MOVE

This chapter covers the following topics:

- Function
- Syntax Description
- Examples

Related Statements: ADD | COMPRESS | COMPUTE | DIVIDE | EXAMINE | MOVE ALL | 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 `MOVE` statement with multiple target operands is identical to the corresponding individual `MOVE` statements:

```
MOVE #SOURCE TO #TARGET1 #TARGET2
```

is identical to

```
MOVE #SOURCE TO #TARGET1
MOVE #SOURCE TO #TARGET2
```

Example:

```
DEFINE DATA LOCAL
1 #ARRAY(I4/1:3) INIT <3,0,9>
1 #INDEX(I4)
1 #RESULT(I4)
END-DEFINE
*
#INDEX := 1
MOVE #ARRAY(#INDEX) TO #INDEX       /* #INDEX is 3
                       #RESULT      /* #RESULT is 9
*
#INDEX := 2
MOVE #ARRAY(#INDEX) TO  #INDEX     /* #INDEX is 0
                        #ARRAY(3)  /* returns run time error NAT1316
```

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*.

If *operand2* is of format C, *operand1* may also be specified as `(`*`parameter`*`)`. Valid parameters are:

Parameters that can be specified with the MOVE statement		**Specification (S = at statement level, E = at element level)**
AD	Attribute Definition	SE
CD	Color Definition	S

For more information on data transfer compatibility and the rules for data transfer, see the section *Data Transfer* in the *Programming Guide*.

Other Considerations

If a database field is used as the result field, the `MOVE` operation results in an update only to the internal value of the field as used within the program. The value of the field in the database remains unchanged.

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*.

Note:
If *operand1* is a time variable (Format T), only the time component of the variable content is transferred, but not the date component (except with `MOVE EDITED`, described under *Syntax 4* and *Syntax 5*).

Syntax Description

Different structures are possible for this statement.

- Syntax 1 - MOVE ROUNDED
- Syntax 2 - MOVE SUBSTRING
- Syntax 3 - MOVE BY NAME / POSITION
- Syntax 4 - MOVE EDITED (Edit Mask Specified with operand2)
- Syntax 5 - MOVE EDITED (Edit Mask Specified with operand1)
- Syntax 6 - MOVE LEFT / RIGHT JUSTIFIED
- Syntax 7 - MOVE NORMALIZED
- Syntax 8 - MOVE ENCODED

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

Syntax 1 - MOVE ROUNDED

`MOVE` [**`ROUNDED`**] *operand1* [(*parameter*)] **`TO`** *operand2* **...**

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:

MOVE ROUNDED	This option causes *operand2* to be rounded. `ROUNDED` is ignored if *operand2* is not numeric. If *operand2* is of format N or P and *operand2* is specified more than once, `ROUNDED` is ignored for target operands with seven positions after the decimal point. See also *Example 1 - Various Samples of MOVE Statement Usage*.	
(*parameter*)	As *parameter*, you can specify the option `PM=I` or the session parameter `DF`:	

	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 `ZYX`:
		
		`MOVE 'XYZ' TO #A`
		`MOVE #A` **`(PM=I)`** `TO #B`
		
		`PM=I` can only be specified if *operand2* has alphanumeric format.
		
		Any trailing blanks in *operand1* will be removed (blanks and binary zeros are removed), then the value is reversed and moved to *operand2*. If *operand1* is not of alphanumeric format, the value will be converted to alphanumeric format before it is reversed.
		
		See also the use of `PM=I` in conjunction with `MOVE LEFT/RIGHT JUSTIFIED`.

	DF	If *operand1* is a date variable and *operand2* is an alphanumeric field, you can specify the session parameter `DF` as parameter for this date variable. The session parameter `DF` is described in the *Parameter Reference*.

Syntax 2 - MOVE SUBSTRING

```
MOVE { operand1                                 } [(parameter)] TO { operand2                                } ...
     { SUBSTRING (operand1,operand3,operand4)   }                  { SUBSTRING (operand2,operand5,operand6)  }
```

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:

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.
	
	If the underlying field format of *operand1* is
	● alphanumeric (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 (A) 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 *operand3/5* or *operand4/6* is a binary variable, it may be used only with a length of less than or equal to 4.
	
	If you omit *operand3/5*, the starting position is assumed to be `1`. If you omit *operand4/6*, 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.
	
	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).

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:

MOVE BY NAME *operand1* TO *operand2*	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*.
MOVE BY POSITION *operand1* TO *operand2*	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*)

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:

MOVE EDITED	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 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.
	
	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	For details on edit masks, see the session parameter `EM` in the *Parameter Reference*.

Syntax 5 - MOVE EDITED (Edit Mask Specified with *operand1*)

MOVE EDITED *operand1* (**EM**=*value*) **TO** *operand2*

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:

MOVE EDITED	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 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.
	
	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	For details on edit masks, see the session parameter `EM` 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:

MOVE LEFT / RIGHT JUSTIFIED	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	With `MOVE LEFT JUSTIFIED`, any leading blanks in *operand1* are removed (blanks and binary zeros 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	With `MOVE RIGHT JUSTIFIED`, any trailing blanks in *operand1* are truncated (blanks and binary zeros are removed) 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	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 format, the value is converted to alphanumeric format. 2. Any trailing blanks in *operand1* are removed (blanks and binary zeros are removed). 3. In the case of `LEFT JUSTIFIED`, any leading blanks in *operand1* are also removed (blanks and binary zeros are 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:

MOVE NORMALIZED	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	Unicode string to be converted.
operand2	Target operand.

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:

MOVE ENCODED	The `MOVE ENCODED` statement 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 *http://icu.sourceforge.net/userguide/*.
operand1	String to be converted.
operand2	Code page of *operand1*. Can only be supplied if *operand1* is of format A or B. See Note 1 and 3.
operand3	Target. 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.
operand4	Code page of *operand3*. Can only be supplied if operand3 is of format A or B. See Note 1 and 3.
operand5	Without the keyword `GIVING`, a Natural error message is returned in case of an error. If the keyword `GIVING` is used, *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.
3. Only code page names defined with the macro `NTCPAGE` in the source module NATCONFG can be used. Other code page names are rejected with a corresponding runtime error.

Examples:

```
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.

Examples

- Example 1 - Various Samples of MOVE Statement Usage
- Example 2 - MOVE BY NAME
- Example 3 - MOVE BY NAME with Arrays
- Example 4- MOVE BY POSITION

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.