND 'LIST'
  VALUE 'P'
    STACK TOP COMMAND 'LIST PROGRAM *'
  VALUE 'T'
    STACK TOP COMMAND 'LAST *'
    STACK TOP COMMAND 'TECH'
    STACK TOP COMMAND 'SYSPROD'
  VALUE '.'
    STOP
  NONE
    REINPUT 'PLEASE ENTER VALID CODE'
END-DECIDE
*
*
END
```

Output of Program STKEX1:

```
PLEASE SELECT COMMAND

LIST VIEW      (V)
LIST PROGRAM * (P)
TECH INFO      (T)
STOP           (.)

          CODE:P
```

After entering and confirming code:

```
16:46:28              ***** NATURAL LIST COMMAND *****              2005-01-19
User HTR                - LIST Objects in a Library -        Library SYSEXSYN

Cmd  Name       Type         S/C  SM Version  User ID    Date        Time
---  *________  P__________  *__  *  *______  *________  *__________ *________
__   ACREX1     Program      S/C  S  4.1.03   RKE        2004-11-11  16:32:37
__   ACREX2     Program      S/C  S  4.1.03   RKE        2005-01-05  10:29:51
__   ADDEX1     Program      S/C  S  4.1.03   RKE        2004-11-11  16:36:49
__   AEDEX1R    Program      S/C  R  4.1.03   RKE        2004-11-11  16:40:34
__   AEDEX1S    Program      S/C  S  4.1.03   RKE        2004-11-11  16:39:57
__   AEPEX1R    Program      S/C  R  4.1.03   RKE        2004-11-11  16:41:57
__   AEPEX1S    Program      S/C  S  4.1.03   RKE        2004-11-11  16:42:31
__   AEPEX2     Program      S/C  S  4.1.03   RKE        2004-11-11  16:43:37
__   ASDEX1R    Program      S/C  R  4.1.03   RKE        2004-11-11  17:00:21
__   ASDEX1S    Program      S/C  S  4.1.03   RKE        2004-11-11  17:00:50
__   ASGEX1R    Program      S/C  R  4.1.03   RKE        2004-11-11  17:02:01
__   ASGEX1S    Program      S/C  S  4.1.03   RKE        2004-11-11  17:02:08
__   ATBEX1R    Program      S/C  R  4.1.03   RKE        2004-11-11  17:03:18
__   ATBEX1S    Program      S/C  S  4.1.03   RKE        2004-11-11  17:03:05
                                                              14 Objects found
Top of List.
Command ===>
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
      Help  Print Exit  Sort        --    -     +     ++          >     Canc
```

135 STOP

```
STOP
```

Function

The STOP statement is used to terminate the execution of a program and return to the command input prompt.

One or more STOP statements may be inserted anywhere within a Natural program.

The STOP statement will terminate the execution of the program immediately. Independent of the positioning of a STOP statement in a subroutine, any end-page condition specified in the main program will be invoked for final end-page processing during execution of the STOP statement.

The STOP statement behaves in the same way as the ESCAPE ROUTINE statement during method execution. Method execution is terminated immediately without producing any return vale.

For Natural RPC: See *Notes on Natural Statements on the Server* in the *Natural RPC (Remote Procedure Call)* documentation.

Example

```
** Example 'STPEX1': STOP
************************************************************************
DEFINE DATA LOCAL
1 #CODE (A1)
END-DEFINE
*
INPUT //
  10X 'PLEASE SELECT COMMAND' //
  10X 'LIST VIEW      (V)' /
  10X 'LIST PROGRAM * (P)' /
  10X 'TECH INFO      (T)' /
  10X 'STOP           (.)' //
  20X 'CODE:' #CODE
*
*
DECIDE ON FIRST #CODE
  VALUE 'V'
    STACK TOP DATA    'VIEW'
    STACK TOP COMMAND 'LIST'
  VALUE 'P'
    STACK TOP COMMAND 'LIST PROGRAM *'
  VALUE 'T'
    STACK TOP COMMAND 'LAST *'
    STACK TOP COMMAND 'TECH'
```

```
    STACK TOP COMMAND 'SYSPROD'
  VALUE '.'
    STOP
  NONE
    REINPUT 'PLEASE ENTER VALID CODE'
END-DECIDE
*
*
END
```

Output of Program STPEX1:

```
PLEASE SELECT COMMAND

LIST VIEW      (V)
LIST PROGRAM * (P)
TECH INFO      (T)
STOP           (.)

          CODE:
```

XV

136 STORE

Structured Mode Syntax

```
STORE   [RECORD] [IN] [FILE] view-name
        [PASSWORD=operand1]
        [CIPHER=operand2]
        [  [  USING   ]  NUMBER operand3  ]
              GIVING
```

Reporting Mode Syntax

```
STORE   [RECORD] [IN] [FILE] view-name
        [PASSWORD=operand1]
        [CIPHER=operand2]
        [  [  USING   ]      NUMBER operand3      ]
              GIVING
        {  [USING] SAME [RECORD] [AS] [STATEMENT [(r)]]  }
        {  [  SET   ]        [operand4=operand5]         }
              WITH           ...
```

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

Related Statements: ACCEPT/REJECT | AT BREAK | AT START OF DATA | AT END OF DATA | BACKOUT TRANSACTION | BEFORE BREAK PROCESSING | DELETE | END TRANSACTION | FIND | GET | GET SAME | GET TRANSACTION DATA | HISTOGRAM | LIMIT | PASSW | PERFORM BREAK PROCESSING | READ | RETRY | UPDATE |

Belongs to Function Group: ***Database Access and Update***

Function

The STORE statement is used to add a record to a database.

Database-Specific Considerations

Adabas	The Natural system variable `*ISN` contains the Adabas ISN assigned to the new record as a result of the `STORE` statement execution. A subsequent reference to `*ISN` must include the statement number of the related `STORE` statement.
SQL	This statement may be used to add a row to a table. The `PASSWORD`, `CIPHER`, and `GIVING NUMBER` clauses cannot be used. The `STORE` statement corresponds with the SQL statement `INSERT`. The Natural system variable `*ISN` is not available.
XML	This statement may be used to add an XML object to a database. The `PASSWORD`, `CIPHER`, and `GIVING NUMBER` clauses cannot be used. For Tamino, the Natural system variable `*ISN` contains the XML object ID assigned to the new record as a result of the `STORE` statement execution. A subsequent reference to `*ISN` must include the statement number of the related `STORE` statement.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S				A													yes	no
operand2	C	S						N											yes	no
operand3	C	S						N	P			B *							no	yes
operand4		S	A			A	U	N	P	I	F	B	D	T	L				no	no
operand5	C	S	A			A	U	N	P	I	F	B	D	T	L				yes	no

* Format B of *operand3* may be used only with a length of less than or equal to 4.

Syntax Element Description:

Syntax Element	Description
view-name	**View Name:** As *view-name*, you specify the name of a view, which must have been defined either in a `DEFINE DATA` statement or outside the program in a global or local data area. In reporting mode, *view-name* is the name of a DDM if no `DEFINE DATA LOCAL` statement is used.

Syntax Element	Description
PASSWORD=*operand1*	**PASSWORD Clause:** The PASSWORD clause is applicable only for an Adabas database. This clause is used to provide a password (*operand1*) when updating data from a file which is password-protected. The password (*operand1*) may be specified as an alphanumeric constant or as an alphanumeric variable. It may consist of up to 8 characters, and must not contain special characters or embedded blanks. If the password is specified as a constant, it must be enclosed in apostrophes. For further information, see the statements FIND and PASSW.
CIPHER=*operand2*	**CIPHER Clause:** The CIPHER clause is applicable only for an Adabas database. This clause is used to provide a cipher key (*operand2*) when updating data from a file which is enciphered. The cipher key (*operand2*) may be specified as an numeric constant with 8 digits or as a user-defined variable with format/length N8. For further information, see the statement FIND.
USING NUMBER *operand3*	**USING NUMBER Clause:** This clause can only be used for an Adabas database.
GIVING NUMBER *operand3*	**GIVING NUMBER Clause:** This clause is used to store a record with a user-supplied Adabas ISN (range from 1 to 4294967295). If a record with the specified ISN already exists, an error message will be returned, and the execution of the program will be terminated unless ON ERROR processing was specified.
SET/WITH *operand4*=*operand5*	**SET/WITH Clause:** SET/WITH can be used in reporting mode to specify the fields for which values are being provided. Any field defined in the file that is not specified in the SET clause will contain a null value in the new record. This clause is not permitted if a DEFINE DATA statement is used, because in that case the STORE statement always refers to the entire view as defined in the DEFINE DATA statement.
USING SAME (*r*)	**USING SAME Clause:** In reporting mode, this clause can be used to indicate that the same field values as read in the statement referenced by the STORE statement (FIND, GET, READ) are to be used to add a new record. The statement reference notation (*r*) may be specified as a source-code line number or as a statement label.

Syntax Element	Description
	This clause is not permitted if a DEFINE DATA statement is used, because in that case the STORE statement would always refers to the entire view, as defined in the DEFINE DATA statement.

Example

```
** Example 'STOEX1S': STORE  (structured mode)
**
** CAUTION: Executing this example will modify the database records!
***********************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
  2 MAR-STAT
  2 BIRTH
  2 CITY
  2 COUNTRY
*
1 #PERSONNEL-ID (A8)
1 #NAME         (A20)
1 #FIRST-NAME   (A15)
1 #BIRTH-D      (D)
1 #MAR-STAT     (A1)
1 #BIRTH        (A8)
1 #CITY         (A20)
1 #COUNTRY      (A3)
1 #CONF         (A1)
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
/*
FIND NUMBER EMPL-VIEW WITH PERSONNEL-ID =  #PERSONNEL-ID
IF *NUMBER > 0
  REINPUT 'PERSON WITH SAME PERSONNEL-ID ALREADY EXISTS'
           MARK 1 AND SOUND ALARM
END-IF
MOVE 'N' TO #CONF
/*
/*  GET FURTHER INFORMATION
/*
INPUT
  'ADDITIONAL PERSONNEL DATA'                        ////
  'PERSONNEL-ID             :' #PERSONNEL-ID (AD=IO) /
  'NAME                     :' #NAME         (AD=IO) /
  'FIRST-NAME               :' #FIRST-NAME   (AD=IO) ///
  'MARITAL STATUS           :' #MAR-STAT             /
  'DATE OF BIRTH (YYYYMMDD) :' #BIRTH                /
  'CITY                     :' #CITY                 /
  'COUNTRY (3 CHARACTERS)   :' #COUNTRY              //
  'ADD THIS RECORD (Y/N)    :' #CONF          (AD=M)
/*
/*   ENSURE REQUIRED FIELDS CONTAIN VALID DATA
/*
IF NOT (#MAR-STAT = 'S'  OR = 'M' OR = 'D' OR = 'W')
  REINPUT TEXT 'ENTER VALID MARITAL STATUS  S=SINGLE ' -
               'M=MARRIED D=DIVORCED W=WIDOWED' MARK 1
END-IF
IF NOT (#BIRTH = MASK(YYYYMMDD) AND #BIRTH = MASK(1582-2699))
  REINPUT TEXT 'ENTER CORRECT DATE' MARK 2
END-IF
IF #CITY  = ' '
  REINPUT TEXT 'ENTER A CITY NAME' MARK 3
END-IF
IF #COUNTRY = ' '
  REINPUT TEXT 'ENTER A COUNTRY CODE' MARK 4
END-IF
IF NOT (#CONF = 'N' OR= 'Y')
  REINPUT TEXT 'ENTER Y (YES) OR N (NO)' MARK 5
END-IF
IF #CONF = 'N'
  ESCAPE TOP
END-IF
/*
/*  ADD THE RECORD
/*
MOVE EDITED #BIRTH TO #BIRTH-D (EM=YYYYMMDD)
/*
EMPL-VIEW.PERSONNEL-ID := #PERSONNEL-ID
EMPL-VIEW.NAME         := #NAME
```

```
  EMPL-VIEW.FIRST-NAME    := #FIRST-NAME
  EMPL-VIEW.MAR-STAT      := #MAR-STAT
  EMPL-VIEW.BIRTH         := #BIRTH-D
  EMPL-VIEW.CITY          := #CITY
  EMPL-VIEW.COUNTRY       := #COUNTRY
  /*
  STORE RECORD IN EMPL-VIEW
  /*
  END OF TRANSACTION
  /*
  WRITE NOTITLE 'RECORD HAS BEEN ADDED'
  /*
END-REPEAT
END
```

Output of Program STOEX1S:

```
ENTER A PERSONNEL ID AND NAME (OR 'END' TO END)

PERSONNEL-ID : 90001100

NAME         : JONES
FIRST-NAME   : EDWARD
```

After entering and confirming the personnel key data, additional personnel data fields are displayed for input:

```
ADDITIONAL PERSONNEL DATA



PERSONNEL-ID             : 90001100
NAME                     : JONES
FIRST-NAME               : EDWARD


MARITAL STATUS           :
DATE OF BIRTH (YYYYMMDD) :
CITY                     :
COUNTRY (3 CHARACTERS)   :

ADD THIS RECORD (Y/N)    : N
```

Equivalent reporting-mode example: STOEX1R.

137 SUBTRACT

Related Statements: ADD | COMPRESS | COMPUTE | DIVIDE | EXAMINE | MOVE | MOVE ALL | MULTIPLY | RESET | SEPARATE

Belongs to Function Group: *Arithmetic and Data Movement Operations*

Function

The SUBTRACT statement is used to subtract one or more arithmetic expressions or operands from another operand.

Syntax 1 - SUBTRACT Statement without GIVING Clause

```
SUBTRACT [ROUNDED] { (arithmetic-expression) } ... FROM operand2
                   { operand1                }
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
operand1	C	S	A		N		N	P	I	F		D	T					yes	no
operand2		S	A		M		N	P	I	F		D	T					yes	no

Syntax Element Description:

Syntax Element	Description
arithmetic-expression	See ***Arithmetic Expression*** in the COMPUTE statement.
operand1 FROM *operand2*	**Operands:** *operand2* is the minuend, *operand1* is the subtrahend, hence the statement is equivalent to: *operand2 := operand2 - operand1* As for the formats of the operands, see also *Rules for Arithmetic Assignments*, *Performance Considerations for Mixed Formats* in the *Programming Guide*.
ROUNDED	**ROUNDED Option:** If you specify the keyword ROUNDED, the result will be rounded. For information on rounding, see *Rules for Arithmetic Assignment*, *Field Truncation and Field Rounding* in the *Programming Guide*.

Syntax 2 - SUBTRACT Statement with GIVING Clause

```
SUBTRACT   { (arithmetic-expression) } ... FROM { (arithmetic-expression) } GIVING
[ROUNDED]  { operand1                }          { operand2                } operand3
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A		N			N	P	I	F		D	T					yes	no
operand2	C	S	A		N			N	P	I	F		D	T					yes	no
operand3		S	A		M	A	U	N	P	I	F	B*	D	T					yes	yes

* Format B of *operand3* may be used only with a length of less than or equal to 4.

Syntax Element Description:

Syntax Element	Description
arithmetic-expression	See ***Arithmetic Expression*** in the COMPUTE statement.
GIVING	**GIVING Clause:** When the GIVING clause is used, *operand2* will *not* be modified, and the result will be stored in *operand3*.
operand1 FROM *operand2* GIVING *operand3*	**Operands:** *operand2* is the minuend, *operand1* is the subtrahend, *operand3* is the result field, hence the statement is equivalent to: *operand3* := *operand2* - *operand1* As for the formats of the operands, see also the section *Performance Considerations for Mixed Formats* in the *Programming Guide*.
ROUNDED	**ROUNDED Option:** If you specify the keyword ROUNDED, the result will be rounded. For information on rounding, see *Rules for Arithmetic Assignment, Field Truncation and Field Rounding* in the *Programming Guide*.

Example

```
** Example 'SUBEX1': SUBTRACT
************************************************************************
DEFINE DATA LOCAL
1 #A (P2) INIT <50>
1 #B (P2)
1 #C (P1.1) INIT <2.4>
END-DEFINE
*
SUBTRACT 6 FROM #A
WRITE NOTITLE 'SUBTRACT 6 FROM #A              '  10X '=' #A
*
SUBTRACT 6 FROM 11 GIVING #A
WRITE         'SUBTRACT 6 FROM 11 GIVING #A    '  10X '=' #A
*
SUBTRACT 3 4 FROM #A GIVING #B
WRITE         'SUBTRACT 3 4 FROM #A GIVING #B  '  10X '=' #A '=' #B
*
SUBTRACT -3 -4 FROM #A GIVING #B
WRITE         'SUBTRACT -3 -4 FROM #A GIVING #B'  10X '=' #A '=' #B
*
SUBTRACT ROUNDED 2.06 FROM #C
WRITE         'SUBTRACT ROUNDED 2.06 FROM #C   '  10X '=' #C
*
END
```

Output of Program SUBEX1:

```
SUBTRACT 6 FROM #A                        #A:  44
SUBTRACT 6 FROM 11 GIVING #A              #A:   5
SUBTRACT 3 4 FROM #A GIVING #B            #A:   5 #B:  -2
SUBTRACT -3 -4 FROM #A GIVING #B          #A:   5 #B:  12
SUBTRACT ROUNDED 2.06 FROM #C             #C:  0.3
```

138 SUSPEND IDENTICAL SUPPRESS

```
SUSPEND IDENTICAL [SUPPRESS] [(rep)]
```

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

Related Statements: AT END OF PAGE | AT TOP OF PAGE | CLOSE PRINTER | DEFINE PRINTER | DISPLAY | EJECT | FORMAT | NEWPAGE | PRINT | SKIP | WRITE | WRITE TITLE | WRITE TRAILER

Belongs to Function Group: ***Creation of Output Reports***

Function

The SUSPEND IDENTICAL SUPPRESS statement is used to suspend the Natural session parameter setting IS=ON (which suppresses the output of identical field values) for the processing of one record.

See also session parameter IS in the *Parameter Reference*.

Syntax Description

Syntax Element	Description
(*rep*)	**Report Specification:** The notation (*rep*) may be used to specify the identification of the report for which the SUSPEND IDENTICAL SUPPRESS statement is applicable. A value in the range 0 - 31 or a logical name which has been assigned using the DEFINE PRINTER statement may be specified. If (*rep*) is not specified, the SUSPEND IDENTICAL SUPPRESS statement will be applicable to the first report (Report 0). For information on how to control the format of an output report created with Natural, see *Report Format and Control* in the *Programming Guide*.

Examples

- Example 1 - Program with SUSPEND IDENTICAL SUPPRESS

- Example 2 - Same as Previous Program, but without SUSPEND IDENTICAL SUPPRESS

Example 1 - Program with SUSPEND IDENTICAL SUPPRESS

```
** Example 'SISEX1': SUSPEND IDENTICAL SUPPRESS
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 CITY
1 VEH-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 15
*
RD. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  /*
  SUSPEND IDENTICAL SUPPRESS
  /*
  FD. FIND VEH-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      MOVE '***NO CAR***' TO MAKE
    END-NOREC
    DISPLAY NOTITLE
            NAME (RD.) (IS=ON)
            FIRST-NAME (RD.) (IS=ON)
            MAKE (FD.)
  END-FIND
  /*
END-READ
END
```

Output of Program SISEX1:

```
        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
JONES                    EDWARD                   ***NO CAR***
JOPER                    MANFRED                  ***NO CAR***
JOUSSELIN                DANIEL                   RENAULT
JUBE                     GABRIEL                  ***NO CAR***
JUNG                     ERNST                    ***NO CAR***
JUNKIN                   JEREMY                   ***NO CAR***
```

Example 2 - Same as Previous Program, but without SUSPEND IDENTICAL SUPPRESS

```
** Example 'SISEX2': SUSPEND IDENTICAL SUPPRESS (compare with SISEX1)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 CITY
1 VEH-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 15
RD. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  /*
  /* SUSPEND IDENTICAL SUPPRESS    /* statement removed
  /*
  FD. FIND VEH-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      MOVE '***NO CAR***' TO MAKE
    END-NOREC
    DISPLAY NOTITLE
            NAME (RD.) (IS=ON)
            FIRST-NAME (RD.) (IS=ON)
            MAKE (FD.)
  END-FIND
  /*
END-READ
END
```

Output of Program SISEX2:

```
        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
                     EDWARD               ***NO CAR***
JOPER                MANFRED              ***NO CAR***
JOUSSELIN            DANIEL               RENAULT
JUBE                 GABRIEL              ***NO CAR***
JUNG                 ERNST                ***NO CAR***
JUNKIN               JEREMY               ***NO CAR***
```

139 TERMINATE

```
TERMINATE [operand1 [operand2]]
```

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

Function

The `TERMINATE` statement is used to terminate a Natural session. A `TERMINATE` statement may be placed anywhere within a Natural program. When a `TERMINATE` statement is executed, no end-of-page or end-loop processing will be performed.

For Natural RPC: See *Notes on Natural Statements on the Server* in the *Natural RPC (Remote Procedure Call)* documentation.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
operand1	C	S					N	P	I									yes	no
operand2	C	S	A			A	U											yes	yes

Syntax Element Description:

Syntax Element	Description
operand1	*operand1* may be used to pass a return code to the program receiving control when Natural terminates. For example, a return code setting may be passed as exit code to the shell. See also *Natural Startup Errors* in the *Operations* documentation. The value supplied for *operand1* must be in the range `0 - 255`.
operand2	*operand2* may be used to pass additional information to the program which receives control after the termination.

Program Receiving Control after Termination

After the termination of the Natural session, the program whose name is specified with the profile parameter `PROGRAM` will receive control.

Natural passes *operand2* and the value of the profile parameter `PRGPAR` to that program, if they are specified. The program receives these parameters in the usual way as arguments:

```
int main(int argc, char *argv[])
{
  /* Number of arguments passed. */
  printf("Number of arguments: %d\n", argc);
  /* Program name. */
  if ( argc > 0 )
    printf("Program: %s\n", argv[0]);
  /* Value of operand2 of the TERMINATE statement. */
  if ( argc > 1 )
    printf("Operand 2: %s\n", argv[1]);
  /* Value of the profile parameter PRGPAR. */
  if ( argc > 2 )
    printf("PRGPAR: %s\n", argv[2]);
  return 0;
}
```

If the `PROGRAM` parameter is not set, the Linux command shell will receive control after the termination.

Example

```
** Example 'TEREX1': TERMINATE
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 SALARY (1)
*
1 #PNUM     (A8)
1 #PASSWORD (A8)
END-DEFINE
*
INPUT 'ENTER PASSWORD:' #PASSWORD
*
IF #PASSWORD NE 'USERPASS'
  /*
```

```
  TERMINATE
  /*
END-IF
*
INPUT 'ENTER PERSONNEL NUMBER:' #PNUM
*
FIND EMPLOY-VIEW WITH PERSONNEL-ID = #PNUM
  DISPLAY NAME SALARY (1)
END-FIND
*
END
```

140 UPDATE

Structured Mode Syntax

```
UPDATE [RECORD] [IN] [STATEMENT] [(r)]
```

Reporting Mode Syntax

```
UPDATE     [RECORD] [IN] [STATEMENT] [(r)]

           [     SET     ]   { SAME [RECORD]                 }
           [     WITH    ]   {                               }
           [     USING   ]   { {operand1=operand2} ...       }
```

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

Related Statements: ACCEPT/REJECT | AT BREAK | AT START OF DATA | AT END OF DATA | BACKOUT TRANSACTION | BEFORE BREAK PROCESSING | DELETE | END TRANSACTION | FIND | GET | GET SAME | GET TRANSACTION DATA | HISTOGRAM | LIMIT | PASSW | PERFORM BREAK PROCESSING | READ | RETRY | STORE

Belongs to Function Group: ***Database Access and Update***

Function

The UPDATE statement is used to update one or more fields of a record in a database. The record to be updated must have been previously selected with a FIND, GET or READ statement (or, for Adabas only, with a STORE statement).

Hold Status

The use of the UPDATE statement causes each record read for processing in the corresponding FIND or READ statement to be placed in exclusive hold.

For further information, see *Record Hold Logic* (in the *Programming Guide*).

Restrictions

The `UPDATE` statement

- must not be entered on the same line as the statement used to select the record to be updated;
- cannot be applied to Entire System Server views.

Database-Specific Considerations

SQL	The `UPDATE` statement can be used to update a row in a database table. It corresponds with the SQL statement `UPDATE WHERE CURRENT OF CURSOR` (**Positioned** `UPDATE`), which means that only the row which was read last can be updated. With most SQL databases, a row that was read with a `FIND SORTED BY` or with a `READ LOGICAL` statement cannot be updated.
XML	The statement cannot be used with XML databases.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1		S	A			A		N	P	I	F	B	D	T	L				no	no
operand2	C	S	A			A		N	P	I	F	B	D	T	L				yes	no

Syntax Element Description:

Syntax Element	Description
`(r)`	**Statement Reference:** The notation `(r)` is used to indicate the statement in which the record to be modified was read. *r* may be specified as a source-code line number or as a statement label. If no reference is specified, the `UPDATE` statement will reference the innermost active `READ` or `FIND` processing loop. If no `READ` or `FIND` loop is active, it will reference the last preceding `GET` (or `STORE`) statement.

Syntax Element	Description
	Note: The UPDATE statement must be placed within the READ or FIND loop it references.
USING SAME	**USING SAME Clause:** This clause is not permitted if a DEFINE DATA statement is used, because in that case the UPDATE statement always refers to the entire view as defined in the DEFINE DATA statement. The layout of the record buffer or format buffer may be declared using the OBTAIN statement. USING SAME can be used in reporting mode to indicate that the same fields as read in the statement referenced by the UPDATE statement are to be used for the update function. In this case, the most recent value assigned to each database field will be used to update the field. If no new value has been assigned, the old value will be used. If the field to be updated is an array range of a multiple-value field or periodic group and you use a variable index for this array range, the latest range will be updated. This means that if the index variable is modified after the record has been read and before the UPDATE USING SAME (reporting mode) or UPDATE (structured mode) statement respectively is executed, the range updated will not be the same as the range read.
SET/WITH *operand1=operand2*	**SET/WITH Clause:** This clause can be used in reporting mode to specify the fields to be updated and the values to be used. This clause is not permitted if a DEFINE DATA statement is used, because in that case the UPDATE statement always refers to the entire view as defined in the DEFINE DATA statement.

Example

```
** Example 'UPDEX1S': UPDATE (structured mode)
**
** CAUTION: Executing this example will modify the database records!
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
*
1 #NAME (A20)
END-DEFINE
```

```
*
INPUT 'ENTER A NAME:' #NAME (AD=M)
IF #NAME = ' '
  STOP
END-IF
*
FIND EMPLOY-VIEW WITH NAME = #NAME
  IF NO RECORDS FOUND
    REINPUT WITH 'NO RECORDS FOUND'  MARK 1
  END-NOREC
  INPUT 'NAME:       ' NAME (AD=O) /
        'FIRST NAME:' FIRST-NAME (AD=M) /
        'CITY:       ' CITY (AD=M)
  UPDATE
  END TRANSACTION
END-FIND
*
END
```

Output of Program SUBEX1S

```
ENTER A NAME: BROWN
```

After entering and confirming name:

```
NAME: BROWN
FIRST NAME: KENNETH
CITY: DERBY
```

Equivalent reporting-mode example: `UPDEX1R`.

141 UPDATE (SQL)

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

Belongs to Function Group: *Database Access and Update*

Function

The SQL `UPDATE` statement is used to perform an `UPDATE` operation on either rows in a table without using a cursor (**"searched"** `UPDATE`) or columns in a row to which a cursor is positioned (**"positioned"** `UPDATE`).

Two different syntax structures are possible.

Syntax 1 - Searched UPDATE

The "Searched" `UPDATE` statement is a stand-alone statement not related to any `SELECT` statement. With a single statement you can update zero, one, multiple or all rows of a table. The rows to be updated are determined by a *search-condition* that is applied to the table. Optionally, view names and table names can be assigned a *correlation-name*.

Note: The number of rows that have actually been updated with a "searched" `UPDATE` can be ascertained by using the system variable `*ROWCOUNT`.

```
UPDATE           { view-name [correlation-name] SET *                     }
                 { table-name [correlation-name] SET assignment-list      }

 [WHERE                       {  RR  }
search-condition] [    WITH   {  RS  }   ]
                              {  CS  }
```

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

Syntax Element Description - Syntax 1:

Syntax Element	Description
view-name	**View Name:** Refers to the name of a Natural view as defined in the `DEFINE DATA` statement. For further information, see *view-name* (in the section *Basic Syntactical Items*).
correlation-name	**Correlation Name:** The item *correlation-name* represents an alias name for a *table-name*.

Syntax Element	Description
	For further information, see `correlation-name` (in the section ***Basic Syntactical Items***).
`SET`	**SET Clause:** If a view has been specified for updating, an asterisk (*) has to be specified in the `SET` clause, because all columns of the view must be updated. If a table has been specified for updating, the `SET` clause must contain either an *`assignment-list`* or the name of the view which contains the columns to be updated.
`assignment-list`	**Assignment List:** See ***Assignment List*** below.
`WHERE` *`search-condition`*	**WHERE Clause:** This clause is used to specify the selection criteria for the rows to be updated. If no `WHERE` clause is specified, the entire table is updated.
`WITH`	**WITH - Isolation Level Clause:** This clause allows the explicit specification of the isolation level used when locating the row to be updated. For detailed information, see ***WITH isolation-level*** in the description of the `SELECT` statement. It is only valid against Db2 databases. When used against other databases, it will cause runtime errors. `CS` — Cursor Stability `RR` — Repeatable Read `RS` — Read Stability

Assignment List

```
{ column-name = { scalar-expression } } , ...
                { NULL              }
```

In an *`assignment-list`*, you can assign values to one or more columns. A value can be either a *`scalar-expression`* or `NULL`. For further information, see ***Scalar Expressions***.

If the value `NULL` has been assigned, it means that the addressed field is to contain no value (not even the value "0" or "blank").

Syntax Element Description:

Syntax Element	Description
column-name	**Column Name:** Specifies the name of a column of the result table of the `MERGE` statement that is not the same name as another include column or a column in the target table.
`NULL`	**NULL Option:** Specifies the null value as the new value of the column. If the value `NULL` has been assigned, it means that the addressed field is to contain no value (not even the value `0` or "blank").

Syntax 2 - Positioned UPDATE

The "positioned" `UPDATE` statement always refers to a cursor within a database loop. Thus, the table or view referenced by a positioned `UPDATE` statement must be the same as the one referenced by the corresponding `SELECT` statement; otherwise an error message is returned. A positioned `UPDATE` cannot be used with a non-cursor selection.

Common Set Syntax:

```
UPDATE { view-name  SET *               } [WHERE CURRENT OF CURSOR (r)]
       { view-name  SET assignment-list }
```

Syntax Element Description - Syntax 2:

Syntax Element	Description
view-name	**Natural View:** Refers to the name of a Natural view as defined in the `DEFINE DATA` statement; see also *view-name* (in the section ***Basic Syntactical Items***).
`SET *` `SET` *assignment-list*	**SET Clause:** If a Natural view has been specified for updating, an asterisk (*) has to be specified in the `SET` clause, because all columns of the view must be updated. If a table has been specified for updating, the `SET` clause must contain either an *assignment-list* or the name of the view which contains the columns to be updated.
`WHERE CURRENT OF CURSOR (`*r*`)`	**Statement Reference:** The `(`*r*`)` notation is used to reference the statement which was used to select the row to be updated. If no statement reference is specified, the `UPDATE` statement is related to the innermost active processing loop in which a database record was selected.

Examples

Example 1 - Searched UPDATE

```
DEFINE DATA LOCAL
1 PERS VIEW OF SQL-PERSONNEL
2 NAME
2 AGE
...
END-DEFINE
...
ASSIGN AGE = 45
ASSIGN NAME = 'SCHMIDT'
UPDATE PERS SET * WHERE NAME = 'SCHMIDT'
...
```

Example 2 - Searched UPDATE with assignment-list

```
DEFINE DATA LOCAL
1 PERS VIEW OF SQL-PERSONNEL
2 NAME
2 AGE
...
END-DEFINE
...
UPDATE SQL-PERSONNEL SET AGE = AGE + 1 WHERE NAME = 'SCHMIDT'
...
```

Example 3 - Positioned UPDATE

```
DEFINE DATA LOCAL
1 PERS VIEW OF SQL-PERSONNEL
2 NAME
2 AGE
...
END-DEFINE
...
SELECT * INTO PERS FROM SQL_PERSONNEL WHERE NAME = 'SCHMIDT'
COMPUTE AGE = AGE + 1
UPDATE PERS SET * WHERE CURRENT OF CURSOR
```

```
END-SELECT
...
```

Example 4 - Positioned UPDATE with assignment-list

```
DEFINE DATA LOCAL
1 PERS VIEW OF SQL-PERSONNEL
2 NAME
2 AGE
...
END-DEFINE
...
SELECT * INTO PERS FROM SQL-PERSONNEL WHERE NAME = 'SCHMIDT'
UPDATE SQL-PERSONNEL SET AGE = AGE + 1 WHERE CURRENT OF CURSOR
END-SELECT
...
```

142 UPDATELOB

```
UPDATELOB [OF] [RECORD] [(r)] [IN] [FILE] view-name
 [PASSWORD=operand1]
 [CIPHER=operand2]
 [[STARTING] [AT] OFFSET [=] operand3]
 [        TRUNCATE {        [REMAINDER] }        ]
 [                 {        [AT] OFFSET }        ]
```

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

Related Statements: READ | FIND | GET | READLOB | UPDATE

Belongs to Function Group: ***Database Access and Update***

Function

The UPDATELOB statement is used to update a data segment of a LOB field (Large OBject field) in a database record. The position of the value modification is freely selectable. The record to be updated must have been previously selected with a FIND, READ, or GET statement or created with a STORE statement.

Hold Status

The use of the UPDATELOB statement causes each record read for processing in the corresponding FIND, READ, or GET statement to be placed in exclusive hold.

For further information, see *Record Hold Logic* in the *Programming Guide*.

Restrictions

The UPDATELOB statement

- can only be used for access to Adabas databases;
- must not be entered on the same line as the statement used to select the record to be updated;
- is only applicable to update a single LOB field.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S				A													yes	no
operand2	C	S						N											yes	no
operand3	C	S						N	P	I		B *							yes	no

* Format B of *operand3* may be used with a length of less than or equal to 4.

Syntax Element Description:

Syntax Element	Description
`(r)`	**Statement Reference:** The notation `(r)` is used to indicate the statement in which the record to be modified was read or created. *r* may be specified as a source-code line number or as a statement label. You may reference a `FIND`, `READ`, `GET` or `STORE` statement. If no reference is specified, the `UPDATELOB` statement will reference the innermost active `READ` or `FIND` processing loop. If no `READ` or `FIND` loop is active, it will reference the last preceding `GET` statement. To reference a `STORE` statement, you have always to provide the `(r)` notation. **Note:** The `UPDATELOB` statement must be placed within the `READ` or `FIND` loop it references.
view-name	**View Name:** As *view-name*, you specify the name of a view, which must have been defined either within a `DEFINE DATA` statement or outside the program in a global or local data area. ■ The view has to contain just a single-valued LOB field, additional fields are not allowed. ■ If the LOB is a MU or PE field, a unique occurrence must be specified; a range notation is not allowed. ■ The LOB field must be defined in the view with a fixed (non-dynamic) length.
`PASSWORD=`*operand1* `CIPHER=`*operand2*	**PASSWORD and CIPHER Clauses:** The `PASSWORD` clause is used to provide a password when retrieving data from a file which is password-protected.

Syntax Element	Description
	The CIPHER clause is used to provide a cipher key when retrieving data from a file which is enciphered. See the statements FIND and PASSW for further information.
STARTING AT OFFSET=*operand3*	**STARTING AT OFFSET Clause:** Provides the start offset within the LOB field, where the operation is executed. The leftmost byte of the LOB field is offset zero (0). *operand3* must be provided either in the form of a numeric constant or as a user-defined variable, without precision digits. The field is not modified by the UPDATELOB execution. If the offset value is greater than the LOB length, the gap is filled with blanks. This means a LOB field can be updated at a position which is beyond its length. If this clause is omitted, start offset (0) is assumed.
TRUNCATE REMAINDER or TRUNCATE AT OFFSET	**TRUNCATE Clause:** If TRUNCATE REMAINDER is specified, the remaining LOB field data is truncated after the new segment has been written into the LOB field. This makes the end of the inserted segment to the end of the LOB field. If TRUNCATE AT OFFSET is specified, the data behind the specified starting offset is truncated. A segment insert into the LOB field is not performed. After this, the LOB length is equal to *operand3*. If this clause is omitted, the data behind the inserted segment is preserved.

System Variable Available with UPDATELOB

The Natural system variable *NUMBER is provided with the UPDATELOB statement.

The format/length of this system variable is P10. This format/length cannot be changed.

System Variable	Explanation
*NUMBER	The system variable *NUMBER returns the sum of the start offset and the number of characters inserted. This value represents the starting offset for the next UPDATELOB, if a consecutive area of the LOB field is replaced with multiple calls. The number of inserted characters is either the byte length of the LOB segment defined in the view or zero (0) if the TRUNCATE AT OFFSET clause was specified. The *NUMBER field returned by the UPDATELOB statement must always be provided with a reference label or number (for example, *NUMBER(0430)) when used.

Functional Considerations

- An UPDATELOB operates a record which was set into hold by an associated FIND, READ, GET or STORE statement. The link is either implicit via the current active reference or explicit with (*r*) notation.
- The view used by the associated statement and the view used by the UPDATELOB have to access the same database and file number. This is automatically assured if the views are derived from the same data definition module (DDM).
- If the insert position *operand3* is greater than the LOB length, the gap is filled with blanks. This means you may update a LOB field at a position which is beyond its length.
- You cannot replace *m* bytes with *n* bytes - or in other words, it is not admissible to substitute a LOB part with a data segment of different length.
- The value returned with *NUMBER is the high-water mark indicating the position inside the LOB where the last insert has ended. If a number of consecutive update operations is demanded, this value should always be retained as STARTING AT value for the next UPDATELOB execution.

Examples

Example 1 - Store New Record and Fill LOB Segment

```
DEFINE DATA LOCAL
1 V1 VIEW OF ..
  2 PERSONNEL-ID
  2 NAME
1 V2 VIEW OF ..
  2 LOBFIELD_SEGMENT  /* LOB field defined in DDM with (A1024).
END-DEFINE
*
**=====================================================
** Store new record
**=====================================================
V1.PERSONNEL-ID := '12345678'
V1.NAME         := 'Smith'
LAB1.
STORE V1   /* Store new record with 2 fixed length fields.
*
MOVE ALL 'X' TO LOBFIELD_SEGMENT
```

```
**=====================================================
** Update LOB field
**=====================================================
UPDATELOB (LAB1.) IN FILE V2  /* INSERT 1 KB SEGMENT (LOBFIELD_SEGMENT)
                              /* IN LOB.
     STARTING AT OFFSET = 2048
                   /* STORE DATA IN LOB RANGE 2049-3072.
                   /* FIRST 2 KBS ARE AUTO-FILLED WITH BLANKS BY THE DB.
END TRANSACTION
END
```

Example 2 - Add LOB Data to Existent Record, Piece by Piece

```
DEFINE DATA LOCAL
1 V1 VIEW OF EMPLOYEES-V2009
  2 PERSONNEL-ID
  2 NAME
  2 L@PICTURE
1 V2 VIEW OF EMPLOYEES-V2009
  2 PICTURE_SEGMENT  /* LOB field defined in DDM with (A1024).
  2 REDEFINE PICTURE
    3 PICTURE_B (B1024)
1 #OFF   (I4)
END-DEFINE
*
**=====================================================
** Read record to be updated
**=====================================================
LAB1.
READ (1) V1 BY PERSONNEL-ID = '60008339'
                        /* Read record and set into exclusive hold.
  RESET #OFF            /* Start to overwrite LOB field from the beginning.
  /*=====================================================
  /* Read data from work file and put into LOB field
  /*=====================================================
  READ WORK FILE 7 PICTURE_B
                        /* Start to read picture data (.jpg) from work file.
LAB2.
    UPDATELOB (LAB1.) IN FILE V2
              STARTING AT OFFSET #OFF
    #OFF := *NUMBER(LAB2.)    /* Keep next position to append.
  END-WORK
END-READ
**=====================================================
END TRANSACTION
END
```

Example 3 - Truncate LOB Field

```
DEFINE DATA LOCAL
1 V1 VIEW OF EMPLOYEES-V2009
  2 PERSONNEL-ID
  2 NAME
  2 L@PICTURE
1 V2 VIEW OF EMPLOYEES-V2009
1 V3 VIEW OF EMPLOYEES-V2009
  2 PICTURE_SEGMENT   /* LOB field defined in DDM with (A1024).
END-DEFINE
*
**=====================================================
** Read record to be updated
**=====================================================
LAB1.
READ V1 BY PERSONNEL-ID        /* Read records.
  IF L@PICTURE > 10240 THEN    /* Check if LOB length is too high.
LAB2.
    GET V2 RECORD *ISN(LAB1.)  /* Set record to be updated into exclusive hold.
    UPDATELOB (LAB2.) IN FILE V3
              STARTING AT OFFSET 10240
              TRUNCATE AT OFFSET         /* Truncate LOB data beyond 10KB.
    END TRANSACTION
  END-IF
END-READ
END
```

Example 4 - Read LOB Data to Existent Record and Update LOB Segment

```
DEFINE DATA LOCAL
1 V1 VIEW OF ..
  2 NAME
1 V2 VIEW OF ..
  2 DOCUMENT_SEGMENT      /* LOB field defined in DDM with (A100).
1 #ISN    (I4)
1 #POS    (I4)
1 #LENGTH (I4) INIT <100>
END-DEFINE
*
**=====================================================
** Read record to be updated
**=====================================================
INPUT (AD=T)
  / ' Read record (ISN):' #ISN
*
G1.
GET V1 RECORD #ISN       /* Get record with ISN and set into exclusive hold.
*
**=====================================================
```

```
** Read LOB data and update segment of LOB field
**==================================================
R1.
READLOB V2 WITH ISN = #ISN
     STARTING AT OFFSET = 3000
   ..
   #POS := *NUMBER(R1.) - #LENGTH
   ..
   IF ..
     DOCUMENT_SEGMENT := ..
     UPDATELOB (G1.) IN FILE V2    /* Update current segment in LOB field.
               STARTING AT OFFSET #POS
   END-IF
   ..
END-READLOB
*
END TRANSACTION
END
```

143 UPLOAD PC FILE

Structured Mode Syntax

```
{ UPLOAD } { PC   } [FILE] work-file-number [ONCE]
{ READ   } { WORK }

{ RECORD operand1                                         }
{                                                         }
{    [AND] [SELECT] { [ OFFSET n   ]    operand2 } ...    }
{                   { [ FILLER nX  ] ...         }        }

[GIVING LENGTH operand3]
[AT [END] [OF] [FILE] statement ... END-ENDFILE]
statement ...
END-WORK
```

Reporting Mode Syntax

```
{ UPLOAD }   { PC   }   [FILE] work-file-number [ONCE]
{ READ   }   { WORK }

{ RECORD {operand1 [FILLER nX]} ...                          }
{ [AND] [SELECT] { [ OFFSET n  ]      operand2 } ...         }
{                { [ FILLER nX ] ...           }             }

[GIVING LENGTH operand3]
[ AT [END] [OF] [FILE] { statement                 } ]
[                      { DO statement ... DOEND    } ]
statement ...
LOOP
```

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

Related Statements: `CLOSE PC FILE` | `DOWNLOAD PC FILE` | `READ WORK FILE`

Belongs to Function Group: ***Control of Work Files / PC Files***

Function

The `UPLOAD PC FILE` statement is used to transfer data from a PC to a Linux platform.

See also:

- *Natural Connection* and Entire Connection documentation
- `READ WORK FILE` statement syntax description

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats											Referencing Permitted	Dynamic Definition
operand1		S	A	G		A	U	N	P	I	F	B	D	T	L	C	yes	yes
operand2		S	A	G		A	U	N	P	I	F	B	D	T	L	C	yes	yes
operand3		S								I							yes	yes

When using the work file types `ENTIRECONNECTION` or `TRANSFER`, *operand2* may not be of Format C.

Syntax Element Description:

Syntax Element	Description
work-file-number	**Work File Number:** The number of the work file to be used. This number must correspond to one of the work file numbers for the PC as defined to Natural.
operand1-2	**Field Specification:** With *operand1* and *operand2* you specify the fields to be uploaded from the PC. The fields may be database fields or user-defined variables.
statement	**Statement(s) to be Executed:** In place of *statement*, you must supply one or several suitable statements, depending on the situation. No I/O statement may be placed with the `UPLOAD PC FILE` processing.

`ONCE, SELECT, GIVING LENGTH`
 	
`RECORD`
	Options:
	
	For a description of the `ONCE`, `SELECT`, `GIVING LENGTH` options, refer to the corresponding sections in the description of the `READ WORK FILE` statement.
	
	The `RECORD` option is not permitted for PC work files. It will be rejected at runtime.
	
	When uploading data: If you wish to define a filler, you must use a dummy variable instead of the standard filler notation.

`END-WORK`	**End of UPLOAD PC FILE Statement:** In structured mode, the Natural reserved keyword `END-WORK` must be used to end the `UPLOAD PC FILE` statement.
`LOOP`	

Syntax Element	Description
	In reporting mode, the Natural statement LOOP is used to end the UPLOAD PC FILE statement.

Example

The following program demonstrates the use of the UPLOAD PC FILE statement. The data is first uploaded from the PC and then processed on the Linux platform.

```
** Example 'PCUPEX1': UPLOAD PC FILE
**
** NOTE: Example requires that Natural Connection is installed.
** CAUTION: Executing this example will modify the database records!
************************************************************************
DEFINE DATA LOCAL
01 EMPL VIEW OF EMPLOYEES
   02 PERSONNEL-ID
   02 INCOME
      03 SALARY (1)
*
01 #PID (A8)                                     /* Personnel ID on PC
01 #NEW-INCREASE (N4)                            /* Increase for salary
END-DEFINE
*
UPLOAD PC FILE 7 #PID #NEW-INCREASE              /* Data upload
*
  FIND EMPL WITH PERSONNEL-ID = #PID             /* Data selection
    ADD #NEW-INCREASE TO SALARY (1)              /* Data update on host
    UPDATE
    END TRANSACTION
    ESCAPE BOTTOM
  END-FIND
*
END-WORK
END
```

Output of Program PCUPEX1:

When you run the program, a window appears in which you specify the name of the PC file from which the data is to be uploaded. The data is then uploaded from the PC.

144 WRITE

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

Related Statements: AT END OF PAGE | AT TOP OF PAGE | CLOSE PRINTER | DEFINE PRINTER | DISPLAY | EJECT | FORMAT | NEWPAGE | PRINT | SKIP | SUSPEND IDENTICAL SUPPRESS | WRITE TITLE | WRITE TRAILER

Belongs to Function Group: ***Creation of Output Reports***

Function

The WRITE statement is used to produce output in free format.

The WRITE statement differs from the DISPLAY statement in the following respects:

- Line overflow is supported. If the line width is exceeded for a line, the next field (or text) is written on the next line. Fields or text elements are not split between lines.
- No default column headers are created. The length of the data determines the number of positions printed for each field.
- A range of values/occurrences for an array is output horizontally rather than vertically.

See also the following topics in the *Programming Guide*:

- *Report Format and Control*
- *Statements DISPLAY and WRITE*
- *Index Notation for Multiple-Value Fields and Periodic Groups*
- *Example of DISPLAY VERT with WRITE Statement*
- *Layout of an Output Page*

Syntax 1 - Dynamic Formatting

```
WRITE [(rep)] [NOTITLE] [NOHDR]
   [(statement-parameters)]
   {  [ nX            ]     { 'text' [(attributes)]           }  }
   {  [ nT            ]     { 'c'(n) [(attributes)]           }  }
   {  [ x/y           ]     { ['='] operand1 [(parameters)]   }  }
   {  [ T*field-name  ]                                          }
   {  [ P*field-name  ]                                          }
   {  [ /             ] ...                                      } ...
```

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

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1		S	A	G	N	A	U	N	P	I	F	B	D	T	L		G	O	yes	no

Syntax Element Description:

Syntax Element	Description
`(rep)`	**Report Specification:** The notation `(rep)` is used to specify the identification of the report if multiple reports are to be produced by the program. As report identification, a value in the range `0 - 31` or a logical name which has been assigned using the `DEFINE PRINTER` statement may be specified. If `(rep)` is not specified, the statement will apply to the first report (Report 0). If this printer file is defined to Natural as PC, the report will be downloaded to the PC, see ***Example 6***. For information on how to control the format of an output report created with Natural, see *Report Format and Control* (in the *Programming Guide*).
`NOTITLE`	**Default Page Title Suppression:** Natural generates a single title line for each page resulting from a `WRITE` statement. This title contains the page number, the time of day, and the date. Time of day is set at the beginning of program execution. This default title line may be overridden by using a `WRITE TITLE` statement, or it may be suppressed by using the `NOTITLE` option in the `WRITE` statement. Examples: ■ Default title will be produced: `WRITE NAME` ■ User title will be produced:

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td></td><td>
<pre><code>WRITE NAME WRITE TITLE 'user-title'</code></pre>
No title will be produced:
<pre><code>WRITE NOTITLE NAME</code></pre>
Note:

If the <code>NOTITLE</code> option is used, it applies to all <code>DISPLAY</code>, <code>PRINT</code> and <code>WRITE</code> statements within the same object which write data to the same report.
Page overflow is checked before execution of a <code>WRITE</code> statement. No new page with title or trailer information is generated during the execution of a <code>WRITE</code> statement.

</td></tr>
<tr><td><code>NOHDR</code></td><td>
Column Header Suppression:
The <code>WRITE</code> statement itself does not produce any column headers. However, if you use the <code>WRITE</code> statement in conjunction with a <code>DISPLAY</code> statement, you can use the <code>NOHDR</code> option of the <code>WRITE</code> statement to suppress the column headers generated by the <code>DISPLAY</code> statement. The <code>NOHDR</code> option only takes effect if the execution of the <code>WRITE</code> statement causes a new page to be output.
Without the <code>NOHDR</code> option, the column headers (if any) of the <code>DISPLAY</code> statement would be output on this new page; with <code>NOHDR</code> they will not.
</td></tr>
<tr><td>statement-parameters</td><td>
Parameter Definition at Statement Level:
One or more parameters, enclosed within parentheses, may be specified at statement level, that is, immediately after the <code>WRITE</code> statement.
Each parameter specified will override the corresponding parameter previously specified in a <code>GLOBALS</code> command, <code>SET GLOBALS</code> (in Reporting Mode only) or <code>FORMAT</code> statement.
If more than one parameter is specified, they must be separated by one or more blanks from one another. Each parameter specification must not be split between two statement lines.
Note: The parameter settings applied here will only be regarded for variable fields, but they have no effect on text-constants. If you would like to set field attributes for a text-constant, they have to be set explicitly for this element, see Parameter Definition at Element (Field) Level.
See also:

List of Parameters
Example of Parameter Usage at Statement and Element (Field) Level

</td></tr>
</table>

Syntax Element	Description
	■ *Example 5 - WRITE Statement Using '=' and Parameters on Statement/Element (Field) Level*
`nX`, `nT` , `x/y`, `T*field-name`, `P*field-name`, `'='`, `/`	**Field Positioning Notation:** See *Field Positioning Notations* in the section *Output Format Definitions*.
`'text'`, `'c'(n)`, `attributes`, `operand1`, `parameters`	**Text/Attribute Assignment:** See *Text/Attribute Assignment* in the section *Output Format Definitions*.

List of Parameters

Parameters that can be specified with the WRITE statement		Specification (S = at statement level, E = at element level)
AD	Attribute Definition	SE
AL	Alphanumeric Length for Output	SE
CD	Color Definition	SE
CV	Control Variable	SE
DF	Date Format	SE
DL	Display Length for Output	SE
DY	Dynamic Attributes	SE
EM	Edit Mask	SE
EMU	Unicode Edit Mask	E
FL	Floating Point Mantissa Length	SE
IS	Identical Suppress	SE
LS	Line Size	S
MC	Multiple-Value Field Count	S
MP	Maximum Number of Pages of a Report	S
NL	Numeric Length for Output	SE
PC	Periodic Group Count	S
PM	Print Mode	SE
PS	Page Size *	S
SG	Sign Position	SE
UC	Underlining Character	S
ZP	Zero Printing	SE

* The PS session parameter setting is not considered if the number of occurrences of an array exceeds the PS value.

The individual session parameters are described in the *Parameter Reference*.

See also the following topics in the *Programming Guide*:

- *Centering of Column Headers - HC Parameter*
- *Width of Column Headers - HW Parameter*
- *Filler Characters for Headers - Parameters FC and GC*
- *Underlining Character for Titles and Headers - UC Parameter*

Example of Parameter Usage at Statement and Element (Field) Level

```
DEFINE DATA LOCAL
1 VARI (A4)     INIT <'1234'>                      /*     Output
END-DEFINE                                         /*    Produced
*                                                  /*    ---------
WRITE           'Text'           VARI              /*    Text 1234
WRITE (AD=U)    'Text'           VARI              /*    Text 1234
WRITE           'Text' (AD=U)    VARI (AD=U)       /*    Text 1234
WRITE           'Text' (AD=U)    VARI              /*    Text 1234
END
```

See also ***Example 5 - WRITE Statement Using '=' and Parameters on Statement/Element (Field) Level***.

Output Format Definitions

```
{ [ nX           ]    
  [ nT           ]    
  [ x/y          ]    { 'text' [(attributes)]                 }
  [ T*field-name ]    { 'c' (n) [(attributes)]                }
  [ P*field-name ]    { ['='] operand1 [(parameters)]         }
  [ /            ] ...                                          } ...
```

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

Field Positioning Notations

Syntax Element	Description
*n*X	**Column Spacing:** This notation inserts *n* spaces between columns. Example:

Syntax Element	Description
	`WRITE NAME 5X SALARY` See also: ■ *Example 2 - WRITE Statement Using nX, nT Notation* (below) ■ *Column Spacing - SF Parameter and nX Notation* in the *Programming Guide*
*n*T	**Tab Setting:** The *n*T notation causes positioning (tabulation) to print position *n*. Backward positioning is not permitted. In the following example, NAME is printed beginning in position 25, and SALARY is printed beginning in position 50: `WRITE 25T NAME 50T SALARY` See also: ■ *Example 2 - WRITE Statement Using nX, nT Notation* (below) ■ *Tab Setting - nT Notation* in the *Programming Guide*
x/*y*	***x*/*y* Positioning:** The *x*/*y* notation causes the next element to be placed *x* lines below the output of the last statement, beginning in column *y*. *y* must not be zero. Backward positioning in the same line is not permitted. See also *Positioning Notation x/y* (in the *Programming Guide*).
T**field-name*	**Field Related Positioning:** The notation T* is used to position to a *specific print position of a field* used in a previous DISPLAY statement. Backward positioning is not permitted. See also: ■ *Example 3 - WRITE Statement Using T* Notation* (below) ■ *Tab Notation - T*field* (in the *Programming Guide*)
P**field-name*	**Field and Line Related Positioning:** The notation P* is used to position to a *specific print position and line of a field* used in a previous DISPLAY statement. It is most often used in conjunction with vertical printing mode. Backward positioning is not permitted. See also: ■ *Example 4 - WRITE Statement Using P* Notation* (below) ■ *Tab Notation P*field* (in the *Programming Guide*)

Syntax Element	Description
'='	**Field Content Positioned behind Field Heading:** When placed before a field, the equal sign '=' results in the display of the field heading (as defined in the DEFINE DATA statement or in the DDM) followed by the field contents. See also: ■ *Example 1 - WRITE Statement Using '=', 'text', '/'* ■ *Example 5 - WRITE Statement Using '=' and Statement/Element Parameters*
/	**Line Advance - Slash Notation:** When placed between fields or text elements, a slash (/) causes positioning to the beginning of the next print line. Example: `WRITE NAME / SALARY` Multiple slash (/) notations may be used to cause multiple line advances. See also: ■ *Example 1 - WRITE Statement Using '=', 'text', '/'* (below) ■ *Line Advance - Slash Notation* (in the *Programming Guide*) ■ *Example 2 - Line Advance in WRITE Statement* (in the *Programming Guide*)

Text/Attribute Assignments

Syntax Element	Description
'*text*'	**Text Assignment:** The character string enclosed by single quotes is displayed. Example: `WRITE 'EMPLOYEE' NAME 'MARITAL/STATUS' MAR-STAT` See also: ■ *Example 1 - WRITE Statement Using '=', 'text', '/'* (below) ■ *Text Notation, Defining a Text to Be Used with a Statement* in the *Programming Guide*
'*c*'(*n*)	**Character Repetition:** The character enclosed by single quotes is displayed *n* times immediately before the field value.

Syntax Element	Description

	For example:
	
	`WRITE '*' (5) '=' NAME`
	
	results in
	
	`***** SMITH`
	
	See also *Text Notation, Defining a Character to Be Displayed* `n` *Times before a Field Value* (in the *Programming Guide*).

`attributes`	**Field Representation and Color Attributes:**
	
	It is possible to assign various attributes for text/field display. These attributes and the syntax that may be used are described in the section ***Output Attributes*** below.
	
	Examples:
	
	`WRITE 'TEXT' (BGR)`
	`WRITE 'TEXT' (B)`
	`WRITE 'TEXT' (BBLC)`

`operand1`	**Field to be Written:** `operand1` specifies the field whose content is to be written in this place.
`parameters`	**Parameter Definition at Element (Field) Level:** One or more parameters, enclosed within parentheses, may be specified at element (field) level, that is, immediately after `operand1`. Each parameter specified in this manner will override the corresponding parameter previously specified **at statement level** or in a `GLOBALS` command, `SET GLOBALS` (in Reporting Mode only) or `FORMAT` statement. If more than one parameter is specified, one or more blanks must be placed between each entry. An entry may not be split between two statement lines. See also: ■ ***List of Parameters*** ■ ***Example of Parameter Usage at Statement and Element (Field) Level***

Output Attributes

`attributes` indicates the output attributes to be used for text display. Attributes can be:

```
⎧ AD=ad-value ... ⎫
⎨ CD=cd-value     ⎬ ...
⎩ PM=pm-value ... ⎭

{ ad-value        } ...
{ cd-value        }
```

Where:

ad-value, *cd-value* and *pm-value* denote the possible values of the corresponding session parameters AD, CD and PM described in the relevant sections of the *Parameter Reference* documentation.

The compiler actually accepts more than one attribute value for an output field. For example, you can specify: AD=BDI. In such a case, however, only the last value applies. In the given example, only the value I becomes effective and the output field is displayed intensified.

For an alphanumeric/Unicode constant (Natural data format A or U), you can specify *ad-value* and/or *cd-value* without preceding CD= or AD=, respectively. The single value entered is then checked against all possible CD values first. For example: a value of IRE will be interpreted as intensified/red but not as intensified/right-justified/mandatory. You cannot combine a single *cd-value* or *ad-value* with a value preceded by CD= or AD=.

Syntax 2 - Using Predefined Form/Map

```
WRITE                                  ⎧ FORM ⎫              ⎡ operand2 ...  ⎤
 [(rep)] [NOTITLE] [NOHDR]             ⎨      ⎬  operand1    ⎢               ⎥
[USING]                                ⎩ MAP  ⎭              ⎣ NO PARAMETER  ⎦
```

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

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S				A													no	no
operand2		S	A	G	N	A	U	N	P	I	F	B	D	T	L				yes	no

Syntax Element Description:

Syntax Element	Description
[USING] FORM [USING] MAP	**Use of Predefined Form/Map Layout:** This option may be used to indicate that a form/map layout previously defined with the map editor is to be used. A map layout used in a WRITE statement does not automatically create a new page each time the map is output. For the line spacing, the LS parameter setting must be 1 byte greater than the LS setting defined in the map.
operand1	**Form/Map Name:** *operand1* is the name of the form/map to be used.
operand2	**Field to be Written:** *operand2* is the name(s) of the field(s) to be written. If *operand1* is a constant and *operand2* is omitted, the fields are taken from the map source at compilation time. The fields must agree in number, sequence, format, length and (for arrays) number of occurrences with the fields in the referenced form/map; otherwise, an error occurs. If FORM or MAP does not require any parameters, specify the NO PARAMETER option.
NOTITLE/NOHDR	**Title Line/Column Header Suppression:** NOTITLE and NOHDR are described under ***Syntax 1*** of the WRITE statement.

Examples

- Example 6 - Report Specification with Output File Defined to Natural as PC

Example 1 - WRITE Statement Using '=', 'text', '/'

```
** Example 'WRTEX1': WRITE (with '=', 'text', '/')
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 FULL-NAME
    3 FIRST-NAME
    3 MIDDLE-I
    3 NAME
  2 CITY
  2 COUNTRY
END-DEFINE
*
LIMIT 1
READ EMPL-VIEW BY NAME
  /*
  WRITE NOTITLE
        '=' NAME '=' FIRST-NAME '=' MIDDLE-I //
        'L O C A T I O N' /
        'CITY:    ' CITY    /
        'COUNTRY:' COUNTRY //
  /*
END-READ
END
```

Output of Program WRTEX1:

```
NAME: ABELLAN             FIRST-NAME: KEPA                 MIDDLE-I:

L O C A T I O N
CITY:    MADRID
COUNTRY: E
```

Example 2 - WRITE Statement Using nX, nT Notation

```
** Example 'WRTEX2': WRITE (with nX, nT notation)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
END-DEFINE
*
LIMIT 4
READ EMPL-VIEW BY NAME
  WRITE NOTITLE 5X NAME 50T JOB-TITLE
```

```
END-READ
END
```

Output of WRTEX2:

```

ABELLAN                                     MAQUINISTA
ACHIESON                                    DATA BASE ADMINISTRATOR
ADAM                                        CHEF DE SERVICE
ADKINSON                                    PROGRAMMER
```

Example 3 - WRITE Statement Using T* Notation

```
** Example 'WRTEX3': WRITE (with T* notation)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 SALARY (1)
END-DEFINE
*
LIMIT 5
READ EMPL-VIEW BY CITY STARTING FROM 'ALBU'
  DISPLAY NOTITLE CITY NAME SALARY (1)
  AT BREAK CITY
    /*
    WRITE / 'CITY AVERAGE:' T*SALARY (1) AVER(SALARY(1)) //
    /*
  END-BREAK
END-READ
END
```

Output of Program WRTEX3:

```
        CITY                 NAME           ANNUAL
                                            SALARY
-------------------- -------------------- ----------

ALBUQUERQUE          HAMMOND                   22000
ALBUQUERQUE          ROLLING                   34000
ALBUQUERQUE          FREEMAN                   34000
ALBUQUERQUE          LINCOLN                   41000

CITY AVERAGE:                                  32750


ALFRETON             GOLDBERG                   4800

CITY AVERAGE:                                   4800
```

Example 4 - WRITE Statement Using P* Notation

```
** Example 'WRTEX4': WRITE (with P* notation)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 BIRTH
  2 SALARY (1)
END-DEFINE
*
LIMIT 3
READ EMPL-VIEW BY CITY FROM 'N'
  DISPLAY NOTITLE NAME CITY
          VERT AS 'BIRTH/SALARY' BIRTH (EM=YYYY-MM-DD) SALARY (1)
  SKIP 1
  AT BREAK CITY
    WRITE  / 'CITY AVERAGE' P*SALARY (1) AVER(SALARY (1)) //
  END-BREAK
END-READ
END
```

Output of Program WRTEX4:

```
        NAME                 CITY           BIRTH
                                            SALARY
-------------------- -------------------- ----------

WILCOX               NASHVILLE            1970-01-01
                                               38000

MORRISON             NASHVILLE            1949-07-10
                                               36000


CITY AVERAGE                                   37000


BOYER                NEMOURS              1955-11-23
                                              195900


CITY AVERAGE                                  195900
```

Example 5 - WRITE Statement Using '=' and Parameters on Statement/Element (Field) Level

```
** Example 'WRTEX5': WRITE (using '=', statement/element parameters)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 PERSONNEL-ID
  2 PHONE
END-DEFINE
*
LIMIT 2
READ EMPL-VIEW BY NAME
  WRITE NOTITLE (AL=16 NL=8)
        '=' PERSONNEL-ID '=' NAME '=' PHONE (AL=10 EM=XXX-XXXXXXX)
END-READ
END
```

Output of Program WRTEX5:

```
PERSONNEL ID: 60008339        NAME: ABELLAN         TELEPHONE: 435-6726
PERSONNEL ID: 30000231        NAME: ACHIESON        TELEPHONE: 523-341
```

Example 6 - Report Specification with Output File Defined to Natural as PC

```
** Example 'PCDIEX1': DISPLAY and WRITE to PC
**
** NOTE: Example requires that Natural Connection is installed.
************************************************************************
DEFINE DATA LOCAL
01 PERS VIEW OF EMPLOYEES
  02 PERSONNEL-ID
  02 NAME
  02 CITY
END-DEFINE
*
FIND PERS WITH CITY = 'NEW YORK'               /* Data selection
  WRITE (7) TITLE LEFT 'List of employees in New York' /
  DISPLAY (7)          /* (7) designates the output file (here the PC).
    'Location'  CITY
    'Surname'   NAME
    'ID'        PERSONNEL-ID
END-FIND
END
```

145 WRITE TITLE

```
WRITE [(rep)] TITLE [LEFT [JUSTIFIED]] [UNDERLINED]
      [(statement-parameters)]
      nX
      nT
      x/y                          'text' [(attributes)]
      T*field-name                 'c' (n)  [(attributes)]
      P*field-name                 ['='] operand1 [(parameters )]
      /             ...                                            ...
      [SKIP operand2 [LINES]]
```

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

Related Statements: AT END OF PAGE | AT TOP OF PAGE | CLOSE PRINTER | DEFINE PRINTER | DISPLAY | EJECT | FORMAT | NEWPAGE | PRINT | SKIP | SUSPEND IDENTICAL SUPPRESS | WRITE | WRITE TRAILER

Belongs to Function Group: ***Creation of Output Reports***

Function

The WRITE TITLE statement is used to override the default page title with a page title of your own. It is executed whenever a new page is initiated.

See also the following sections in the *Programming Guide*:

- *Report Format and Control*
- *Report Specification - (rep) Notation*
- *Layout of an Output Page*
- *Page Titles, Page Breaks, Blank Lines*
- *Define Your Own Page Title - WRITE TITLE Statement*
- *Text Notation*

Processing

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

If a report is produced by statements in different objects, the `WRITE TITLE` statement is only executed if it is contained in the same object as the statement that causes a new page to be initiated.

Restrictions

- `WRITE TITLE` may be specified only once per report.
- `WRITE TITLE` cannot be specified within a special condition statement block.
- `WRITE TITLE` cannot be specified within a subroutine.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1		S	A	G	N	A	U	N	P	I	F	B	D	T	L		G	O	yes	no
operand2	C	S						N	P	I		B							yes	no

Syntax Element Description:

Syntax Element	Description
`(rep)`	**Report Specification:** If multiple reports are to be produced, the notation `(rep)` may be used to specify the identification of the report for which the `WRITE TITLE` statement is applicable. As report identification, a value in the range `0 - 31` or a logical name which has been assigned using the `DEFINE PRINTER` statement may be specified. If `(rep)` is not specified, the `WRITE TITLE` statement applies to the first report (Report 0). For information on how to control the format of an output report created with Natural, see *Report Format and Control* (in the *Programming Guide*).
`LEFT JUSTIFIED`	**Page Title Justification and/or Underlining:**

Syntax Element	Description
UNDERLINED	By default, page titles are centered and not underlined. LEFT JUSTIFIED and UNDERLINED may be specified to override these defaults. If UNDERLINED is specified, the underlining character (system default or specified with the session parameter UC (Underlining Character) in a FORMAT statement) is printed underneath the title and runs the width of the line size (see session parameter LS). Natural first applies all spacing or tab specifications and creates the line before centering the whole line. For example, a notation of 10T as the first element would cause the centered header to be positioned five positions to the right.
statement-parameters	**Parameter Definition at Statement Level:** One or more parameters, enclosed within parentheses, may be specified at statement level, that is, immediately after the WRITE TITLE statement. Each parameter specified in this manner will override the corresponding parameter previously specified in a GLOBALS command, SET GLOBALS (in Reporting Mode only) or FORMAT statement. If more than one parameter is specified, one or more blanks must be present between each entry. An entry may not be split between two statement lines. **Note:** The parameter settings applied here will only be regarded for variable fields, but they have no effect on text-constants. If you would like to set field attributes for a text-constant, they have to be set explicitly for this element, see *Parameter Definition at Element (Field) Level*. For information on which parameters may be used, see *List of Parameters* (in the WRITE statement documentation).
*n*X *n*T *x*/*y* T**field-name* P**field-name* /	**Format Notation and Spacing Elements:** See *Format Notation and Spacing Elements* (below).
'*text*' '*c*' (*n*) *attributes*	**Text/Attribute Assignment:** See *Text/Attribute Assignments* (below).
operand1	**Field to Be Displayed in Title:** *operand1* represents the field(s) to be displayed within the title.
parameters	**Parameter Definition at Element (Field) Level:** One or more parameters, enclosed within parentheses, may be specified at element (field) level, that is, immediately after *operand1*. Each parameter specified in this manner will override the corresponding parameter previously specified **at statement level** or in a GLOBALS command, SET GLOBALS (in Reporting Mode only) or FORMAT statement.

Syntax Element	Description
	If more than one parameter is specified, one or more blanks must be present between each entry. An entry may not be split between two statement lines. For information on which parameters may be used, see ***List of Parameters*** (in the `WRITE` statement documentation).
`SKIP` *operand2* `LINES`	**Lines to Be Skipped:** `SKIP` may be used to cause lines to be skipped immediately after the title line. The number of lines to be skipped may be specified in *operand2* as a numeric constant or as the content of a numeric variable. **Note:** `SKIP` after `WRITE TITLE` is always interpreted as the `SKIP` clause of the `WRITE TITLE` statement, and not as an independent statement. If you wish an independent `SKIP` statement after a `WRITE TITLE` statement, use a semicolon (;) to separate the two statements from one another.

Format Notation and Spacing Elements

Syntax Element	Description
*n*X	**Column Spacing:** This notation inserts *n* spaces between columns.
*n*T	**Tab Setting:** The *n*T notation causes positioning (tabulation) to print position *n*. Backward positioning is not permitted.
x/*y*	***x*/*y* Positioning:** Causes the next element to be placed *x* lines below the output of the last statement, beginning in column *y*. *y* must not be zero. Backward positioning in the same line is not permitted.
T**field-name*	**Field-Related Positioning:** The T* notation causes positioning to a *specific print position of a field* used in a previous `DISPLAY` statement. Backward positioning is not permitted.
P**field-name*	**Field- and Line-Related Positioning:** The P* notation causes positioning to a *specific print position and line of a field* used in a previous `DISPLAY` statement. It is most often used in conjunction with vertical printing mode. Backward positioning is not permitted.
/	**Line Advance - Slash Notation:** When placed between fields or text elements, a slash (/) causes positioning to the beginning of the next print line.

Text/Attribute Assignments

Syntax Element	Description
'text'	**Text Assignment:** The character string enclosed by single quotes is displayed.
'*c*'(*n*)	**Character Repetition:** The character enclosed by single quotes is displayed *n* times immediately before the field value.

attributes	**Field Representation and Color Attributes:**
	
	It is possible to assign various attributes for text/field display. These attributes and the syntax that may be used are described in the section ***Output Attributes*** below.
	
	Examples:
	
	`WRITE TITLE 'TEXT' (BGR)`
	`WRITE TITLE 'TEXT' (B)`
	`WRITE TITLE 'TEXT' (BBLC)`

Output Attributes

attributes indicates the output attributes to be used for text display. Attributes can be:

```
{ AD=ad-value ... }
{ CD=cd-value     }
{ PM=pm-value ... } ...

{ ad-value }
{ cd-value } ...
```

Where:

ad-value, *cd-value* and *pm-value* denote the possible values of the corresponding session parameters AD, CD and PM described in the relevant sections of the *Parameter Reference* documentation.

The compiler actually accepts more than one attribute value for an output field. For example, you can specify: AD=BDI. In such a case, however, only the last value applies. In the given example, only the value I becomes effective and the output field is displayed intensified.

For an alphanumeric/Unicode constant (Natural data format A or U), you can specify *ad-value* and/or *cd-value* without preceding CD= or AD=, respectively. The single value entered is then checked against all possible CD values first. For example: a value of IRE will be interpreted as intensified/red but not as intensified/right-justified/mandatory. You cannot combine a single *cd-value* or *ad-value* with a value preceded by CD= or AD=.

Example

```
** Example 'WTIEX1': WRITE  (with TITLE option)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 JOB-TITLE
END-DEFINE
*
*
FORMAT LS=70
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED
      *TIME 3X 'PEOPLE LIVING IN NEW YORK CITY'
      11X 'PAGE:' *PAGE-NUMBER
SKIP 1
*
FIND EMPL-VIEW WITH CITY = 'NEW YORK'
  DISPLAY NAME FIRST-NAME 3X JOB-TITLE
END-FIND
END
```

Output of Program WTIEX1:

```
09:33:16.5   PEOPLE LIVING IN NEW YORK CITY           PAGE:      1
---------------------------------------------------------------------


        NAME              FIRST-NAME                CURRENT
                                                   POSITION
-------------------- --------------------   -------------------------

RUBIN                SYLVIA                 SECRETARY
WALLACE              MARY                   ANALYST
```

146 WRITE TRAILER

```
WRITE [(rep)] TRAILER [LEFT [JUSTIFIED]] [UNDERLINED]
      [(statement-parameters)]
  ⎧ ⎡ nX          ⎤                                                ⎫
  ⎪ ⎢ nT          ⎥                                                ⎪
  ⎪ ⎢ x/y         ⎥       ⎧ 'text' [(attributes)]               ⎫  ⎪
  ⎨ ⎢ T*field-name⎥       ⎨ 'c' (n)  [(attributes)]             ⎬  ⎬
  ⎪ ⎢ P*field-name⎥       ⎩ ['='] operand1 [(parameters)]       ⎭  ⎪
  ⎩ ⎣ /           ⎦ ...                                            ⎭ ...
      [SKIP operand2 [LINES]]
```

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

Related Statements: AT END OF PAGE | AT TOP OF PAGE | CLOSE PRINTER | DEFINE PRINTER | DISPLAY | EJECT | FORMAT | NEWPAGE | PRINT | SKIP | SUSPEND IDENTICAL SUPPRESS | WRITE | WRITE TITLE

Belongs to Function Group: ***Creation of Output Reports***

Function

The WRITE TRAILER statement is used to output text or the contents of variables at the bottom of a page.

See also the following sections (in the *Programming Guide*):

- *Report Format and Control*
- *Report Specification - (rep) Notation*
- *Layout of an Output Page*
- *Page Trailer - WRITE TRAILER Statement*
- *Text Notation*

Processing

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

This statement is executed when an end-of-page or end-of-data condition is detected, or when a SKIP or NEWPAGE statement causes a page advance. It is not executed as a result of an EJECT statement.

The end-of-page condition is checked only after the processing of an entire DISPLAY/WRITE statement. If a DISPLAY/WRITE statement produces multiple lines of output, overflow of the physical page may occur before the end-of-page condition is reached.

If a report is produced by statements in different objects, the WRITE TRAILER statement is only executed if it is contained in the same object as the statement that causes the end-of-page condition.

Logical Page Size

The logical page size (specified with the session parameter PS) should be less than the physical page size to ensure that the trailer information appears at the bottom of the same page.

Restrictions

- WRITE TRAILER may be specified only once per report.
- WRITE TRAILER cannot be specified within a special condition statement block.
- WRITE TRAILER cannot be specified within a subroutine.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1		S	A	G	N	A	U	N	P	I	F	B	D	T	L		G	O	yes	no
operand2	C	S						N	P	I		B							yes	no

Syntax Element Description:

Syntax Element	Description
(*rep*)	**Report Specification:** If multiple reports are to be produced, the notation (*rep*) may be used to specify the identification of the report for which the WRITE TRAILER statement is applicable. As report identification, a value in the range 0 - 31 or a logical name which has been assigned using the DEFINE PRINTER statement may be specified. If (*rep*) is not specified, the WRITE TRAILER statement applies to the first report (Report 0).

Syntax Element	Description
	For information on how to control the format of an output report created with Natural, see *Report Format and Control* (in the *Programming Guide*).
`LEFT JUSTIFIED` `UNDERLINED`	**Title Justification and/or Underlining:** By default, the trailer lines are centered and not underlined. `LEFT JUSTIFIED` and `UNDERLINED` may be specified to override these defaults. If `UNDERLINED` is specified, the underlining character (either default or specified with the session parameter `UC`) is printed underneath the trailer and runs the width of the line size (session parameter `LS`). Natural first applies all spacing or tab specifications and creates the line before centering the whole line. For example, a notation of `10T` as the first element would cause the centered header to be positioned five positions to the right.
statement-parameters	**Parameter Definition at Statement Level:** One or more parameters, enclosed within parentheses, may be specified at statement level, that is, immediately after the `WRITE TRAILER` statement. Each parameter specified in this manner will override the corresponding parameter previously specified in a `GLOBALS` command, `SET GLOBALS` (in Reporting Mode only) or `FORMAT` statement. If more than one parameter is specified, one or more blanks must be present between each entry. An entry may not be split between two statement lines. **Note:** The parameter settings applied here will only be regarded for variable fields, but they have no effect on text-constants. If you would like to set field attributes for a text-constant, they have to be set explicitly for this element, see ***Parameter Definition at Element (Field) Level***. For information on which parameters may be used, see ***List of Parameters*** (in the `WRITE` statement documentation).
*n*X *n*T *x*/*y* T**field-name* P**field-name* /	**Format Notation and Spacing Elements:** See ***Format Notation and Spacing Elements*** (below).
'*text*' '*c*'(*n*) *attributes*	**Text/Attribute Assignments:** See ***Text/Attribute Assignments*** (below).
operand1	**Trailer Information:** *operand1* represents the field/fields to be output as trailer information.
parameters	**Parameter Definition at Element (Field) Level:**

Syntax Element	Description
	One or more parameters, enclosed within parentheses, may be specified at element (field) level, that is, immediately after *operand1*. Each parameter specified in this manner will override the corresponding parameter previously specified **at statement level** or in a GLOBALS command, SET GLOBALS (in Reporting Mode only) or FORMAT statement. If more than one parameter is specified, one or more blanks must be present between each entry. An entry may not be split between two statement lines. For information on which parameters may be used, see ***List of Parameters*** in the WRITE statement documentation.
SKIP *operand2* LINES	**Lines to Be Skipped:** SKIP may be used to cause lines to be skipped immediately after the trailer line. The number of lines to be skipped (*operand2*) may be specified as a numeric constant or as the content of a numeric variable. **Note:** SKIP after WRITE TRAILER is always interpreted as the SKIP clause of the WRITE TRAILER statement, and not as an independent statement. If you wish an independent SKIP statement after a WRITE TRAILER statement, use a semicolon (;) to separate the two statements from one another.

Format Notation and Spacing Elements

Syntax Element	Description
*n*X	**Column Spacing:** This notation inserts *n* spaces between columns.
*n*T	**Tab Setting:** The *n*T notation causes positioning (tabulation) to print position *n*. Backward positioning is not permitted.
x/*y*	***x*/*y* Positioning:** Causes the next element to be placed *x* lines below the output of the last statement, beginning in column *y*. *y* must not be zero. Backward positioning in the same line is not permitted.
*T*field-name*	**Field-Related Positioning:** The *T** notation causes positioning to a *specific print position of a field* used in a previous DISPLAY statement. Backward positioning is not permitted.
*P*field-name*	**Field- and Line-Related Positioning:** The *P** notation causes positioning to a *specific print position and line of a field* used in a previous DISPLAY statement. It is most often used in conjunction with vertical printing mode. Backward positioning is not permitted.
/	**Line Advance - Slash Notation:**

Syntax Element	Description
	When placed between fields or text elements, a slash (/) causes positioning to the beginning of the next print line.

Text/Attribute Assignments

Syntax Element	Description
'text'	**Text Assignment:** The character string enclosed by single quotes is displayed.
'*c*'(*n*)	**Character Repetition:** The character enclosed by single quotes is displayed *n* times immediately before the field value.

attributes	**Field Representation and Color Attributes:**
	
	It is possible to assign various attributes for text/field display. These attributes and the syntax that may be used are described in the section ***Output Attributes*** below.
	
	Examples:
	
	`WRITE TRAILER 'TEXT' (BGR)`
	`WRITE TRAILER 'TEXT' (B)`
	`WRITE TRAILER 'TEXT' (BBLC)`

Output Attributes

attributes indicates the output attributes to be used for text display. Attributes can be:

$$\left\{\begin{array}{l}\texttt{AD=}\textit{ad-value}\ \ldots\\ \texttt{CD=}\textit{cd-value}\\ \texttt{PM=}\textit{pm-value}\ \ldots\end{array}\right\}\ldots$$

$$\left\{\begin{array}{l}\textit{ad-value}\\ \textit{cd-value}\end{array}\right\}\ldots$$

Where:

ad-value, *cd-value* and *pm-value* denote the possible values of the corresponding session parameters AD, CD and PM described in the relevant sections of the *Parameter Reference* documentation.

The compiler actually accepts more than one attribute value for an output field. For example, you can specify: AD=BDI. In such a case, however, only the last value applies. In the given example, only the value I becomes effective and the output field is displayed intensified.

For an alphanumeric/Unicode constant (Natural data format A or U), you can specify *ad-value* and/or *cd-value* without preceding `CD=` or `AD=`, respectively. The single value entered is then checked against all possible `CD` values first. For example: a value of `IRE` will be interpreted as intensified/red but not as intensified/right-justified/mandatory. You cannot combine a single *cd-value* or *ad-value* with a value preceded by `CD=` or `AD=`.

Example

```
** Example 'WTLEX1': WRITE (with TRAILER option)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 JOB-TITLE
END-DEFINE
*
FORMAT PS=15
WRITE TITLE LEFT JUSTIFIED UNDERLINED
      *TIME 3X 'PEOPLE LIVING IN BARCELONA'
      14X 'PAGE:' *PAGE-NUMBER
SKIP 1
*
WRITE TRAILER LEFT JUSTIFIED UNDERLINED
      / 'CITY OF BARCELONA REGISTER'
*
LIMIT 10
FIND EMPL-VIEW WITH CITY = 'BARCELONA'
  DISPLAY NAME FIRST-NAME 3X JOB-TITLE
END-FIND
END
```

Output of Program WTLEX1 - Page 1:

```
09:36:09.5   PEOPLE LIVING IN BARCELONA            PAGE:      1
-------------------------------------------------------------------------------

        NAME              FIRST-NAME                 CURRENT
                                                    POSITION
-------------------- --------------------   -------------------------

DEL CASTILLO         ANGEL                  EJECUTIVO DE VENTAS
GARCIA               M. DE LAS MERCEDES     SECRETARIA
GARCIA               ENDIKA                 DIRECTOR TECNICO
MARTIN               ASUNCION               SECRETARIA
MARTINEZ             TERESA                 SECRETARIA
YNCLAN               FELIPE                 ADMINISTRADOR
```

```
FERNANDEZ            ELOY                   OFICINISTA
TORRES               ANTONI                 OBRERA

CITY OF BARCELONA REGISTER
-------------------------------------------------------------------------------
```

Output of Program WTLEX1 - Page 2:

```
09:37:26.0   PEOPLE LIVING IN BARCELONA              PAGE:      2
-------------------------------------------------------------------------------

        NAME              FIRST-NAME                 CURRENT
                                                    POSITION
-------------------- --------------------   -------------------------

RODRIGUEZ            VICTORIA               SECRETARIA
GARCIA               GERARDO                INGENIERO DE PRODUCCION

CITY OF BARCELONA REGISTER
-------------------------------------------------------------------------------
```

147 WRITE WORK FILE

```
WRITE WORK [FILE] work-file-number [VARIABLE] operand1 ...
```

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

Related Statements: `DEFINE WORK FILE` | `READ WORK FILE` | `CLOSE WORK FILE` | `DOWNLOAD PC FILE`

Belongs to Function Group: ***Control of Work Files / PC Files***

Function

The `WRITE WORK FILE` statement is used to write records to a physical sequential work file.

It is possible to create a work file in one program or processing loop and to read the same file in a subsequent independent processing loop or in a subsequent program using the `READ WORK FILE` statement.

Note: For Unicode and code page support, see *Work Files and Print Files on Windows and Linux Platforms* in the *Unicode and Code Page Support* documentation.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats													Referencing Permitted	Dynamic Definition
operand1	C	S	A	G	N	A	U	N	P	I	F	B	D	T	L	C	G		yes	no

Note: When using the work file types ENTIRECONNECTION or TRANSFER, *operand1* may neither be of format C, nor G.

Syntax Element Description:

Syntax Element	Description
work-file-number	**Work File Number:** The work file number (as defined to Natural) to be used. The work file number is either ■ a numeric constant in the value range 1:32 or ■ a numeric variable of type (B/N/P/I) defined with a `CONST` clause which assigning a value in range (1:32). Variable is a scalar (non-array) without precision digits for type (N/P), length in between 1-4 for type (B), and no redefinition field.

<table>
<tr><th>Syntax Element</th><th>Description</th></tr>
<tr><td><code>VARIABLE</code></td><td> Variable Entry:
It is possible to write records with different fields to the same work file with different <code>WRITE WORK FILE</code> statements. In this case, the <code>VARIABLE</code> entry must be specified in all <code>WRITE WORK FILE</code> statements. The records on the external file will be written in variable format.
When the operand list includes a dynamic variable (that could change in size for different executions of the <code>WRITE WORK FILE</code> statement), the <code>VARIABLE</code> entry must be specified in all <code>WRITE WORK FILE</code> statements.
Variable Index Range:
When writing an array to a work file, you can specify a variable index range for the array. For example:
<pre>WRITE WORK FILE work-file-number VARIABLE #ARRAY (I:J)</pre></td></tr>
<tr><td>operand1</td><td> Fields to Be Written:
With operand1 you specify the fields to be written to the work file. These fields may be database fields, user-defined variables, system variables and/or fields read from another work file using the <code>READ WORK FILE</code> statement.
An array may be referenced completely or partially to select the occurrences that are to be written to the work file.
Group Operands to be Written:
A group may be referenced using the group name. All fields belonging to the referenced group will be written to the work file, the sequence is determined by the sequence of the fields in the group. Fields resulting from a redefinition of the referenced group are not written to the work file. If the referenced group is defined as an array, the individual fields of the group are written to the work file as arrays in the definition sequence.
For the group definition
<pre>1 GROUP1 (1:3)
    2 FIELD1 (A2)
    2 FIELD2 (A3)
  1 REDEFINE GROUP1
    2 FIELD3 (A15)</pre>
the statement </td></tr>
</table>

Syntax Element	Description

	`WRITE WORK FILE 1 GROUP1(*)`
	is equivalent to
	`WRITE WORK FILE 1 GROUP1.FIELD1(*) GROUP1.FIELD2(*)`
	The statement
	`WRITE WORK FILE 1 GROUP1.FIELD3`
	is equivalent to
	`WRITE WORK FILE 1 GROUP1.FIELD1(1) GROUP1.FIELD2(1)`
	`GROUP1.FIELD1(2) GROUP1.FIELD2(2)`
	`GROUP1.FIELD1(3) GROUP1.FIELD2(3)`

External Representation of Fields

Fields written with a `WRITE WORK FILE` statement are represented in the external file according to their internal definition. No editing is performed on the field values.

For fields of format A and B, the number of bytes in the external file is the same as the internal length definition as defined in the Natural program. No editing is performed and a decimal point is not represented in the value.

For fields of format N, the number of bytes on the external file is the sum of internal positions before and after the decimal point. The decimal point is not represented on the external file.

For fields of format P, the number of bytes on the external file is the sum of positions before and after the decimal point, plus 1 for the sign, divided by 2, rounded upward to a full byte.

Note: No format conversion is performed for fields that are written to a work file.

Examples of field representations:

Field Definition	Output Record
`#FIELD1 (A10)`	10 bytes
`#FIELD2 (B15)`	15 bytes
`#FIELD3 (N1.3)`	4 bytes
`#FIELD4 (N0.7)`	7 bytes
`#FIELD5 (P1.2)`	2 bytes
`#FIELD6 (P6.0)`	4 bytes

Special Considerations for System Functions

For the special considerations that apply when `WRITE WORK FILE` is used for the Natural system function `AVER`, `NAVER`, `SUM` or `TOTAL`, see *Format/Length Requirements for AVER, NAVER, SUM and TOTAL* in the *System Functions* documentation.

Handling of Large and Dynamic Variables

Work File Type	Handling
`ASCII` `ASCII-COMPRESSED`	The work file types `ASCII` and `ASCII-COMPRESSED` can handle dynamic and large variables with a maximum field/record length of 32766 bytes.
`SAG` (binary)	The work file type `SAG` (binary) cannot handle dynamic variables and will produce an error. It can, however, handle large variables with a maximum field/record length of 32766 bytes.
`TRANSFER` `ENTIRECONNECTION`	The work file type `TRANSFER` can handle dynamic variables with a maximum field/record length of 32766 bytes. The work file type `ENTIRECONNECTION` cannot handle dynamic variables. They can both, however, handle large variables with a maximum field/record length of 1073741824 bytes.
`PORTABLE` `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. In the `WRITE WORK FILE` statement, fields are written to the file specified 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 FILE` 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).
`CSV`	The maximum field/record length is 32766 bytes for dynamic and large variables. Dynamic variables are supported. X-arrays are not allowed and will result in an error message.

Example

```
** Example 'WWFEX1': WRITE WORK FILE
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
END-DEFINE
*
FIND EMPLOY-VIEW WITH CITY = 'LONDON'
  WRITE WORK FILE 1
        PERSONNEL-ID NAME
END-FIND
*
END
```