

Accessing Data in an Adabas Database

This chapter describes various aspects of accessing data in an Adabas database with Natural.

The following topics are covered:

- Adabas Database Management Interfaces ADA and ADA2
 - Data Definition Modules - DDMs
 - Database Arrays
 - Defining a Database View
 - Statements for Database Access
 - Multi-Fetch Clause
 - Database Processing Loops
 - Database Update - Transaction Processing
 - Selecting Records Using ACCEPT/REJECT
 - AT START/END OF DATA Statements
 - Unicode Data
-

Adabas Database Management Interfaces ADA and ADA2

Natural's Adabas database management interfaces ADA and ADA2 are considered distinct database interfaces like, for example, ADA and SQL.

Database type ADA is Natural's default interface to Adabas databases. It is the appropriate choice if new Adabas functionality as introduced with Adabas Version 6 on Open Systems and Adabas Version 8 on mainframes is not concerned.

Database type ADA2 is provided as an extended interface to Adabas databases as of Version 6 on Open Systems and Adabas as of Version 8 on mainframes. In particular, it supports Adabas LA fields, Adabas large object fields and extended Adabas buffer lengths. The support of Adabas LA and large object fields implies the use of Natural format (A) DYNAMIC in a view definition, the support of extended Adabas buffer lengths enables the definition of view sizes that exceed 64 KB. For further information, refer to *Defining a Database View*.

Database type ADA2 does not support multi-fetch processing. Corresponding global and local definitions are ignored at runtime.

Software AG products which have their own system files require a corresponding physical database of database type ADA.

Natural objects that were compiled with database type ADA can be executed in an environment where the corresponding Adabas database is defined as database type ADA2.

Data Definition Modules - DDMs

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

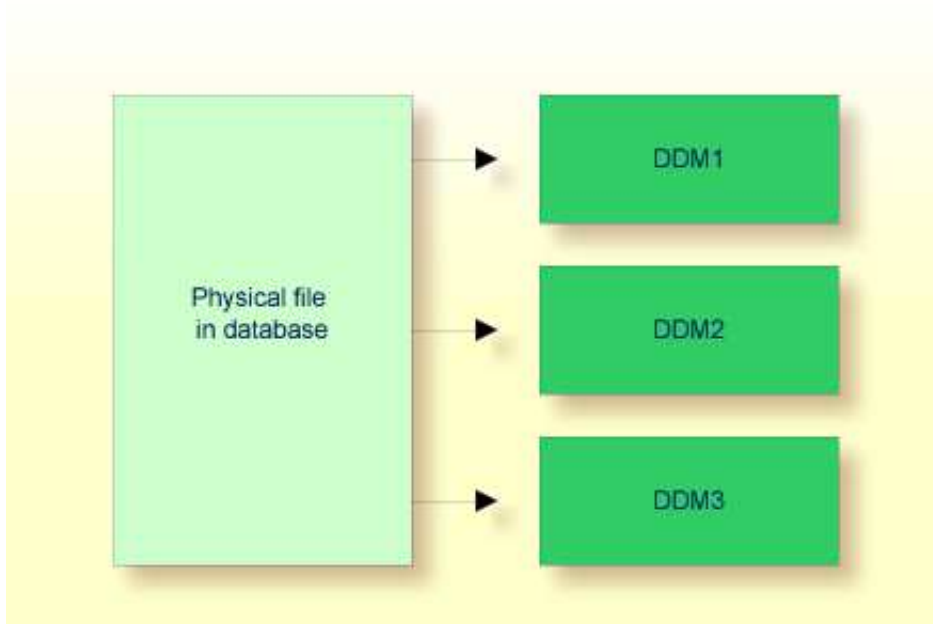
This section covers the following topics:

- Use of Data Definition Modules
- Maintaining DDMs
- Listing/Displaying DDMs

Use of Data Definition Modules

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

For each physical file of a database, one or more DDMs can be defined. And for each DDM one or more data views can be defined as described *View Definition* in the `DEFINE DATA` statement documentation and explained in the section *Defining a Database View*.



DDMs are defined by the Natural administrator with Predict (or, if Predict is not available, with the corresponding Natural function).

Maintaining DDMs

Use the system command `SYSDDM` to invoke the `SYSDDM` utility. The `SYSDDM` utility is used to perform all functions needed for the creation and maintenance of Natural data definition modules.

For further information on the `SYSDDM` utility, see the section *DDM Services* in the *Editors* documentation.

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

For the field attributes defined in a DDM, refer to *Using the DDM Editor Screen* in the section *DDM Services* of the *Editors* documentation.

Listing/Displaying DDMs

Database Arrays

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

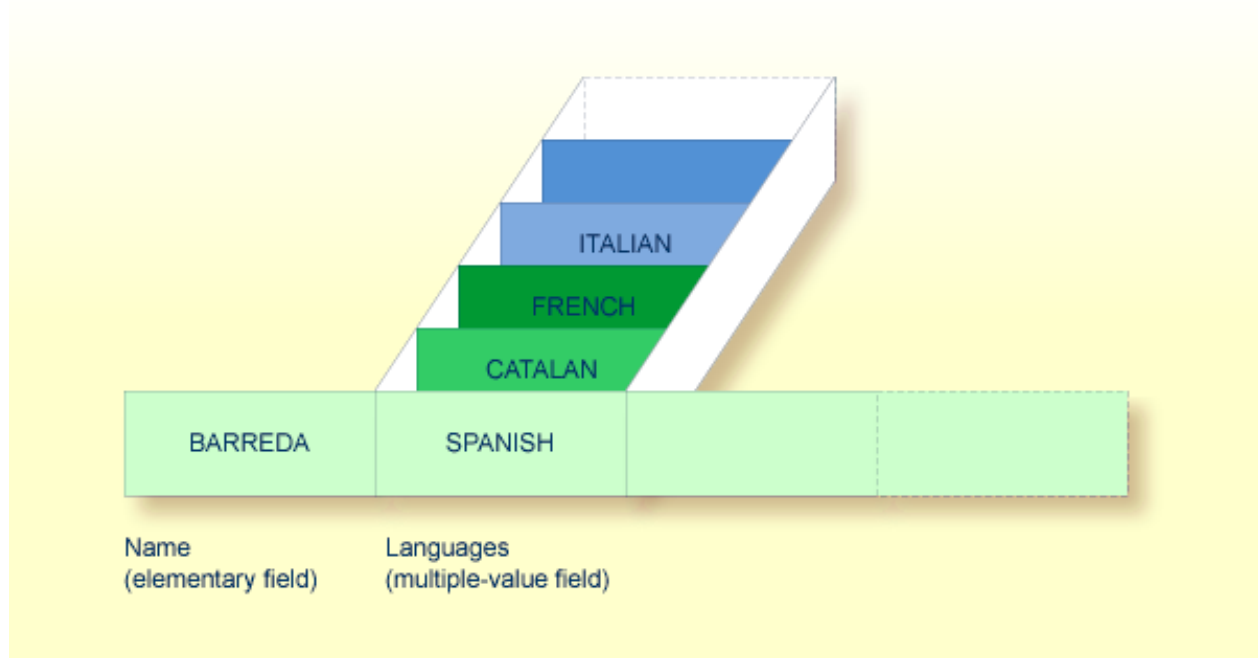
This section covers the following topics:

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

Multiple-Value Fields

A multiple-value field is a field which can have more than one value (up to 65534, depending on the Adabas version and definition of the FDT) within a given record.

Example:



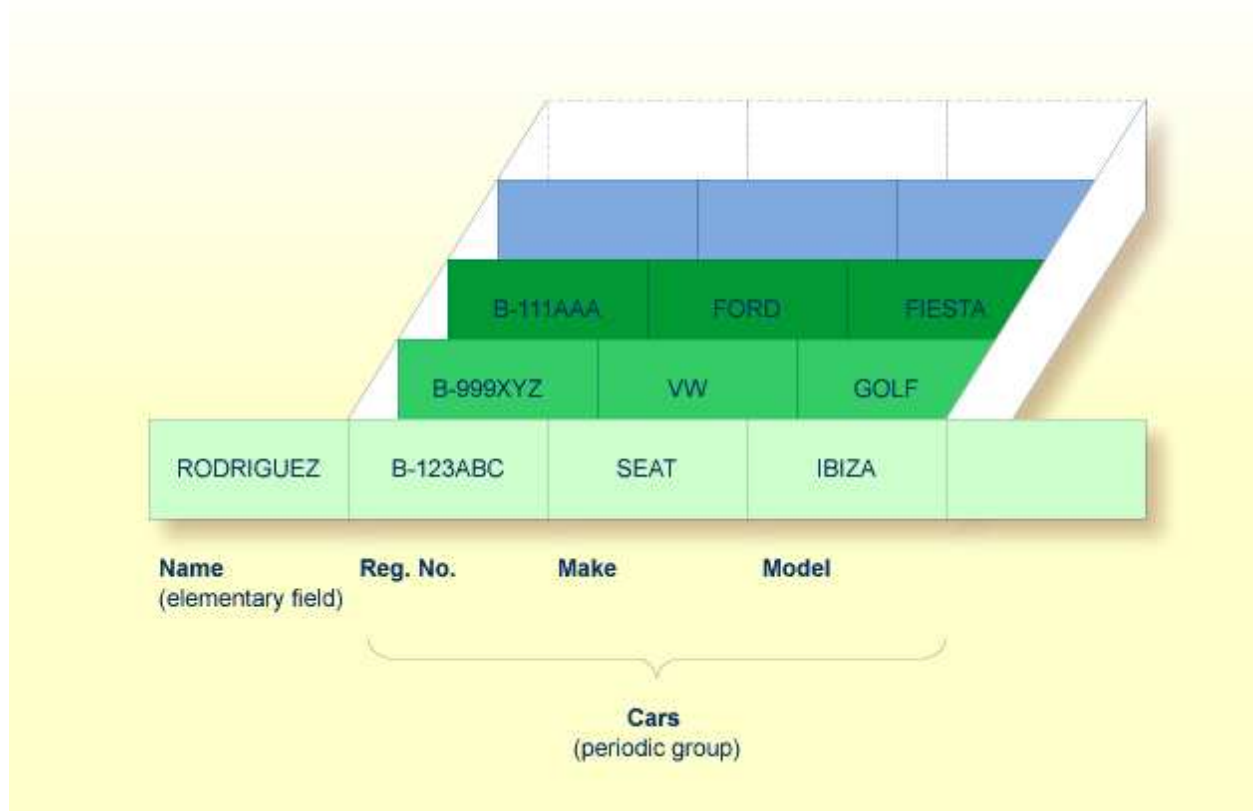
Assuming that the above is a record in an employees file, the first field (Name) is an elementary field, which can contain only one value, namely the name of the person; whereas the second field (Languages), which contains the languages spoken by the person, is a multiple-value field, as a person can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

Periodic Groups

A periodic group is a group of fields (which may be elementary fields and/or multiple-value fields) that may have more than one occurrence (up to 65534, depending on the Adabas version and definition of the field definition table (FDT)) within a given record.

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

Example:



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

Referencing Multiple-Value Fields and Periodic Groups

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

Examples:

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

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

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

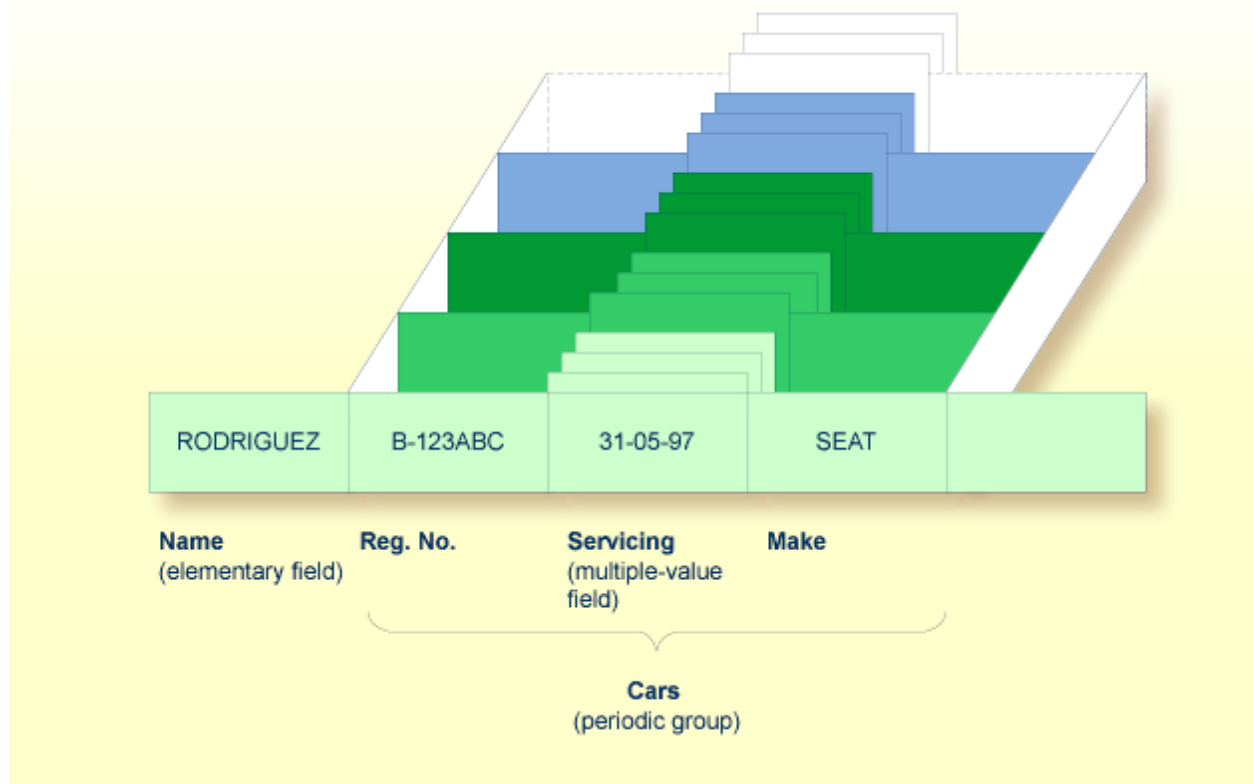
The various occurrences of the periodic group CARS can be referenced in the same manner:

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

Multiple-Value Fields within Periodic Groups

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

Example:



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

Referencing Multiple-Value Fields within Periodic Groups

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

Examples:

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

Example	Explanation
<code>SERVICING (1,1)</code>	References the first value of <code>SERVICING</code> in the first occurrence of <code>CARS</code> (31-05-97).
<code>SERVICING (1:5,1)</code>	References the first value of <code>SERVICING</code> in the first five occurrences of <code>CARS</code> .
<code>SERVICING (1:5,1:10)</code>	References the first ten values of <code>SERVICING</code> in the first five occurrences of <code>CARS</code> .

Referencing the Internal Count of a Database Array

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

The count is returned in format/length N3. See *Referencing the Internal Count for a Database Array* for further details.

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

Defining a Database View

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

In the view, you specify the name of the data definition module (see *Data Definition Modules - DDMs*) from which the fields are to be taken, and the names of the database fields themselves (that is, their long names, not their database-internal short names).

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

As described in the section *Statements for Database Access*, the view name is used in the statements READ, FIND, HISTOGRAM to determine which database is to be accessed.

For further information on the complete syntax of the view definition option or on the definition/redefinition of a group of fields, see *View Definition* in the description of the DEFINE DATA statement in the *Statements* documentation.


Basically, you have the following options to define a database view:

- **Inside the Program**

You can define a database view inside the program, that is, directly within the DEFINE DATA statement of the program.

- **Outside the Program**

You can define a database view outside the program, that is, in a separate programming object: either a local data area (LDA) or a global data area (GDA), with the DEFINE DATA statement of the program referencing that data area.

 **To define a database view inside the program**

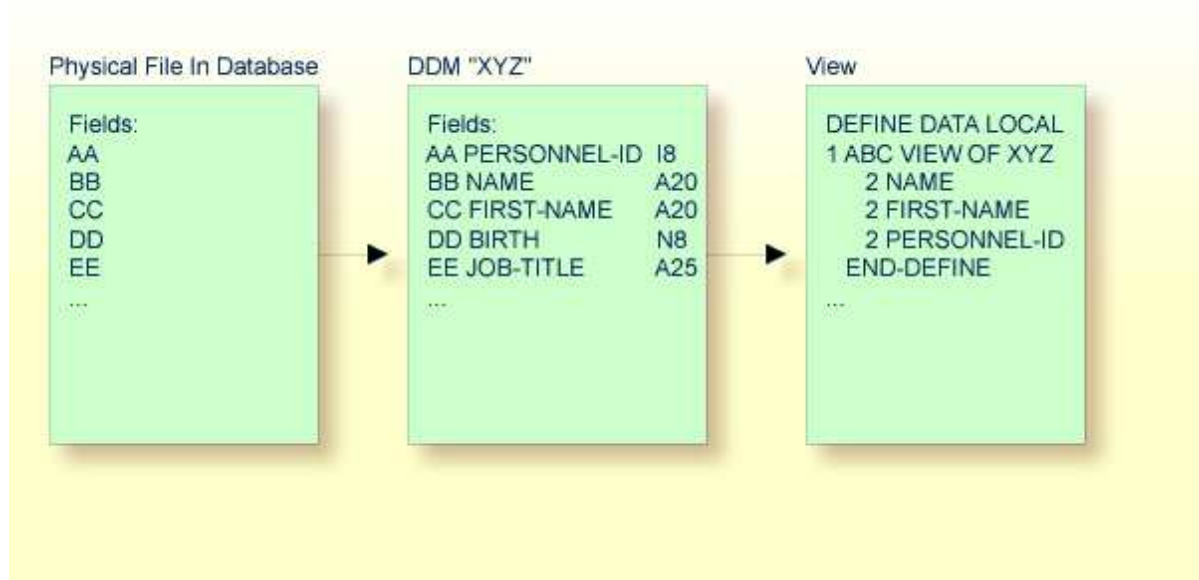
- At Level 1, specify the view name as follows:

```
1 view-name VIEW OF ddm-name
```

where *view-name* is the name you choose for the view, *ddm-name* is the name of the DDM from which the fields specified in the view are taken.

- At Level 2, specify the names of the database fields from the DDM.

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



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

Sample Program:

In this example, the *view-name* is VIEWEMP, and the *ddm-name* is EMPLOYEES, and the names of the database fields taken from the DDM are NAME, FIRST-NAME and PERSONNEL-ID.

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

▶ To define a database view outside the program

- In the program, specify:

```
DEFINE DATA LOCAL
    USING <data-area-name>
END-DEFINE
    . . .
```

where *data-area-name* is the name you choose for the local or global data area, for example, LDA39.

2. In the data area to be referenced:
 1. At Level 1 in the Name column, specify the name you choose for the view, and in the Miscellaneous column, the name of the DDM from which the fields specified in the view are taken.
 2. At Level 2, specify the names of the database fields from the DDM.

Example LDA39:

In this example, the view name is VIEWEMP, the DDM name is EMPLOYEES, and the names of the database fields taken from the DDM are PERSONNEL-ID, FIRST-NAME and NAME.

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

Considerations Concerning Databases of Type ADA2

With databases of type ADA2 (specified in the table DBMS Assignments in the Configuration Utility, see *Database Management System Assignments* in the *Configuration Utility* documentation), the following applies:

- If large alphanumeric (LA) or large object (LOB) fields (Adabas LA/LB option) are to be used, these fields can be specified within the view definition with both fixed format/length, for example, A20 or U20, and dynamic format/length, for example, (A) DYNAMIC or U(DYNAMIC).
- Length indicator fields L@. . . can also be specified within views if they are related to LA or LOB fields.

Statements for Database Access

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

Statement	Meaning
READ	Select a range of records from a database in a specified sequence.
FIND	Select from a database those records which meet a specified search criterion.
HISTOGRAM	Read only the values of one database field, or determine the number of records which meet a specified search criterion.

READ Statement

The following topics are covered:

- Use of READ Statement
- Basic Syntax of READ Statement
- Example of READ Statement
- Limiting the Number of Records to be Read
- STARTING/ENDING Clauses
- WHERE Clause
- Further Example of READ Statement

Use of READ Statement

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

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

In this document, only READ IN LOGICAL SEQUENCE is discussed, as it is the most frequently used form of the READ statement.

For information on the other two options, please refer to the description of the READ statement in the *Statements* documentation.

Basic Syntax of READ Statement

The basic syntax of the READ statement is:

```
READ view IN LOGICAL SEQUENCE BY descriptor
```

or shorter:

```
READ view LOGICAL BY descriptor
```

- where

<i>view</i>	is the name of a view defined in the DEFINE DATA statement and as explained in <i>Defining a Database View</i> .
<i>descriptor</i>	is the name of a database field defined in that view. The values of this field determine the order in which the records are read from the database.

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

```
READ view BY descriptor
```

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

```
READ view LOGICAL
```

Example of READ Statement

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

Output of Program READX01:

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

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

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

NAME	PERSONNEL ID	CURRENT POSITION
ABELLAN	60008339	MAQUINISTA
ACHIESON	30000231	DATA BASE ADMINISTRATOR
ADAM	50005800	CHEF DE SERVICE
ADKINSON	20008800	PROGRAMMER
ADKINSON	20009800	DBA
ADKINSON	2001100	

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

```
READ MYVIEW BY BIRTH
```

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

Limiting the Number of Records to be Read

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

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

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

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

STARTING/ENDING Clauses

The READ statement also allows you to qualify the selection of records based on the *value* of a descriptor field. With an EQUAL TO/STARTING FROM option in the BY clause, you can specify the value at which reading should begin. (Instead of using the keyword BY, you may specify the keyword WITH, which would have the same effect). By adding a THRU/ENDING AT option, you can also specify the value in the logical sequence at which reading should end.

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

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

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

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

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

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

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

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

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

WHERE Clause

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

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

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

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

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

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

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

The following logical operators are possible in a WHERE clause:

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

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

```
** Example 'READX02': READ (with STARTING, ENDING and WHERE clause)
*****
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 INCOME      (1:2)
    3 CURR-CODE
    3 SALARY
  3 BONUS      (1:1)
END-DEFINE
*
READ (3) MYVIEW WITH JOB-TITLE
STARTING FROM 'TRAINEE' ENDING AT 'TRAINEE'
                WHERE CURR-CODE (*) = 'USD'
```

```

DISPLAY NOTITLE NAME / JOB-TITLE 5X INCOME (1:2)
SKIP 1
END-READ
END
    
```

Output of Program READX02:

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

Further Example of READ Statement

See the following example program:

- *READX03 - READ statement*

FIND Statement

The following topics are covered:

- Use of FIND Statement
- Basic Syntax of FIND Statement
- Limiting the Number of Records to be Processed
- WHERE Clause
- Example of FIND Statement with WHERE Clause
- IF NO RECORDS FOUND Condition
- Further Examples of FIND Statement

Use of FIND Statement

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

Basic Syntax of FIND Statement

The basic syntax of the FIND statement is:

```

FIND RECORDS IN view WITH field = value
    
```

or shorter:

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

- where

<i>view</i>	is the name of a view as defined in the DEFINE DATA statement and as explained in <i>Defining a Database View</i> .
<i>field</i>	is the name of a database field as defined in that view.

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

For the complete syntax, refer to the FIND statement documentation.

Limiting the Number of Records to be Processed

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

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

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

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

Note:

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

WHERE Clause

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

Example of FIND Statement with WHERE Clause

```
** Example 'FINDX01': FIND (with WHERE)
*****
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 CITY
END-DEFINE
*
FIND MYVIEW WITH CITY = 'PARIS'
      WHERE JOB-TITLE = 'INGENIEUR COMMERCIAL'
      DISPLAY NOTITLE CITY JOB-TITLE PERSONNEL-ID NAME
END-FIND
END
```


Note:

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

Output of Program FINDX01:

CITY	CURRENT POSITION	PERSONNEL ID	NAME
PARIS	INGENIEUR COMMERCIAL	50007300	CAHN
PARIS	INGENIEUR COMMERCIAL	50006500	MAZUY
PARIS	INGENIEUR COMMERCIAL	50004700	FAURIE
PARIS	INGENIEUR COMMERCIAL	50004400	VALLY
PARIS	INGENIEUR COMMERCIAL	50002800	BRETON
PARIS	INGENIEUR COMMERCIAL	50001000	GIGLEUX
PARIS	INGENIEUR COMMERCIAL	50000400	KORAB-BRZOZOWSKI

IF NO RECORDS FOUND Condition

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

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

Example:

```

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

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

Output of Program FINDX02:

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

NAME	FIRST-NAME
NO PERSON FOUND.	

Further Examples of FIND Statement

See the following example programs:

- *FINDX07 - FIND (with several clauses)*
- *FINDX08 - FIND (with LIMIT)*
- *FINDX09 - FIND (using *NUMBER, *COUNTER, *ISN)*
- *FINDX10 - FIND (combined with READ)*
- *FINDX11 - FIND NUMBER (with *NUMBER)*

HISTOGRAM Statement

The following topics are covered:

- Use of HISTOGRAM Statement
- Syntax of HISTOGRAM Statement
- Limiting the Number of Values to be Read
- STARTING/ENDING Clauses
- WHERE Clause
- Example of HISTOGRAM Statement

Use of HISTOGRAM Statement

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

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

Syntax of HISTOGRAM Statement

The basic syntax of the HISTOGRAM statement is:

HISTOGRAM VALUE IN <i>view</i> FOR <i>field</i>
--

or shorter:

HISTOGRAM <i>view</i> FOR <i>field</i>

- where

<i>view</i>	is the name of a view as defined in the DEFINE DATA statement and as explained in <i>Defining a Database View</i> .
<i>field</i>	is the name of a database field as defined in that view.

For the complete syntax, refer to the HISTOGRAM statement documentation.

Limiting the Number of Values to be Read

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

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

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

Without the limit notation, all values would be read.

STARTING/ENDING Clauses

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

Examples:

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

WHERE Clause

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

Example of HISTOGRAM Statement

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

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

Output of Program HISTOX01:

CITY	NUMBER OF PERSONS	CNT
MADISON	3	1
MADRID	41	2
MAILLY LE CAMP	1	3
MAMERS	1	4
MANSFIELD	4	5
MARSEILLE	2	6
MATLOCK	1	7
MELBOURNE	2	8

Multi-Fetch Clause

This section covers the multi-fetch record retrieval functionality for Adabas databases.

The multi-fetch functionality described in this section is only supported for databases of type ADA, which can be defined in the DBMS Assignments table in the Configuration Utility; see *Database Management System Assignments* in the *Configuration Utility* documentation. With database type ADA2, the multi-fetch clause is not supported.

The following topics are covered:

- Purpose of Multi-Fetch Feature
- Statements Supported
- Considerations for Multi-Fetch Usage

Purpose of Multi-Fetch Feature

In standard mode, Natural does not read multiple records with a single database call; it always operates in a one-record-per-fetch mode. This kind of operation is solid and stable, but can take some time if a large number of database records are being processed. To improve the performance of those programs, you can use multi-fetch processing.

By default, Natural uses single-fetch to retrieve data from Adabas databases. This default can be configured using the Natural profile parameter MFSET.

Values ON (multi-fetch) and OFF (single-fetch) define the default behavior. If MFSET is set to NEVER, Natural always uses single-fetch mode and ignores any settings at statement level.

The default processing mode can also be overridden at statement level.

Statements Supported

Multi-fetch processing is supported for the following statements that do not involve database modification:

- FIND

- READ
- HISTOGRAM

For more information on the syntax, see the description of the `MULTI-FETCH` clause of the `FIND`, `READ` or `HISTOGRAM` statements.

Considerations for Multi-Fetch Usage

If nested database loops that refer to the same Adabas file contain `UPDATE` statements in one of the inner loops, Natural continues processing the outer loops with the updated values. This implies in multi-fetch mode, that an outer logical `READ` loop has to be repositioned if an inner database loop updates the value of the descriptor that is used for sequence control in the outer loop. If this attempt leads to a conflict for the current descriptor, an error is returned. To avoid this situation, we recommend that you disable multi-fetch in the outer database loops.

In general, multi-fetch mode improves performance when accessing Adabas databases. In some cases, however, it might be advantageous to use single-fetch to enhance performance, especially if database modifications are involved.

Database Processing Loops

This section discusses processing loops required to process data that have been selected from a database as a result of a `FIND`, `READ` or `HISTOGRAM` statement.

The following topics are covered:

- Creation of Database Processing Loops
- Hierarchies of Processing Loops
- Example of Nested `FIND` Loops Accessing the Same File
- Further Examples of Nested `READ` and `FIND` Statements

Creation of Database Processing Loops

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

Example:

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

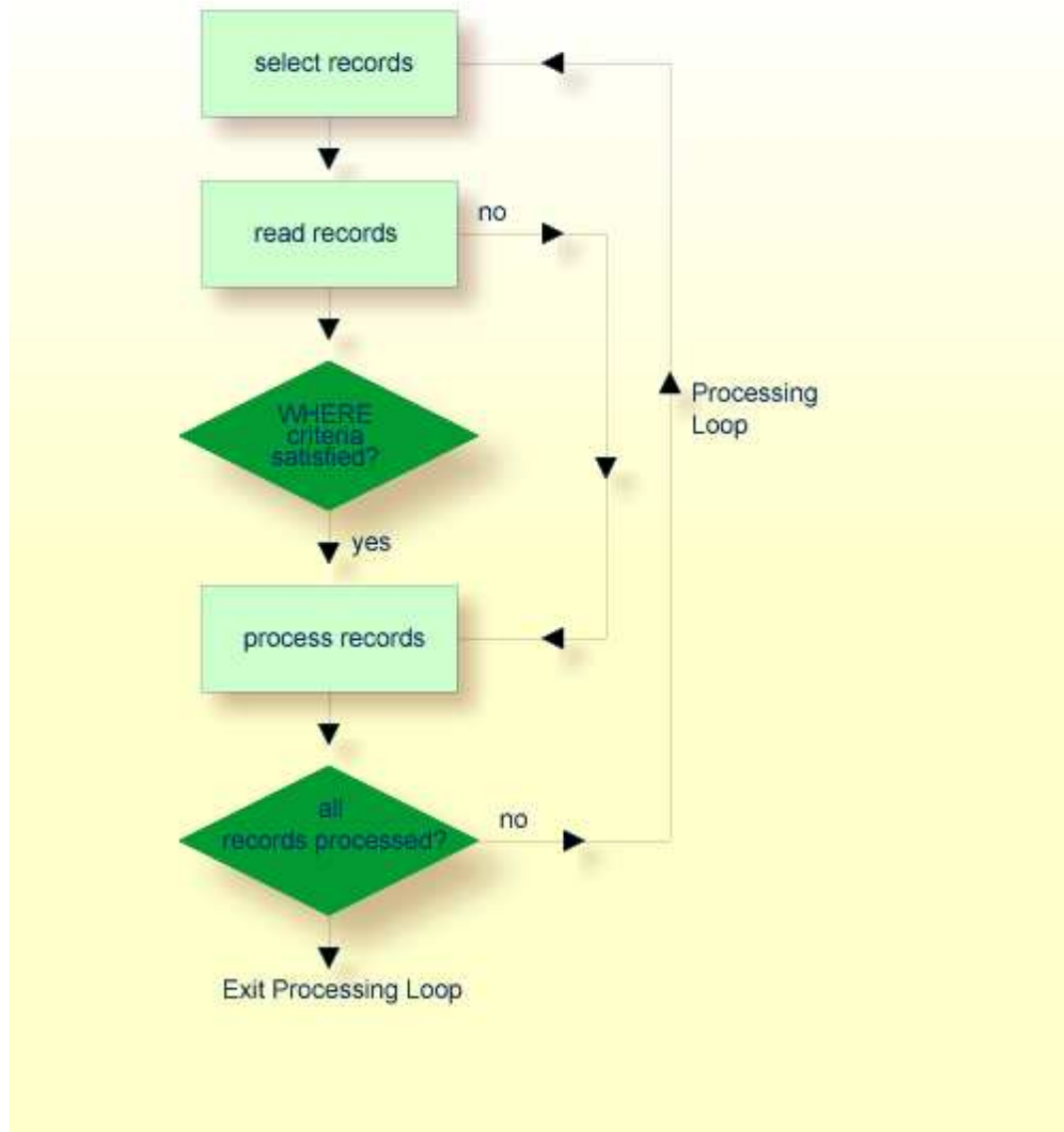
```
** Example 'FINDX03': FIND
*****
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
END-DEFINE
```

```

*
FIND MYVIEW WITH NAME = 'ADKINSON'
  DISPLAY NAME FIRST-NAME CITY
END-FIND
END
    
```

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

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



Hierarchies of Processing Loops

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

Example of Processing Loop Hierarchy

```

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

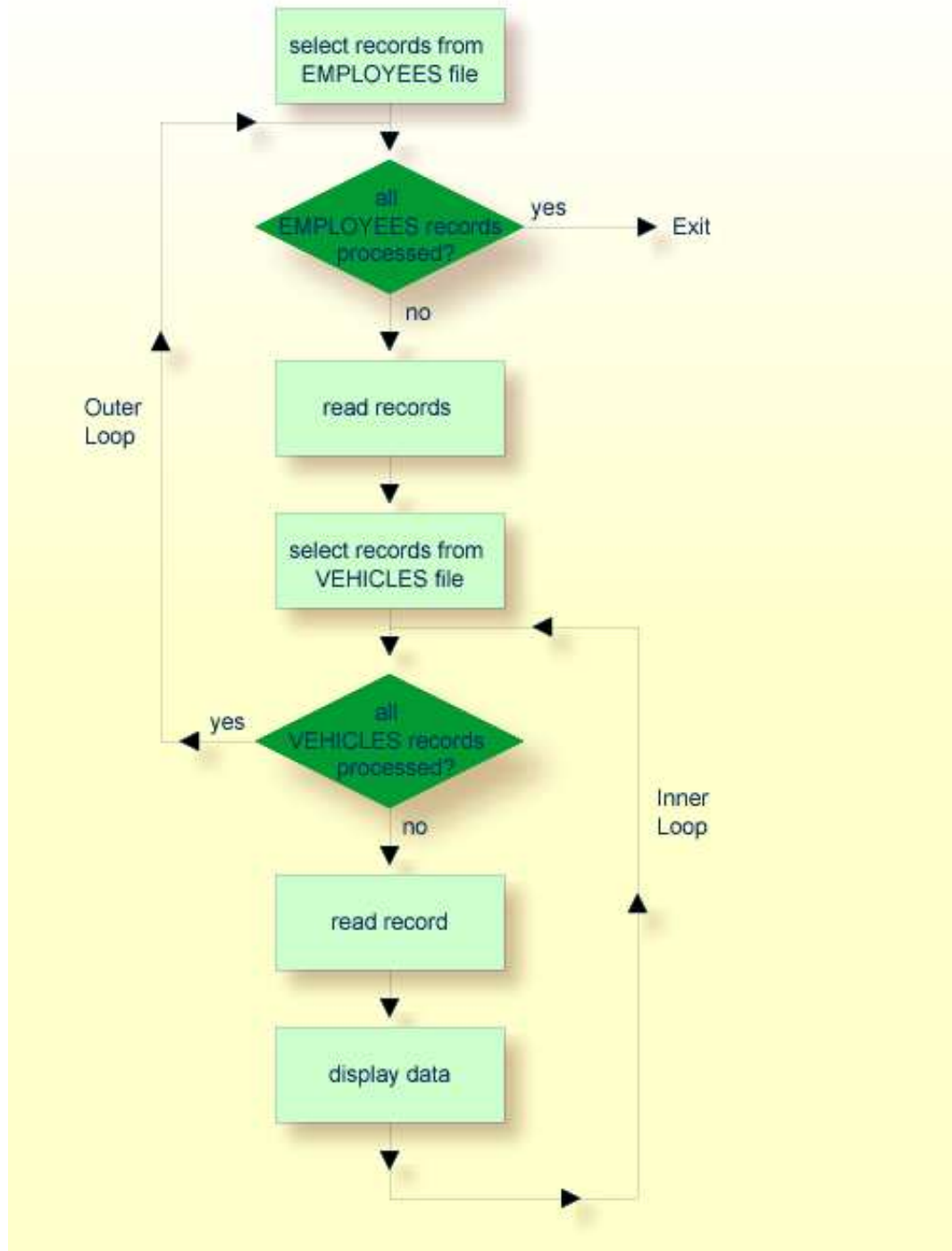
```

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

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

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

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



Example of Nested FIND Loops Accessing the Same File

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

```

** Example 'FINDX05': FIND (two FIND statements on same file nested)
*****
DEFINE DATA LOCAL
1 PERSONVIEW VIEW OF EMPLOYEES
  2 NAME
  
```



```

      2 FIRST-NAME
      2 CITY
1 #NAME (A40)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED
      'PEOPLE IN SAME CITY AS:' #NAME / 'CITY:' CITY SKIP 1
*
FIND PERSONVIEW WITH NAME = 'JONES'
      WHERE FIRST-NAME = 'LAUREL'
      COMPRESS NAME FIRST-NAME INTO #NAME
/*
      FIND PERSONVIEW WITH CITY = CITY
      DISPLAY NAME FIRST-NAME CITY
      END-FIND
END-FIND
END

```

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

Output of Program FINDX05:

```

PEOPLE IN SAME CITY AS: JONES LAUREL
CITY: BALTIMORE

```

NAME	FIRST-NAME	CITY
JENSON	MARTHA	BALTIMORE
LAWLER	EDDIE	BALTIMORE
FORREST	CLARA	BALTIMORE
ALEXANDER	GIL	BALTIMORE
NEEDHAM	SUNNY	BALTIMORE
ZINN	CARLOS	BALTIMORE
JONES	LAUREL	BALTIMORE

Further Examples of Nested READ and FIND Statements

See the following example programs:

- *READX04 - READ statement (in combination with FIND and the system variables *NUMBER and *COUNTER)*
- *LIMITX01 - LIMIT statement (for READ, FIND loop processing)*

Database Update - Transaction Processing

This section describes how Natural performs database updating operations based on transactions.

The following topics are covered:

- Logical Transaction

- Record Hold Logic
- Backing Out a Transaction
- Restarting a Transaction
- Example of Using Transaction Data to Restart a Transaction

Logical Transaction

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

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

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

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

Example:

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

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

Note:

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

Example of STORE Statement:

The following example program adds new records to the EMPLOYEES file.

```
** Example 'STOREX01': STORE (Add new records to EMPLOYEES file)
*
** CAUTION: Executing this example will modify the database records!
*****
DEFINE DATA LOCAL
1 EMPLOYEE-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID(A8)
```

```

2 NAME          (A20)
2 FIRST-NAME    (A20)
2 MIDDLE-I      (A1)
2 SALARY        (P9/2)
2 MAR-STAT      (A1)
2 BIRTH         (D)
2 CITY          (A20)
2 COUNTRY       (A3)
*
1 #PERSONNEL-ID (A8)
1 #NAME         (A20)
1 #FIRST-NAME   (A20)
1 #INITIAL      (A1)
1 #MAR-STAT     (A1)
1 #SALARY       (N9)
1 #BIRTH        (A8)
1 #CITY         (A20)
1 #COUNTRY      (A3)
1 #CONF         (A1)   INIT <'Y'>
END-DEFINE
*
REPEAT
  INPUT 'ENTER A PERSONNEL ID AND NAME (OR ''END'' TO END)' //
        'PERSONNEL-ID : ' #PERSONNEL-ID //
        'NAME          : ' #NAME /
        'FIRST-NAME    : ' #FIRST-NAME
  /******
  /* validate entered data
  /******
  IF #PERSONNEL-ID = 'END' OR #NAME = 'END'
    STOP
  END-IF
  IF #NAME = ' '
    REINPUT WITH TEXT 'ENTER A LAST-NAME'
              MARK 2 AND SOUND ALARM
  END-IF
  IF #FIRST-NAME = ' '
    REINPUT WITH TEXT 'ENTER A FIRST-NAME'
              MARK 3 AND SOUND ALARM
  END-IF
  /******
  /* ensure person is not already on file
  /******
  FIP2. FIND NUMBER EMPLOYEE-VIEW WITH PERSONNEL-ID = #PERSONNEL-ID
  /*
  IF *NUMBER (FIP2.) > 0
    REINPUT 'PERSON WITH SAME PERSONNEL-ID ALREADY EXISTS'
              MARK 1 AND SOUND ALARM
  END-IF
  /******
  /* get further information
  /******
  INPUT
    'ENTER EMPLOYEE DATA' // //
    'PERSONNEL-ID          : ' #PERSONNEL-ID (AD=IO) /
    'NAME                  : ' #NAME (AD=IO) /
    'FIRST-NAME            : ' #FIRST-NAME (AD=IO) //
    'INITIAL               : ' #INITIAL /
    'ANNUAL SALARY         : ' #SALARY /
    'MARITAL STATUS        : ' #MAR-STAT /
    'DATE OF BIRTH (YYYYMMDD) : ' #BIRTH /
    'CITY                  : ' #CITY /

```

```

' COUNTRY (3 CHARS)          : ' #COUNTRY          //
' ADD THIS RECORD (Y/N)     : ' #CONF            (AD=M)
/*****
/* ENSURE REQUIRED FIELDS CONTAIN VALID DATA
*****/
IF #SALARY < 10000
    REINPUT TEXT 'ENTER A PROPER ANNUAL SALARY' MARK 2
END-IF
IF NOT (#MAR-STAT = 'S' OR = 'M' OR = 'D' OR = 'W')
    REINPUT TEXT 'ENTER VALID MARITAL STATUS S=SINGLE ' -
                'M=MARRIED D=DIVORCED W=WIDOWED' MARK 3
END-IF
IF NOT(#BIRTH = MASK(YYYYMMDD) AND #BIRTH = MASK(1582-2699))
    REINPUT TEXT 'ENTER CORRECT DATE' MARK 4
END-IF
IF #CITY = ' '
    REINPUT TEXT 'ENTER A CITY NAME' MARK 5
END-IF
IF #COUNTRY = ' '
    REINPUT TEXT 'ENTER A COUNTRY CODE' MARK 6
END-IF
IF NOT (#CONF = 'N' OR= 'Y')
    REINPUT TEXT 'ENTER Y (YES) OR N (NO)' MARK 7
END-IF
IF #CONF = 'N'
    ESCAPE TOP
END-IF
/*****
/* add the record with STORE
*****/
MOVE #PERSONNEL-ID TO EMPLOYEE-VIEW.PERSONNEL-ID
MOVE #NAME          TO EMPLOYEE-VIEW.NAME
MOVE #FIRST-NAME    TO EMPLOYEE-VIEW.FIRST-NAME
MOVE #INITIAL       TO EMPLOYEE-VIEW.MIDDLE-I
MOVE #SALARY        TO EMPLOYEE-VIEW.SALARY (1)
MOVE #MAR-STAT      TO EMPLOYEE-VIEW.MAR-STAT
MOVE EDITED #BIRTH TO EMPLOYEE-VIEW.BIRTH (EM=YYYYMMDD)
MOVE #CITY          TO EMPLOYEE-VIEW.CITY
MOVE #COUNTRY       TO EMPLOYEE-VIEW.COUNTRY
/*
STP3. STORE RECORD IN FILE EMPLOYEE-VIEW
/*
/*****
/* mark end of logical transaction
*****/
END OF TRANSACTION
RESET INITIAL #CONF
END-REPEAT
END

```

Output of Program STOREX01:

```

ENTER A PERSONNEL ID AND NAME (OR 'END' TO END)

PERSONNEL ID :

NAME          :
FIRST NAME    :

```

Record Hold Logic

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

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

To prevent users from being placed in wait status, the session parameter `WH` (Wait for Record in Hold Status) can be used (see the *Parameter Reference*).

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

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

Example of Hold Logic:

```
** Example 'GETX01': GET (put single record in hold with UPDATE stmt)
**
** CAUTION: Executing this example will modify the database records!
*****
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 SALARY (1)
END-DEFINE
*
RD. READ EMPLOY-VIEW BY NAME
  DISPLAY EMPLOY-VIEW
  IF SALARY (1) > 1500000
    /*
    GE. GET EMPLOY-VIEW *ISN (RD.)
    /*
    WRITE '=' (50) 'RECORD IN HOLD:' *ISN(RD.)
    COMPUTE SALARY (1) = SALARY (1) * 1.15
    UPDATE (GE.)
    END TRANSACTION
  END-IF
END-READ
END
```

Backing Out a Transaction

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

Restarting a Transaction

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

Example of Using Transaction Data to Restart a Transaction

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

```
** Example 'GETTRX01': GET TRANSACTION
*
** CAUTION: Executing this example will modify the database records!
*****
DEFINE DATA LOCAL
01 PERSON VIEW OF EMPLOYEES
  02 PERSONNEL-ID      (A8)
  02 NAME              (A20)
  02 FIRST-NAME        (A20)
  02 MIDDLE-I          (A1)
  02 CITY              (A20)
01 AUTO VIEW OF VEHICLES
  02 PERSONNEL-ID      (A8)
  02 MAKE              (A20)
  02 MODEL             (A20)
*
01 ET-DATA
  02 #APPL-ID          (A8) INIT <' '>
  02 #USER-ID          (A8)
  02 #PROGRAM          (A8)
  02 #DATE             (A10)
  02 #TIME             (A8)
  02 #PERSONNEL-NUMBER (A8)
END-DEFINE
*
GET TRANSACTION DATA #APPL-ID #USER-ID #PROGRAM
                    #DATE      #TIME      #PERSONNEL-NUMBER
*
IF #APPL-ID NOT = 'NORMAL' /* if last execution ended abnormally
AND #APPL-ID NOT = ' '
  INPUT (AD=OIL)
  // 20T '*** LAST SUCCESSFUL TRANSACTION ***' (I)
  / 20T '*****'
  /// 25T 'APPLICATION:' #APPL-ID
  / 32T 'USER:' #USER-ID
  / 29T 'PROGRAM:' #PROGRAM
  / 24T 'COMPLETED ON:' #DATE 'AT' #TIME
```

```

      / 20T 'PERSONNEL NUMBER:' #PERSONNEL-NUMBER
END-IF
*
REPEAT
  /*
  INPUT (AD=MIL) // 20T 'ENTER PERSONNEL NUMBER:' #PERSONNEL-NUMBER
  /*
  IF #PERSONNEL-NUMBER = '99999999'
    ESCAPE BOTTOM
  END-IF
  /*
  FIND1. FIND PERSON WITH PERSONNEL-ID = #PERSONNEL-NUMBER
  IF NO RECORDS FOUND
    REINPUT 'SPECIFIED NUMBER DOES NOT EXIST; ENTER ANOTHER ONE.'
  END-NOREC
  FIND2. FIND AUTO WITH PERSONNEL-ID = #PERSONNEL-NUMBER
  IF NO RECORDS FOUND
    WRITE 'PERSON DOES NOT OWN ANY CARS'
    ESCAPE BOTTOM
  END-NOREC
  IF *COUNTER (FIND2.) = 1 /* first pass through the loop
    INPUT (AD=M)
      / 20T 'EMPLOYEES/AUTOMOBILE DETAILS' (I)
      / 20T '-----'
      /// 20T 'NUMBER:' PERSONNEL-ID (AD=O)
      / 22T 'NAME:' NAME ' ' FIRST-NAME ' ' MIDDLE-I
      / 22T 'CITY:' CITY
      / 22T 'MAKE:' MAKE
      / 21T 'MODEL:' MODEL
    UPDATE (FIND1.) /* update the EMPLOYEES file
  ELSE /* subsequent passes through the loop
    INPUT NO ERASE (AD=M IP=OFF) ////////// 28T MAKE / 28T MODEL
  END-IF
  /*
  UPDATE (FIND2.) /* update the VEHICLES file
  /*
  MOVE *APPLIC-ID TO #APPL-ID
  MOVE *INIT-USER TO #USER-ID
  MOVE *PROGRAM TO #PROGRAM
  MOVE *DAT4E TO #DATE
  MOVE *TIME TO #TIME
  /*
  END TRANSACTION #APPL-ID #USER-ID #PROGRAM
                #DATE #TIME #PERSONNEL-NUMBER
  /*
  END-FIND /* for VEHICLES (FIND2.)
  END-FIND /* for EMPLOYEES (FIND1.)
END-REPEAT /* for REPEAT
*
STOP /* Simulate abnormal transaction end
END TRANSACTION 'NORMAL '
END

```

Selecting Records Using ACCEPT/REJECT

This section discusses the statements ACCEPT and REJECT which are used to select records based on user-specified logical criteria.

The following topics are covered:

- Statements Usable with ACCEPT and REJECT
- Example of ACCEPT Statement
- Logical Condition Criteria in ACCEPT/REJECT Statements
- Example of ACCEPT Statement with AND Operator
- Example of REJECT Statement with OR Operator
- Further Examples of ACCEPT and REJECT Statements

Statements Usable with ACCEPT and REJECT

The statements ACCEPT and REJECT can be used in conjunction with the database access statements:

- READ
- FIND
- HISTOGRAM

Example of ACCEPT Statement

```

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

Output of Program ACCEPX01:

Page	1		04-11-11 11:11:11
NAME	CURRENT POSITION	ANNUAL SALARY	
ADKINSON	DBA	46700	
ADKINSON	MANAGER	47000	
ADKINSON	MANAGER	47000	
AFANASSIEV	DBA	42800	
ALEXANDER	DIRECTOR	48000	
ANDERSON	MANAGER	50000	
ATHERTON	ANALYST	43000	
ATHERTON	MANAGER	40000	

Logical Condition Criteria in ACCEPT/REJECT Statements

The statements ACCEPT and REJECT allow you to specify logical conditions in addition to those that were specified in WITH and WHERE clauses of the READ statement.

The logical condition criteria in the IF clause of an ACCEPT / REJECT statement are evaluated *after* the record has been selected and read.

Logical condition operators include the following (see *Logical Condition Criteria* for more detailed information):

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

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

Example of ACCEPT Statement with AND Operator

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

```

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

Output of Program ACCEPX02:

NAME	CURRENT POSITION	ANNUAL SALARY
AFANASSIEV	DBA	42800
ATHERTON	ANALYST	43000
ATHERTON	MANAGER	40000

Example of REJECT Statement with OR Operator

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

```

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

Output of Program ACCEPX03:

NAME	CURRENT POSITION	ANNUAL SALARY
AFANASSIEV	DBA	42800
ATHERTON	ANALYST	43000
ATHERTON	MANAGER	40000

Further Examples of ACCEPT and REJECT Statements

See the following example programs:

- ACCEPX04 - ACCEPT IF ... LESS THAN ...
- ACCEPX05 - ACCEPT IF ... AND ...
- ACCEPX06 - REJECT IF ... OR ...

AT START/END OF DATA Statements

This section discusses the use of the statements AT START OF DATA and AT END OF DATA.

The following topics are covered:

- AT START OF DATA Statement
- AT END OF DATA Statement
- Example of AT START OF DATA and AT END OF DATA Statements
- Further Examples of AT START OF DATA and AT END OF DATA

AT START OF DATA Statement

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

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

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

AT END OF DATA Statement

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

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

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

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

The following example program illustrates the use of the statements AT START OF DATA and AT END OF DATA.

The Natural system variable *TIME has been incorporated into the AT START OF DATA statement to display the time of day.

The Natural system function OLD has been incorporated into the AT END OF DATA statement to display the name of the last person selected.

```
** Example 'ATSTAX01': AT START OF DATA
*****
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
  2 INCOME (1:1)
  3 CURR-CODE
```

```

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

```

The program produces the following output:

```

                XYZ EMPLOYEE ANNUAL SALARY AND BONUS REPORT

      NAME                CURRENT          INCOME
                        POSITION
                        CURRENCY   ANNUAL   BONUS
                        CODE       SALARY
-----
RUN TIME: 12:43:19.1

DUYVERMAN    PROGRAMMER    USD      34000      0
PRATT        SALES PERSON    USD      38000      9000
MARKUSH      TRAINEE        USD      22000      0

LAST PERSON SELECTED: MARKUSH

AVERAGE SALARY:      31333

```

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

See the following example programs:

- *ATENDX01 - AT END OF DATA*
- *ATSTAX02 - AT START OF DATA*
- *WRITEX09 - WRITE (in combination with AT END OF DATA)*

Unicode Data

Natural enables users to access wide-character fields (format W) in an Adabas database.

The following topics are covered:

- Data Definition Module
- Access Configuration
- Restrictions

Data Definition Module

Adabas wide-character fields (W) are mapped to Natural format U (Unicode).

The length definition for a Natural field of format U corresponds to half the size of the Adabas field of format W. An Adabas wide-character field of length 200 is, for example, mapped to (U100) in Natural.

Access Configuration

Natural receives data from Adabas and sends data to Adabas using UTF-16 as common encoding.

This encoding is specified with the OPRB parameter and sent to Adabas with the open request. It is used for wide-character fields and applies to the entire Adabas user session.

Restrictions

Wide-character fields (W) of variable length are not supported.

Collating descriptors are not supported.

For further information on Adabas and Unicode support refer to the specific Adabas product documentation.