

Using Dynamic and Large Variables

This chapter covers the following topics:

- General Remarks
 - Assignments with Dynamic Variables
 - Initialization of Dynamic Variables
 - String Manipulation with Dynamic Alphanumeric Variables
 - Logical Condition Criterion (LCC) with Dynamic Variables
 - AT/IF-BREAK of Dynamic Control Fields
 - Parameter Transfer with Dynamic Variables
 - Work File Access with Large and Dynamic Variables
 - DDM Generation and Editing for Varying Length Columns
 - Accessing Large Database Objects
 - Performance Aspects with Dynamic Variables
 - Outputting Dynamic Variables
 - Dynamic X-Arrays
-

General Remarks

Generally, the following rules apply:

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

Exception:

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

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

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

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

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

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

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

Assignments with Dynamic Variables

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

Example:

```
#MYDYNTXT1 := OPERAND
MOVE OPERAND TO #MYDYNTXT1
/* #MYDYNTXT1 IS AUTOMATICALLY EXTENDED UNTIL THE SOURCE OPERAND CAN BE COPIED
```

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

- MOVE ALL moves the source operand repeatedly to the target operand until the used length (*LENGTH) of the target operand is reached. The system variable *LENGTH is not modified. If *LENGTH is zero, the statement will be ignored.

- MOVE ALL *operand1* TO *operand2* UNTIL *operand3* moves *operand1* repeatedly to *operand2* until the length specified in *operand3* is reached. If *operand3* is greater than `*LENGTH(operand2)`, *operand2* is extended and `*LENGTH(operand2)` is set to *operand3*. If *operand3* is less than `*LENGTH(operand2)`, the used length is reduced to *operand3*. If *operand3* equals `*LENGTH(operand2)`, the behavior is equivalent to MOVE ALL.

Example:

```
#MYDYNTXT1 := 'ABCDEFGHIJKLMNO'          /* *LENGTH(#MYDYNTXT1) = 15
MOVE ALL 'AB' TO #MYDYNTXT1             /* CONTENT OF #MYDYNTXT1 = 'ABABABABABABABA';
                                          /* *LENGTH IS STILL 15
MOVE ALL 'CD' TO #MYDYNTXT1 UNTIL 6     /* CONTENT OF #MYDYNTXT1 = 'CDCDCD';
                                          /* *LENGTH = 6
MOVE ALL 'EF' TO #MYDYNTXT1 UNTIL 10    /* CONTENT OF #MYDYNTXT1 = 'EFEFEFEFEF';
                                          /* *LENGTH = 10
```

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

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

Valid syntax:

```
#OP2 := *LENGTH(#MYDYNTXT1)
MOVE SUBSTR (#MYDYNTXT1, #OP2) TO OPERAND          /* MOVE LAST CHARACTER TO OPERAND
#OP2 := *LENGTH(#MYDYNTXT1) + 1
MOVE OPERAND TO SUBSTR(#MYDYNTXT1, #OP2, #LEN_OPERAND) /* CONCATENATE OPERAND TO #MYDYNTXT1
```

Invalid syntax:

```
#OP2 := *LENGTH(#MYDYNTXT1) + 1
MOVE SUBSTR (#MYDYNTXT1, #OP2, 10) TO OPERAND      /* LEADS TO RUNTIME ERROR; UNDEFINED SUB-STRING
#OP2 := *LENGTH(#MYDYNTXT1 + 10)
MOVE OPERAND TO SUBSTR(#MYDYNTXT1, #OP2, #EN_OPERAND) /* LEADS TO RUNTIME ERROR; UNDEFINED GAP
#OP2 := *LENGTH(#MYDYNTXT1) + 1
MOVE OPERAND TO SUBSTR(#MYDYNTXT1, #OP2)          /* LEADS TO RUNTIME ERROR; UNDEFINED LENGTH
```

Assignment Compatibility

Example:

```
#MYDYNTXT1 := #MYSTATICVAR1
#MYSTATICVAR1 := #MYDYNTXT2
```

If the source operand is a static variable, the used length of the dynamic destination operand (`*LENGTH(#MYDYNTXT1)`) is set to the format length of the static variable and the source value is copied in this length including trailing blanks (alphanumeric and Unicode fields) or binary zeros (for binary fields).

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

is filled with blanks (for alphanumeric and Unicode fields) or binary zeros (for binary fields).

Initialization of Dynamic Variables

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

Example:

```
DEFINE DATA LOCAL
1 #MYDYNTXT1 (A) DYNAMIC
END-DEFINE
#MYDYNTXT1 := 'SHORT TEXT'
WRITE *LENGTH(#MYDYNTXT1) /* USED LENGTH = 10
RESET #MYDYNTXT1 /* USED LENGTH = 10, VALUE = 10 BLANKS
```

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

Example:

```
MOVE ALL 'Y' TO #MYDYNTXT1 UNTIL 15 /* #MYDYNTXT1 CONTAINS 15 'Y'S, USED LENGTH = 15
```

String Manipulation with Dynamic Alphanumeric Variables

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

Example:

```
** Example 'DYNAMX01': Dynamic variables (with COMPRESS and SEPARATE)
*****
DEFINE DATA LOCAL
1 #MYDYNTXT1 (A) DYNAMIC
1 #TEXT (A20)
1 #DYN1 (A) DYNAMIC
1 #DYN2 (A) DYNAMIC
1 #DYN3 (A) DYNAMIC
END-DEFINE
*
MOVE ' HELLO WORLD ' TO #MYDYNTXT1
WRITE #MYDYNTXT1 (AL=25) 'with length' *LENGTH (#MYDYNTXT1)
/* dynamic variable with leading and trailing blanks
*
MOVE ' HELLO WORLD ' TO #TEXT
*
MOVE #TEXT TO #MYDYNTXT1
WRITE #MYDYNTXT1 (AL=25) 'with length' *LENGTH (#MYDYNTXT1)
/* dynamic variable with whole variable length of #TEXT
*
COMPRESS #TEXT INTO #MYDYNTXT1
WRITE #MYDYNTXT1 (AL=25) 'with length' *LENGTH (#MYDYNTXT1)
/* dynamic variable with leading blanks of #TEXT
*
*
```

```

#MYDYNTXT1 := 'HERE COMES THE SUN'
SEPARATE #MYDYNTXT1 INTO #DYN1 #DYN2 #DYN3 IGNORE
*
WRITE / #MYDYNTXT1 (AL=25) 'with length' *LENGTH (#MYDYNTXT1)
WRITE #DYN1 (AL=25) 'with length' *LENGTH (#DYN1)
WRITE #DYN2 (AL=25) 'with length' *LENGTH (#DYN2)
WRITE #DYN3 (AL=25) 'with length' *LENGTH (#DYN3)
/* #DYN1, #DYN2, #DYN3 are automatically extended or reduced
*
EXAMINE #MYDYNTXT1 FOR 'SUN' REPLACE 'MOON'
WRITE / #MYDYNTXT1 (AL=25) 'with length' *LENGTH (#MYDYNTXT1)
/* #MYDYNTXT1 is automatically extended or reduced
*
END

```

Note:

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

Logical Condition Criterion (LCC) with Dynamic Variables

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

Example:

```

IF #MYDYNTXT1 = #MYDYNTXT2 OR #MYDYNTXT1 = "***" THEN ...
IF #MYDYNTXT1 < #MYDYNTXT2 OR #MYDYNTXT1 < "***" THEN ...
IF #MYDYNTXT1 > #MYDYNTXT2 OR #MYDYNTXT1 > "***" THEN ...

```

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

Example:

```

#MYDYNTXT1 := 'HELLO' /* USED LENGTH IS 5
#MYDYNTXT2 := 'HELLO ' /* USED LENGTH IS 10
IF #MYDYNTXT1 = #MYDYNTXT2 THEN ... /* TRUE
IF #MYDYNTXT1 = #MYDYNTXT2 AND
   *LENGTH(#MYDYNTXT1) = *LENGTH(#MYDYNTXT2) THEN ... /* FALSE

```

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

Example:

```
#MYDYNTTEXT1 := 'HELLO1'           /* USED LENGTH IS 6
#MYDYNTTEXT2 := 'HELLO2'           /* USED LENGTH IS 10
IF #MYDYNTTEXT1 < #MYDYNTTEXT2 THEN ... /* TRUE
#MYDYNTTEXT2 := 'HALLO'
IF #MYDYNTTEXT1 > #MYDYNTTEXT2 THEN ... /* TRUE
```

Comparison Compatibility

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

Example:

```
#MYSTATTEXT1 := 'HELLO'             /* FORMAT LENGTH OF MYSTATTEXT1 IS A20
#MYDYNTTEXT1 := 'HELLO'             /* USED LENGTH IS 5
IF #MYSTATTEXT1 = #MYDYNTTEXT1 THEN ... /* TRUE
IF #MYSTATTEXT1 > #MYDYNTTEXT1 THEN ... /* FALSE
```

AT/IF-BREAK of Dynamic Control Fields

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

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

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

Parameter Transfer with Dynamic Variables

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

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

Call By Reference

Caller	Parameter	
	Static	Dynamic
Static	Yes	No
Dynamic	No	Yes

The formats of dynamic variables A or B must match.

Call by Value (Result)

Caller	Parameter	
	Static	Dynamic
Static	Yes	Yes
Dynamic	Yes	Yes

Note:

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

Example 1:

```

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

```

Subprogram DYNAMX03:

```

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

```

Example 2:

```

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

```

Subprogram DYNAMX05:

```

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

```

CALL 3GL Program

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

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

- No limitation on the number of passed parameters (former limit 40).
- No limitation on the parameter's data size (former limit 64 KB per parameter).
- Full parameter information can be passed to the 3GL program including array information. Exported functions are provided which allow secure access to the parameter data (formerly you had to take care not to overwrite memory inside of Natural)

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

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

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

Example:

```
MOVE ALL ' ' TO #MYDYNTXT1 UNTIL 10000
  /* a buffer of length 10000 is allocated
  /* #MYDYNTXT1 is initialized with blanks
  /* and *LENGTH(#MYDYNTXT1) = 10000
CALL INTERFACE4 'MYPROG' USING #MYDYNTXT1
WRITE *LENGTH(#MYDYNTXT1)
  /* *LENGTH(#MYDYNTXT1) may have changed in the 3GL program
```

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

Work File Access with Large and Dynamic Variables

The following topics are covered below:

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

PORTABLE and UNFORMATTED

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

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

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

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

ASCII, ASCII-COMPRESSED and SAG

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

Special Conditions for TRANSFER and ENTIRE CONNECTION

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

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

DDM Generation and Editing for Varying Length Columns

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

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

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

Example DDM:

```

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

```

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

Example:

```

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

```

Accessing Large Database Objects

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

Example Program:

```

DEFINE DATA LOCAL

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

END-DEFINE

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

        WRITE 'length of PICTURE1: ' L@PICTURE1  /* the L-INDICATOR will contain the length
                                                /* of PICTURE1 (= *LENGTH(PICTURE1))

        /* do something with PICTURE1 and TEXT1

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

END-SELECT

```

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

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

(equivalent for Format B variables)

Parameter with LINDICATOR Clause in SQL Statements

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

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

Performance Aspects with Dynamic Variables

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

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

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

Example:

```

** Example 'DYNAMX06': Dynamic variables (allocated memory)
*****
DEFINE DATA LOCAL
1 #MYDYNTXT1 (A) DYNAMIC
1 #LEN (I4)
END-DEFINE
*
#MYDYNTXT1 := 'a' /* used length is 1, value is 'a'
/* allocated size is still 1
WRITE *LENGTH(#MYDYNTXT1)

```

```

*
EXPAND DYNAMIC VARIABLE #MYDYNTXT1 TO 100
                        /* used length is still 1, value is 'a'
                        /* allocated size is 100
*
CALLNAT 'DYNAMX05' USING #MYDYNTXT1
WRITE *LENGTH(#MYDYNTXT1)
                        /* used length and allocated size
                        /* may have changed in the subprogram
*
#LEN := *LENGTH(#MYDYNTXT1)
REDUCE DYNAMIC VARIABLE #MYDYNTXT1 TO #LEN
                        /* if allocated size is greater than used length,
                        /* the unused memory is released
*
REDUCE DYNAMIC VARIABLE #MYDYNTXT1 TO 0
WRITE *LENGTH(#MYDYNTXT1)
                        /* free allocated memory for dynamic variable
END

```

Rules:

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

Outputting Dynamic Variables

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

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

Dynamic X-Arrays

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

Example:

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