Requirements of ABAP Programs in Unicode Systems

Version: July 16, 2003

Contents Page

1. Codes

In the past, SAP developers used various codes to encode characters of different alphabets, for example, ASCII, EBCDI, or double-byte code pages.

- ASCII (American Standard Code for Information Interchange) encodes each character using 1 byte = 8 bit. This makes it possible to represent a maximum of 2^8 = 256 characters to which the combinations [00000000, 11111111] are assigned. Common code pages are, for example, ISO88591 for West European or ISO88595 for Cyrillic fonts.

- EBCDIC (Extended Binary Coded Decimal Interchange) also uses 1 byte to encode each character, which again makes it possible to represent 256 characters. EBCDIC 0697/0500 is an old IBM format that is used on AS/400 machines for West European fonts, for example.

- Double-byte code pages require 1 or 2 bytes for each character. This allows you to form 2^{16} = 65536 combinations where usually only 10,000 - 15,000 characters are used. Double-byte code pages are, for example, SJIS for Japanese and BIG5 for traditional Chinese.

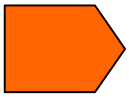
Using these character sets, you can account for each language relevant to the SAP System. However, problems occur if you want to merge texts from different incompatible character sets in a central system. Equally, exchanging data between systems with incompatible character sets can result in unprecedented situations.

One solution to this problem is to use a code comprising all characters used on earth. This code is called **Unicode** (ISO/IEC 10646) and consists of at least 16 bit = 2 bytes, alternatively of 32 bit = 4 bytes per character. Although the conversion effort for the SAP kernel and applications is considerable, the migration to Unicode provides great benefits in the long run:

- The Internet (**www**) and consequently also **mySAP.com** are entirely based on Unicode, which thus is a basic requirement for international competitiveness.
- Unicode allows all SAP users to install a central system that covers all business processes worldwide.
- Companies using different distributed systems frequently want to aggregate their worldwide corporate data. Without Unicode, they would be able to do this only to a limited degree.
- With Unicode, you can use multiple languages simultaneously at a single frontend computer.
- Unicode is required for cross-application data exchange without loss of data due to incompatible character sets. One way to present documents in the World Wide Web (www) is XML, for example.

ABAP programs must be modified wherever an explicit or implicit assumption is made with regard to the internal length of a character. As a result, a new level of abstraction is reached which makes it possible to run one and the same program both in conventional and in Unicode systems. In addition, if new characters are

added to the Unicode character set, SAP can decide whether to represent these characters internally using 2 or 4 bytes.



The examples presented in the following sections are based on a Unicode encoding using 2 bytes per character.

2. ABAP Development Under Unicode

A Unicode-enabled ABAP program (**UP**) is a program in which all Unicode checks are effective. Such a program returns the same results in a non-Unicode system (**NUS**) as in a Unicode system (**US**). In order to perform the relevant syntax checks, you must activate the Unicode flag in the screens of the program and class attributes.

In a US, you can only execute programs for which the Unicode flag is set. In future, the Unicode flag must be set for all SAP programs to enable them to run on a US. If the Unicode flag is set for a program, the syntax is checked and the program executed according to the rules described in this document. This is regardless of whether it is a Unicode or non-Unicode program. From now on, the Unicode flag must be set for all new programs and classes that are created.

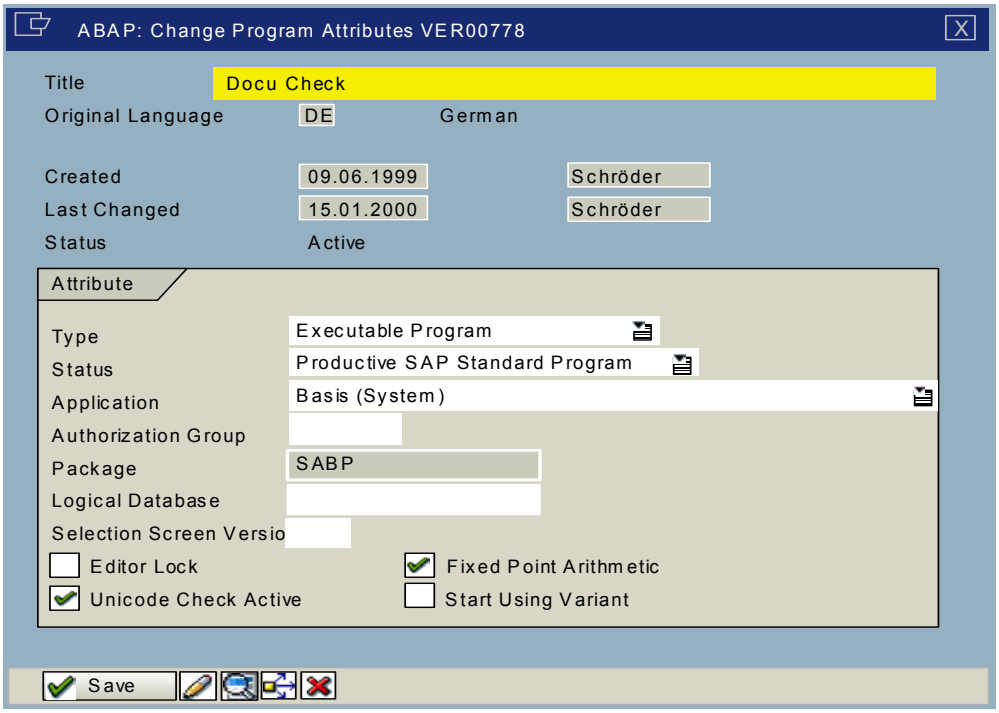
If the Unicode flag is not set, a program can only be executed in an NUS. The syntactical and semantic changes described below do not apply to such programs. However, you can use all language extensions that have been introduced in the process of the conversion to Unicode.
As a result of the modifications and restrictions associated with the Unicode flag, programs are executed in both Unicode and non-Unicode systems with the same semantics to a large degree. In rare cases, however, differences may occur. Programs that are designed to run on both systems therefore need to be tested on **both** platforms.
Additionally, as part of the introduction of Unicode, the following modifications have been made in the syntax check to the Unicode flag:

1. In Unicode programs, unreachable statements now cause a syntax error. In non-Unicode programs, this previously only caused a syntax warning.
2. In Unicode programs, calling a function module, whose parameter names are specified statically as a literal or constant, will raise an exception that can be handled if an incorrect parameter name is specified. This only applies to function modules that are not called via Remote Function Call. In non-Unicode programs, an incorrect name was previously ignored.

You are recommended to follow the procedure below to make your programs US-compliant:

- The UNICODE task in transaction SAMT performs first an NUS and then a US syntax check for a selected program set. For an overview of the syntax errors by systems, programs and authors, consult the following document in SAPNet: Alternatively, you can start the ABAP program RSUNISCAN_FINAL to determine the Unicode-relevant syntax errors for a single program.

- Before you can set the Unicode flag in the NUS in the attributes of the program concerned, all syntax errors must be removed.
- Having enabled the Unicode flag in the NUS, you can run the syntax check for this program. To display a maximum of 50 syntax errors simultaneously, choose Utilities -> Settings -> Editor in the ABAP Editor and select the corresponding checkbox.
- Once all syntactical requirements are met in the NUS, you must test the program both in the NUS and US. The purpose of this test is to recognize any runtime errors and make sure that the results are correct in both systems. To rule out runtime errors in advance, you should always type field symbols and parameters so that any potential problems can be detected during the syntax check.



3. Concepts and Conventions

3.1 Data Types

The data types that can be interpreted as **character-type** in a UP include:

- C — Character (letters, numbers, special characters)
- N — Numeric character (numbers)
- D — Date
- T — Time
- STRING — Character string
- Character type structures — Structures which either directly or in substructures contain only fields of types C, N, D or T.

In an NUS, a character of this type has a length of 1 byte, and in a US a length corresponding to the length of one character on the relevant platform. The data type **W** is no longer supported.

Variables of the types X and XSTRING are called **byte-type**. The main characteristics of the different kinds of **structures** are:

- Flat structures contain only fields of the elementary types C, N, D, T, F, I, P, and X, or structures containing these types.
- Deep structures contain strings, internal tables and field or object references in addition to the elementary types.
- Nested structures are structures that contain substructures as components.
- Non-nested structures are structures that do not contain any substructures.

3.2 Data Layout of Structures

For several data types, such as I and F or object references, certain alignment requirements are in place that depend on the platform used. Fields of these types must begin in memory at an address divisible by 4 or 8. Character-type types must begin at a memory address divisible by 2 or 4 depending on their Unicode representation.

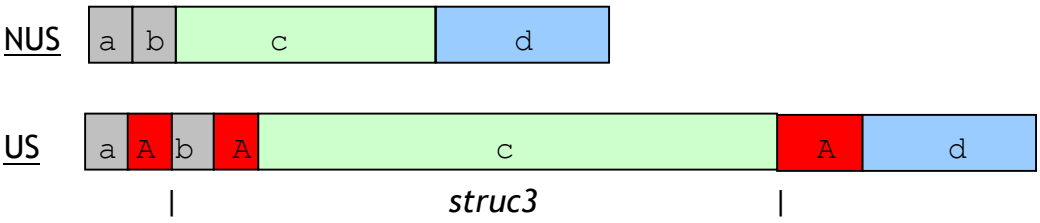
Within structures, bytes can be inserted before or after components with alignment requirements to achieve the necessary alignment. These bytes are referred to as **alignment** (A). A substructure is aligned according the field with the biggest alignment requirement. In this case a contained substructure counts as a field. Includes in structures are treated as substructures.

In the sample structure below that contains three fields, no alignments are created in an NUS or US.

```
BEGIN OF struc1,
  a(1) TYPE X,
  b(1) TYPE X,
  c(6) TYPE C,
END OF struc1.
```

In the next example, however, alignments are created in a US but not in an NUS. The first alignment gap is created because of the alignment of structure *struc3*, the second because of the alignment of C field *c*, and the third because of the addressing of integer *d*.

```
BEGIN OF struc2,
  a(1) TYPE X,
  BEGIN OF struc3,
    b(1) TYPE X,
    c(6) TYPE C,
  END OF struc3,
  d    TYPE I,
END OF struc2.
```



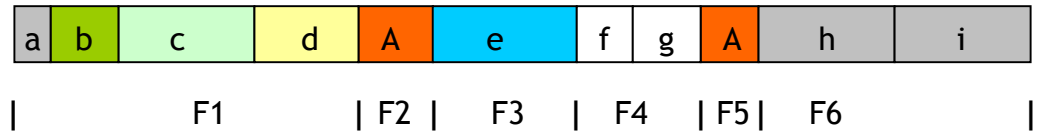
3.3 Unicode-Fragment View

The data layout of structures is relevant to UP checks with regard to the reliability of assignments and comparisons, for example. This data layout is represented in the Unicode fragment view. The fragment view breaks down the structure into alignment gaps, in byte and character-type areas, and all other types such as P, I, F, strings, references or internal tables.

Juxtaposed character-type components of a structure except strings are internally combined into a group if no alignment gaps exist between these components. All possible alignment requirements for characters are considered. Juxtaposed byte type components are grouped together in the same way.

```
BEGIN OF struc,
  a(2) TYPE C,
  b(4) TYPE N,
  c    TYPE D,
  d    TYPE T,
  e    TYPE F,
  f(2) TYPE X,
  g(4) TYPE X,
  h(8) TYPE C,
  i(8) TYPE C,
END OF struc.
```

In the following example, F1 - F6 show the individual fragments of structure *struc:*



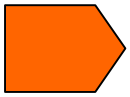
3.4 Permitted Characters

In a US, all ABAP program sources are also stored as Unicode. As in ABAP Objects, you may only use the following characters as identifiers in programs for which the Unicode flag is set:

1. Letters a - z and A - Z without the German 'umlauts'
2. Numbers 0 - 9
3. The underscore _

For compatibility reasons, the characters %, $, ?, -, #, *, and / are also still permitted, but they should only be used for good reason in exceptional cases. Note

that the slash can only be used to separate namespaces in the form /name/. There must be at least three characters between two slashes.



To ensure that programs can be transported from a US to a NUS without any loss of information in the process of conversion, you should not use any characters for comments and literals even in a US that cannot be represented in an NUS.

4. Restrictions in Unicode Programs

The adjustments you have to make and the restrictions that apply in the Unicode context have been limited to the essentials on the ABAP development side to keep the conversion effort for ABAP users to a minimum. In some cases, however, this has led to the emergence of more complex rules, for example, with regard to assignments and comparisons between incompatible structures.

4.1 Character and Numeric Type Operands

Up to now, you have been able to use flat structures as arguments of ABAP statements wherever single fields of type C were expected. In a UP this is no longer generally permitted. In a UP, you can use a structured field in a statement expecting a single field only if this structured field consists of character-type elementary types or purely character-type substructures. The structure is treated like a single field of type C.

The main restrictions applying to a UP in contrast to an NUS result from the fact that flat structures are only considered character-type on a limited basis, and fields of type X or STRING are never considered character-type. In addition, flat structures are only considered numeric-type if they are purely character-type. Numeric-type arguments include, for example, offset or index specifications as in READ TABLE ... INDEX i. The following examples show a structure that is character-type and a structure that is not:

```
BEGIN OF struc1,
  a(2) TYPE C,
  b(2) TYPE C,
  x(1) TYPE X,
  i    TYPE I,
END OF struc.
```

Not character-type

```
BEGIN OF struc2,
  a(2) TYPE C,
  n(6) TYPE N,
  d    TYPE D,
  t    TYPE T,
END OF struc.
```

Character-type

Another example is a control break in an internal table, triggered by the AT keyword. In a NUS, fields of type X to the right of the control key are treated as character-type, and are thus filled with an asterisk. In Unicode systems, conversely, the same type is filled with its initial value.

4.2 Access Using Offset and Length Specifications

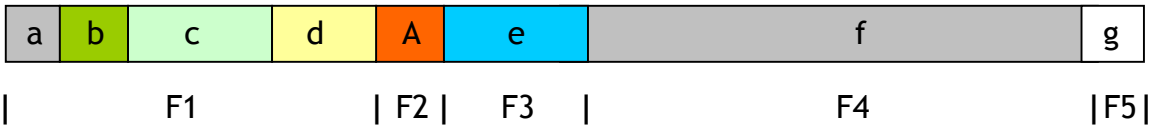
Offset and length specifications are generally critical since the length of each character is platform-dependent. As a result, it is initially unclear as to whether the byte unit or the character unit is referred to in mixed structures. This forced us to put in place certain considerable restrictions. However, access using offset or length specifications is still possible to the degree described in the following. The tasks subject to this rule include accessing single fields and structures, passing parameters to subroutines and working with field symbols.

- **Single field access**
  Offset-based or length-based access is supported for character-type single fields, strings and single fields of types X and XSTRING. For character-type fields and fields of type STRING, offset and length are interpreted on a character-by-character basis. Only for types X and XSTRING, the values for offset and length are interpreted in bytes.

- **Structure access**
  Offset-based or length-based access to structured fields is a programming technique that should be avoided. This access type results in errors if both character and non-character-type components exist in the area identified by offset and length.

  Offset-based or length-based access to structures is only permitted in a UP if the structures are flat and the offset/length specification includes only character-type fields from the beginning of the structure. The example below shows a structure with character-type and non-character-type fields. Its definition in the ABAP program and the resulting assignment in the main memory is as follows:

```
BEGIN OF STRUC,
  a(3)  TYPE C,        "Length 3 characters
  b(4)  TYPE N,        "Length 4 characters
  c     TYPE D,        "Length 8 characters
  d     TYPE T,        "Length 6 characters
  e     TYPE F,        "Length  8 bytes
  f(26) TYPE C,        "Length 28 characters
  g(4)  TYPE X,        "Length  2 bytes
END OF STRUC.
```



  Internally, the fragment view contains four fragments [F1-F4]. Offset-based or length-based access in this case is only possible in the initial part F1. Statements like *struc(21)* or *struc+7(14)* are accepted by the ABAP interpreter and treated like a single field of type C. By contrast, *struc+57(2)* access is now only allowed in an NUP. If offset-based or length-based access to a structure is permitted, both the offset and length specifications are generally interpreted as characters in a UP.

- **Passing parameters to subroutines**

  Up to now, parameter passing with PERFORM has allowed you to use cross-field offset and length specifications. In future, this will no longer be allowed in a UP. In a UP, offset-based and length-based access beyond field boundaries returns a syntax or runtime error. For example, access types c+15 or c+5(10) would trigger such an error for a ten-digit C field c.

  If only an offset but no length is specified for a parameter, the entire length of the field instead of the remaining length was previously used for access. As a result, parameter specifications are cross-field if you use only an offset, and therefore trigger a syntax error in a UP. `PERFORM test USING c+5` is consequently not permitted.

  In addition, in a UP, you can continue to specify the remaining length starting from the offset *off* for parameters using the form *field+off(*)*.

- **Ranges for offset-based and length-based access when using field symbols**

  A UP ensures that offset-based or length-based access with **ASSIGN** is only permitted within a predefined range. Normally, this range corresponds to the field boundaries in case of elementary fields or, in case of flat structures, to the purely character-type initial part. Using a special RANGE addition for ASSIGN, you can expand the range beyond these boundaries.

  Field symbols are assigned a range allowed for offset/length specifications. If the source of an ASSIGN statement is specified using a field symbol, the target field symbol adopts the range of the source. If not explicitly specified otherwise, the RANGE is determined as follows:

  ```
  ASSIGN field TO <f>.
  ```
  In a UP, the field boundaries of *field* are assigned to <*f*> as the range, where *field* is no field symbol.

  ```
  ASSIGN <g> TO <f>.
  ```
  <*f*> adopts the range of <*g*>.

  ```
  ASSIGN elfield+off(len) TO <f>.
  ```
  In a UP, the field boundaries of the elementary field *elfield* are assigned to <*f*> as the range.

  ```
  ASSIGN <elfield>+off(len) TO <f>.
  ```
  <*f*> adopts the range of the elementary field <*elfield*>.

  ```
  ASSIGN struc+off(len)   TO <f>.
  ASSIGN <struc>+off(len) TO <f>.
  ```
  In a UP, the purely character-type initial part of the flat structures *struc* or <*struc*> determines the range boundaries.

  If the assignment to the field symbol is not possible because the offset or length specification exceeds the range permitted, the field symbol is set to UNASSIGNED in a UP. Other checks such as type or alignment checks return a runtime error in a UP. As a rule, offset and length specifications are counted in

characters for data types C, N, D, and T as well as for flat structures, and in bytes in all other cases.

- **Offset without length specification when using field symbols**

  Up to now, `ASSIGN field+off TO <f>` has shown the special behavior that the field length instead of the remaining length of *field* was used if only an offset but not length was specified. Since an ASSIGN with a cross-field offset is therefore problematic under Unicode, you must observe the following rules:

  1. Using `ASSIGN field+off(*)...` you can explicitly specify the remaining length.
  2. `ASSIGN <f>+off TO <g>` is only permitted if the runtime type of <f> is flat and elementary, that is, C, N, D, T (offset in characters) or X (offset in bytes).
  3. `ASSIGN field+off TO <g>` is generally forbidden from a syntax point of view since any offset other than 0 would cause the range to be exceeded. Exceptions are cases in which a RANGE addition is used.

  These rules enable you also in future to pass a field symbol through an elementary field using `ASSIGN <f>+n TO <f>`, as it is the case in a loop, for example.

4.3 Assignments

This section deals with implicit and explicit type conversions using the equal sign (=) or the MOVE statement. Two fields can be converted if the content of one field can be assigned to the other field without triggering a runtime error.

For conversions between structured fields or a structured field and a single field, flat structures were previously treated like C fields. With the implementation of Unicode, this approach has become too error-prone since it is not clear if programs can be executed with platform-independent semantics.

Two fields are compatible if they have the same type and length. If deep structures are assigned, the fragment views must therefore be identical. One requirement in connection with the assignment and comparison of deep structures has been that type compatibility must exist between the operands, which requires both operands to have the same structure. This requirement will continue to apply to Unicode systems.

- **Conversion between flat structures**

  To check whether conversion is permitted at all, the Unicode fragment view of the structures is set up initially by combining character and byte type groups and alignment gaps as well as any other components. If the type and length of the fragments of the source structure are identical in the length of the shorter structure, conversion is permitted. Assignment is allowed subject to the fulfillment of the following conditions:

1. The fragments of both structures up to the second-last fragment of the shorter structure are identical.
2. The last fragment of the shorter structure is a character or byte type group.
3. The corresponding fragment of the longer structure is a character or byte type group with a greater length.

If the target structure is longer than the source structure, the character-type components of the remaining length are filled with blank characters. All other components of the remaining length are filled with the type-adequate initial value, and alignment gaps are filled with zero bytes. Since longer structures were previously filled with blanks by default, using initial values for non-character-type component types is incompatible. This incompatible change is, however, rather an error correction. For reasons of compatibility, character-type components are not filled with initial values.

```
BEGIN OF struc1,       BEGIN OF struc2,
   a(1) TYPE C,          a(1) TYPE C,
   x(1) TYPE X,          b(1) TYPE C,
END OF struc1.         END OF struc2.
```

The assignment *struc1* = *struc2* is not allowed under Unicode since *struc1-x* in contrast to *struc2-b* occupies only one byte.

```
BEGIN OF struc3,       BEGIN OF struc4,
   a(2) TYPE C,          a(8) TYPE C,
   n(6) TYPE N,          i    TYPE I,
  i     TYPE I,          f    TYPE F,
END OF struc3.         END OF struc4.
```

The assignment *struc3* = *struc4* is allowed since the fragment views of the character-type fields and the integer are identical.

```
BEGIN OF struc5,       BEGIN OF struc6,
   a(1) TYPE X,          a(1) TYPE X,
   b(1) TYPE X,          BEGIN OF struc0,
   c(1) TYPE C,            b(1) TYPE X,
END OF struc5.             c(1) TYPE C,
                         END OF struc0,
                       END OF struc6.
```

struc5 = *struc6* is again not permitted since the fragment views of both structures are not identical due to the alignment gaps before *struc0* and *struc0-c*.

```
BEGIN OF struc7,       BEGIN OF struc8,
   p(8) TYPE P,          p(8) TYPE P,
   c(1) TYPE C,          c(5) TYPE C,
END OF struc7.           o(8) TYPE P,
                       END OF struc8.
```

The assignment *struc7* = *struc8* works since the Unicode fragment views are identical with regard to the length of structure *struc7*.
For deep structures, the operand types must be compatible as usual. As an enhancement measure, we slightly generalized the convertibility in case of object references and table components.

- **Conversion between structures and single fields**

  The following rules apply for converting a structure into a single field and vice versa:

  1. If a structure is purely character-type, it is treated like a C field during conversion.
  2. If the single field is of type C, but only part of the structure is character-type, conversion is only possible if the structure begins with a character-type structure and if this structure is at least as long as the single field. Conversion now takes place between the first character-type group of the structure and the single field. If the structure is the target field, the character type sections of the remainder are filled with blanks, and all other components are filled with the type-adequate initial value.
  3. Conversion is not permitted if the structure is not purely character-type and if the single field is not of type C.

  As with the assignment between structures, filling non-character-type components with the initial value is incompatible.

- **Conversion between internal tables**

  Tables can be converted if their row types are convertible. The restrictions described above therefore also effect the conversion of tables.

- **Implicit conversions**

  The above rules also apply to all ABAP statements that use implicit conversions according to the MOVE semantics. For example, this is true for the following statements for internal tables:

```
APPEND wa TO itab.
APPEND LINES OF itab1 TO itab2.

INSERT wa INTO itab.
INSERT LINES OF itab1 INTO [TABLE] itab2.

MODIFY itab FROM wa.
MODIFY itab ... TRANSPORTING ... WHERE ... ki = vi ...

READ TABLE itab ...INTO wa.
READ TABLE itab ...WITH KEY ...ki = vi ...

LOOP AT itab INTO wa.
LOOP AT itab .... WITH KEY ... ki = vi ...
```

  The restrictions for explicit conversion also apply to the implicit conversion of VALUE specifications.

4.4 Comparisons

In general, the rule applies that operands that can be assigned to one another with the MOVE statement can also be compared. An exception is object references, which can be compared but not always assigned.

- **Comparison of flat structures**

  Structures can also be compared if they are not compatible. As in the MOVE statement, the fragment views must be the same for the length of the shorter structure. If the structures have different lengths, the shorter structure is filled until it has the length of the other structure. As in the assignment, all character-type components are filled with spaces and all other components with initial values of the right type. The structures are compared fragment by fragment as defined by the fragment view.

- **Comparison of single fields and structures**

  The following rules are valid when single fields are compared with structures:
  1. If a structure is purely character-type, it is treated like a C field in the comparison.
  2. If the single field is of character-type, but the structure is only partly of character-type, the comparison is only possible if the first fragment of character-type in the structure is longer than the single field. The single field is extended to the structure length at runtime and filled with initial values for the comparison. The comparison is the same as for structured fields, where the fields are filled as in the MOVE statement.

```
c0(10)    TYPE C.

BEGIN OF struc,
  c1(15) TYPE C,
  i      TYPE I,
  c2(5)  TYPE C,
  n(7)   TYPE N,
END OF struc.
```

  In this example, *c0* is extended to the length of *struc* in storage. All areas > 10 are filled with initial values of the correct type for components that are not character-type and filled with space for other components.

- **Comparison of deep structures**

  As previously, mainly type compatibility of the operands is needed for comparing deep structures. The compatibility test for comparability was generalized so that structure components with references to classes or interfaces can be compared with one another, whatever the class hierarchy and implementation relation, as for single fields. Only comparability of table types is required for table components.

- **Comparison of internal tables**

  Tables can be compared if their row types can be compared. The restrictions described above therefore also affect table comparisons.

4.5 Processing Strings

String processing statements, whose arguments were all interpreted as fields of type C until now, are now divided into statements with character arguments and those with byte arguments.

- **String processing statements**

```
CLEAR ... WITH
CONCATENATE
CONDENSE
CONVERT TEXT ... INTO SORTABLE CODE
OVERLAY
REPLACE
SEARCH
SHIFT
SPLIT
TRANSLATE ... TO UPPER/LOWER CASE
TRANSLATE ... USING
```

  The arguments of these instructions must be single fields of type C, N, D, T or STRING or purely character-type structures. There is a syntax or runtime error if arguments of a different type are passed. A subset of this function is provided with the addition `IN BYTE MODE` for processing byte strings – that is, operands of type X or XSTRING. A statement such as `CONCATENATE a x b INTO c` is thus no longer possible when a, b, and c are all character-type, but x is of type X.

```
TRANSLATE ... CODEPAGE ...
TRANSLATE ... NUMBER FORMAT ...
```

  The above statements are not allowed in Unicode programs. Instead, you can use the new conversion classes, which are described in more detail on page 37.

- **Comparison operators for string processing**

```
CO
CN
CA
NA
CS
NS
CP
NP
```

  As with the string processing statements, these operators need single fields of type C, N, D, T or STRING as arguments and again purely character-type

structures are allowed. Special compare operators defined with the prefix BYTE- are provided for byte strings.

- **Functions for string processing**

  Function **STRLEN** only works with character-type fields and returns the length in characters. The new function XSTRLEN finds the length of byte strings.

  Until now, function **CHARLEN** returned the value1 for a text field beginning with a single byte character under an NUS. The value 2 is returned for text fields beginning with a double byte character. Under a US, **CHARLEN** returns the value 1 if *text* begins with a single Unicode character. If *text* begins with a Unicode double character from the surrogate area, the value 2 is returned.

  Function **NUMOFCHAR** returns the number of characters in a string or a character-type field. In single byte code pages, the function behaves like **STRLEN**. In multi-byte code pages, characters filling more than 1 byte are nevertheless considered to have length 1.

- **Output in fields and lists**

  In WRITE ... TO, any flat data types that were handled like C fields were allowed as target. For the WRITE statement, the following rules apply in Unicode programs: TO ... requires the target field to be of character-type. For the table variant WRITE ... TO itab INDEX idx the line type of the table must be of character-type. The offset and length are counted in characters.

  Until now, any flat structures could be output with WRITE. If the source field is a flat structure in a WRITE, it must have character-type only, in a UP. This affects the following statements:

```
WRITE f.
WRITE f TO g[+off][(len)].
WRITE (name) TO g.
WRITE f TO itab[+off][(len)] INDEX idx.
WRITE (name) TO itab[+off][(len)] INDEX idx.
```

4.6 Type Checks and Type Compatibility

For historical reasons, the types of field symbols and parameters in subroutines or function modules can be defined with the STRUCTURE addition.

- If the types of **field symbols** are defined with `FIELD-SYMBOLS <f> STRUCTURE s DEFAULT wa` and they are later assigned a data object *wa* with `ASSIGN wa TO <f> ...`, in a NUP both statements are checked to see if *wa* is at least as long as *s* and *wa* satisfies the alignment requirements of *s* at runtime.

If **parameter types** in function modules or subroutines are defined with `FORM form1 USING/CHANGING arg STRUCTURE s ...` or `FORM form2 TABLES itab_a STRUCTURE s ...` and the parameters are passed actual parameters with `PERFORM form1 USING/CHANGING wa` or `PERFORM form2 USING/CHANGING itab_b`, the NUP also only checks if *wa* or the line type of *itab_b* is at least as long as *s* and *wa* or the line type of *itab_b* satisfies the alignment requirements of *s*. The same is true for function module parameters whose types are defined with STRUCTURE.

The following extra rules are checked in a UP after defining the type with STRUCTURE when assigning data objects, that is for the DEFAULT addition in the FIELD-SYMBOLS statement, for ASSIGN, and when passing actual parameters.

1. If *wa* or the line type of *itab_b* is a flat or deep structure, the length of *s* must be the same for the Unicode fragment views of *wa* or of *itab_b* and *s*.
2. If *wa* is a single field, only the character-types C, N, D or T are allowed and the structure *s* must be purely character-type.

Checking both these rules requires additional runtime. It is therefore recommended that, if possible, you type the parameters using TYPE, since the test for actual compatibility is much faster.

- If the type of an argument in a function module was defined with ... **LIKE** *struc*, where *struc* is a flat structure, the NUP only checks if the argument is a flat structure with the same length when the parameters are passed. In the UP, it also checks that the fragment views of the current and formal parameters are the same. For performance reasons, it is again recommended that you use TYPE to assign types.
- Furthermore, two structures of which one or both contain **Includes**, are only compatible if the alignment gaps caused by the Include are the same on all platforms. In the following example, struc1 and struc2 are not compatible because a further alignment gap occurs in the US before the INCLUDE:

```
BEGIN OF struc1,      BEGIN OF struc2,      BEGIN OF struc3,
  a(1) TYPE X,          a(1) TYPE X.          b(1) TYPE X,
  b(1) TYPE X,          INCUDE struc3.        c(1) TYPE C,
  c(1) TYPE C,        END OF struc2.        END OF struc3.
END OF struc1.
```

Since the type compatibility can differ in a UP and an NUP, the type compatibility rules of the calling program are valid in an NUS for checking the parameters. This means that if an NUP calls a UP, the type compatibility is defined as in the NUP. Conversely, the Unicode check is activated if a UP calls an NUP.

4.7 Changes to Database Operations

Until now, in an NUP the data is copied to field *wa* or to table line *itab* as defined by the structure of the table work area *dbtab* without taking its structure into consideration. Only the length and alignment are checked.

```
SELECT * FROM dbtab ... INTO wa ...
SELECT * FROM dbtab ... INTO TABLE itab  ...
SELECT * FROM dbtab ... APPENDING TABLE itab ...

FETCH NEXT CURSOR c ... INTO wa.
FETCH NEXT CURSOR c ... INTO TABLE itab.
FETCH NEXT CURSOR c ... APPENDING TABLE itab.

INSERT INTO dbtab ... FROM wa.
INSERT dbtab ... FROM wa.
INSERT dbtab ... FROM TABLE itab.

UPDATE dbtab ... FROM wa.
UPDATE dbtab ... FROM TABLE itab.

MODIFY dbtab ... FROM wa.
MODIFY dbtab ... FROM TABLE itab.

DELETE dbtab FROM wa.
DELETE dbtab FROM TABLE itab.
```

The following rules are now valid in a UP:

If the work area or the line of the internal table is a structure, there is also a check if the fragment views of the work area and the database table are the same up to the length of the database table. If the work area is a single field, the field must be character-type and the database table must be purely character-type. These requirements are valid for all the commands mentioned above.

Only the types C, N, D, T, - and flat structures of these types - are now valid for the version field in any statement that processes database tables (READ, MODIFY, DELETE, LOOP) and uses the VERSION addition. Otherwise, a warning is triggered in an NUS, and a syntax error in a US.

4.8 Determining the Length and Distance

You may no longer use the DESCRIBE DISTANCE statement to define the lengths and distances of fields. It must be replaced with one of the new statements `DESCRIBE DISTANCE ...IN BYTE MODE` or `DESCRIBE DISTANCE ... IN CHARACTER MODE`.

The `DESCRIBE FIELD ...LENGTH` statement is also obsolete and must be replaced with one of the new statements `DESCRIBE FIELD ... LENGTH ... IN BYTE MODE` or `DESCRIBE FIELD ... LENGTH ... IN CHARACTER MODE`. Until now, the `DESCRIBE FIELD ... TYPE field` statement returned type C for flat structures. In a UP, type **u** is now returned for flat structures. This can be queried in the ABAP source code.

There are no changes for the `DESCRIBE FIELD ... TYPE ... COMPONENTS ...` statement under US. Similarly, the `DESCRIBE ... OUTPUT LENGTH ...` statement still returns the output length in characters.

4.9 Other Changes

The following text describes the file interface, key definitions for tables and the bit and bit mask operations. The introduction of Unicode results in the following changes:

- The `OPEN DATASET` command was completely revised In the **file interface**. At least one of the additions `IN TEXT MODE ENCODING`, `IN BINARY MODE`, `IN LEGACY MODE`, or `IN LEGACY BINARY MODE` must be defined in a UP.

  In a US, you can only read and write files with `READ DATASET` and `TRANSFER` if the file to be edited was first opened explicitly. A runtime error is triggered if there is no `OPEN` statement for these statements.

  If the file was opened in `TEXT MODE`, only character type fields, strings and purely character-type structures are allowed for `READ DATASET dsn INTO f` for *f*, and the type is only checked at runtime.

  The LENGTH addition defines the length of the data record in characters in TEXT MODE. In all other cases it is defined in bytes.

- A syntax error is triggered in a UP for the obsolete statements **`LOOP AT`** `dbtab`, **`READ TABLE`** `dbtab`, and **`READ TABLE`** `itab` if the key is purely character-type.

  A syntax or runtime error is triggered for the READ TABLE itab statement if the standard key of the internal table contains types X or XSTRING. With this READ variant, the key that is actually used is determined by hiding all the components filled with spaces. The comparison with SPACE must be allowed in a UP.

  A syntax or runtime error is also triggered when you access the database with generic key if the key is not purely character-type. This affects the following commands:

  ```
  READ TABLE dbtab ...SEARCH GKEQ ...
  READ TABLE dbtab ...SEARCH GKGE ...
  LOOP AT dbtab ...
  REFRESH itab FROM TABLE dbtab.
  ```

  The actual table key is determined by truncating the closing spaces of the database key in these statements. In a UP you must make sure that all the components of the key can be compared with SPACE.

- Until now, there was a check in **bit statements** `SET BIT i OF f [TO g]` and `GET BIT i OF f [INTO g]` to see if field *f* has character type, where normally X fields, X strings and flat structures were also considered to have character type. This no longer is meaningful in a UP because on the one hand types X and XSTRING are no longer considered to have character-type, and on the other hand bit-by-bit access to fields or structures of character-type is no longer platform-independent. In an UP, field *f* must therefore be of type X or XSTRING for these operations.

Until now, all numeric types and thus all character types were allowed for the left operand *f* in the **bit mask operations** `f O x`, `f Z x` and `f M x`. Operand *f* now must have type X or XSTRING in a UP. In a UP, the operand *f* must have the type X or XSTRING.

- There are certain restrictions in UP for the following statements when **adding field strings**:

```
ADD n1 THEN n2 UNTIL nz [ ACCORDING TO sel ] GIVING m ...
ADD n1 THEN n2 UNTIL nz TO m [ RANGE str ].
```

1. Operands *n1, n2,* and *nz* must have compatible types.
2. The distance between *nz* and *n1* must be an integer multiple of the distance between *n2* and *n1*.
3. There is a syntax or runtime error if fields *n1*, *n2* and *nz* are not in a structure. Either the syntax check must be able to recognize this fact or its valid range must be marked explicitly with a RANGE addition.
4. The system ensures that the RANGE area is not left at runtime.

```
ADD n1 FROM i1 GIVING m [ RANGE str ].
```

1. The field *n1* must lie within a structure. Field *n1* must lie within a structure that must be explicitly defined with a RANGE addition if the syntax check cannot recognize this fact.
2. This variant also checks at runtime if n1 and the addressed values lie within the structure.

- Loops with the **VARY or VARYING** addition also cause Unicode problems because on the one hand you cannot be sure to access the contents of memory with the correct type and on the other hand memory could be overwritten inadvertently.

```
DO ... VARYING f FROM f1 NEXT f2.
```

Fields *f*, *f1* and *f2* must have compatible types in this statement. To prevent storage contents from being overwritten, a RANGE for valid accesses is implicitly or explicitly introduced for the following statements:

```
DO ... TIMES VARYING f FROM f1 NEXT f2 [ RANGE f3 ].
WHILE ... VARY f FROM f1 NEXT f2       [ RANGE f3 ].
```

A syntax or runtime error is also triggered if *f1* or *f2* are not included in *f3*. If the RANGE addition is missing, it is implicitly defined as follows with `FROM f1 NEXT f2`:

1. If the syntax check recognizes that both *f1* and *f2* are components of the same structure, the valid RANGE range is defined from the smallest structure containing *f1* and *f2*.
2. There is a syntax error if the syntax check recognizes that *f1* and *f2* do not belong to the same structure.

3. A valid range must be defined explicitly with RANGE if the syntax check cannot recognize that *f1* and *f2* are related.

   If a deep structure is defined as RANGE addition, there is a check that there are no field or object references, tables or strings within the range just being scanned each time the loop is executed.

- When **subroutines are generated** automatically with the `GENERATE SUBROUTINE POOL itab NAME name` statement, the generated program inherits the contents of the Unicode flag of the generating program.

- In the **INSERT REPORT** statement, you can set the Unicode flag explicitly at runtime using the `UNICODE ENABLING uc` addition. If this addition is omitted, the program is characterized as follows:

  1. A Unicode program generates a Unicode program
  2. A non-Unicode program generates a non-Unicode program
  3. A non-Unicode program becomes a Unicode program after it has been overwritten by a Unicode program
  4. A Unicode program remains a Unicode program after it has been overwritten by a non-Unicode program

- In the `GET/SET PARAMETER ID pid FIELD f` statement, f must be character-type. You can use the EXPORT and IMPORT statements to store non-character-type fields and structures

5. New ABAP Statements for Unicode

5.1 String Processing for Byte Strings

- The X variants of the string statements are distinguished from the character string commands with the IN BYTE MODE addition. The IN CHARACTER MODE addition can be used optionally for the character string variants.

  If you define the IN BYTE MODE addition, only X fields and X strings are allowed as arguments. There is a syntax or runtime error if arguments of a different type are passed.

```
CLEAR ... WITH                IN BYTE MODE
CONCATENATE x1 ... xn INTO x  IN BYTE MODE
FIND ...                      IN BYTE MODE
REPLACE ...                   IN BYTE MODE
REPLACE f WITH g INTO h       IN BYTE MODE
SEARCH x FOR x1               IN BYTE MODE
SHIFT x ...                   IN BYTE MODE
SPLIT ...                     IN BYTE MODE
```

The full length of the X fields are always used in all string commands for byte strings, so that bytes with contents X'00' are never truncated at the end of the field. There are no variants that use search patterns for the FIND, SEARCH and REPLACE statements with the IN BYTE MODE addition.

- The **string length** for byte strings can be defined with the XSTRLEN function. XSTRLEN returns the current length for X strings and the defined length in bytes for X fields, where null bytes at the end of fields are also counted.
- The X variants of the **string comparison operations** are distinguished from the string variants by the BYTE prefix. Only X fields and fields of type XSTRING are allowed as arguments for these operations.

  BYTE-CO

  BYTE-CN

  BYTE-CA

  BYTE-NA

  BYTE-CS

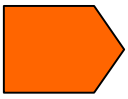
  BYTE-NS

5.2 Determining the Length and Distance

The field length or the distance between two fields can be determined with the DESCRIBE FIELD and DESCRIBE DISTANCE statements. The IN BYTE/CHARACTER additions must be defined for the following variants under Unicode:

- `DESCRIBE FIELD f LENGTH len IN BYTE MODE.`

  The length of field *f* is determined in bytes and passed to field *len*.

  This variant returns the length of the reference, and not the length of the contents, for all internally referenced data types (strings as well as field and object references). Only variant IN BYTE MODE is therefore allowed for such fields.

- `DESCRIBE FIELD f LENGTH len IN CHARACTER MODE.`

  The length of field *f* is returned here in characters if the argument is purely character-type. *f* may not have type STRING. The relevant check is performed statically and dynamically and causes a syntax or runtime error, depending on the type.

- `DESCRIBE DISTANCE BETWEEN a AND b INTO x IN BYTE MODE.`

  This statement returns the distance between fields *a* and *b* in bytes in field *x*.

- `DESCRIBE DISTANCE BETWEEN a AND b INTO x IN CHARACTER MODE.`

  This statement returns the distance in characters, where the result always refers to the start of the field. Whether or not the distance between the arguments can be divided by the platform-specific length in characters and whether both fields *a* and *b* have the right alignment is only checked at runtime. Otherwise there is a runtime error.

5.3 Assignments to Field Symbols

Until now the ASSIGN statement made it possible to define addresses past field limits by specifying the offset or length. There was only a runtime error when addressing past the limits of the data segment. Cross-field accesses to the offset/length in an ASSIGN, for example, could be used to edit repeating groups.

With Unicode, however, problems occur since it is not possible to ensure that cross-field offset or length definitions can be interpreted as bytes or characters in an identical and meaningful manner in both a US and an NUS. For this reason, the ASSIGN statement was enhanced with the RANGE and INCREMENT additions while the CASTING addition now supports all variants of this statement. The RANGE addition is offered for all valid variants of ASSIGN and can be combined with the CASTING addition.

- `ASSIGN feld1  TO <fs> RANGE feld2.`

  This addition explicitly sets the limits of the range. It makes it possible to define addresses past field limits, for example to edit repeating groups with the ASSIGN INCREMENT statement.

  1. The *field limits* of *field2* are used as the range for <*fs*>.
  2. In a UP, the limits specified by the RANGE definition must include the range limits that would otherwise result from the rules described above.
  3. If the memory area of *field1* is not completely contained in *field2*, there is a catchable runtime error.
  4. Field *field2*, which defines the range, may also be deep. Repeating groups with deep types therefore can also be processed.

- `ASSIGN field INCREMENT n TO <fs>.`

  The field symbol is incremented by *n* times the length of *field*, starting with the position defined by *field*.

  First the range for the access is defined from the length of *field* and the INCREMENT definition of the range for the access as defined by ASSIGN fld+n*sizeof[field] (sizeof[fld]) TO <fs>. The addressed range must lie within the range limits. If it is not possible to make the assignment because the range limits were violated, SY-SUBRC is set to > 0 and the field symbol is not changed.

The range limits for ASSIGN field INCREMENT n TO <fs> are defined in exactly the same way as ASSIGN field TO <fs>. The definition of the INCREMENT therefore has no effect on the definition of the range limits.

```
* Loop through an elementary field

DATA: c(10) TYPE C VALUE 'abcdefghij'.
FIELD-SYMBOLS: <cf> TYPE C.

 ASSIGN c(2) TO <cf>.                "Range limits c =
 DO 5 TIMES.                         "Field boundaries of cf
   WRITE / <cf>.
   ASSIGN  <cf> INCREMENT 1 TO <c>. "Same limits <c>
*  ASSIGN  <c>+2 TO <cf>.            "Like ASSIGN INCREMENT
 ENDDO.
```

```
  * Structured repeating group

  TYPES: BEGIN OF comp,
           f1 type string,
           ...
         END OF comp.
  DATA:  BEGIN OF stru,
           x1(1) TYPE x,
           k1    TYPE comp,
           k2    TYPE comp,
           k3    TYPE comp,
           k4    TYPE comp,
         END OF stru.
  FIELD-SYMBOLS: <comp> TYPE comp.

  ASSIGN stru-k1 TO <comp> RANGE stru.

  * Specify that range limits are to exceed field
  * boundaries
  DO 4 TIMES.
    ...
    ASSIGN <comp> INCREMENT 1 TO <comp>.
  ENDDO.
```

```
* Dynamic access to an
* element in a repeating group

DATA: BEGIN OF stru,
        x1(1) TYPE x,
        k1    TYPE comp,
        k2    TYPE comp,
        k3    TYPE comp,
        k4    TYPE comp,
      END OF stru,
      incr    TYPE i.
FIELD-SYMBOLS: <comp> TYPE comp.

incr = 4 - 1.
ASSIGN stru-k1 INCREMENT incr TO <comp> RANGE stru.
...
* <comp> now points to stru-k4
```

- The **CASTING** addition is allowed for all variants of the ASSIGN statement:

```
ASSIGN field TO <fs> CASTING.
ASSIGN field TO <fs> CASTING TYPE  type.
ASSIGN field TO <fs> CASTING TYPE (typename)
ASSIGN field TO <fs> CASTING LIKE  fld.
ASSIGN field TO <fs> CASTING DECIMALS dec.
```

  You can use ASSIGN ... CASTING to look at the contents of a field as a value of another type using a field symbol . One application for this statement would be to provide different views on a structure with casts on different types.

  One wide-spread ABAP technique is to use C fields or structures as containers for storing structures of different types that are frequently only known at runtime. The components of the structure are selected with offset/length accesses to the container. Since this technique no longer works with Unicode, you can also look upon an existing memory area as a container with the suitable type definition using a field symbol with the ASSIGN ... CASTING statement. In the next example, a certain field of database table X031L is read, whereby the field and table names are only defined at runtime.

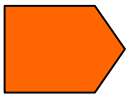
```
*  Read a field from the table X031L

PARAMETERS:
    TAB_NAME   LIKE SY-TNAME,           "Table name
    TAB_COMP   LIKE X031L-FIELDNAME,    "Field name
    ANZAHL     TYPE I DEFAULT 10.       "Number of lines
DATA:
  BEGIN OF BUFFER,
    ALIGNMENT  TYPE F,                  "Alignment
    C(8000)    TYPE C,                  "Table content
END OF BUFFER.

FIELD-SYMBOLS: <WA>   TYPE ANY,
               <COMP> TYPE ANY.

* Set field symbol with appropriate type
* to buffer area

ASSIGN BUFFER TO <WA> CASTING TYPE (TAB_NAME).
SELECT * FROM (TAB_NAME) INTO <WA>
  UP TO number of ROWS.
  ASSIGN COMPONENT TAB_COMP OF STRUCTURE <WA> TO <COMP>.
  WRITE: / TAB_COMP, <COMP>.
ENDSELECT.
```



Until now, in the `ASSIGN field TO <f> CASTING...` statement, the system checked to ensure that field was at least as long as the type that was assigned to the field symbol, <f>. (Field symbols can either be typed at declaration or the type specified in an ASSIGN statement using CASTING TYPE). The syntax check is now more thorough. Now, you can only assign the field field provided it is at least as long - in both the UP and the NUP - as the type assigned to the field symbol <f>. Otherwise, the system returns a syntax error. At runtime, the system only checks to see whether or not the lengths are compatible in the current system (as before).

If the field type or field symbol type is a deep structure, the system also checks that the offset and type of all the reference components match in the area of field that is covered by <f>. The syntax check is now more thorough. Now, the system checks that these components must be compatible in all systems, whether they have a one-byte, double-byte, or four-byte character length. At runtime, the system only checks to see whether or not the reference components are compatible in the current system.

In US, in the `ASSIGN str TO <f> TYPE C/N` and `ASSIGN str TO <fs> CASTING TYPE C/N` statements, the length of str may not always be a multiple of the character length, in which case the program aborts at runtime.

5.4 Includes with Group Names

By redefining structures using INCLUDES with group names, you can select groups of fields symbolically beyond the boundaries of individual components. You can assign group names either in the ABAP Dictionary or in an ABAP program. The statement takes one of the following two forms:

```
INCLUDE TYPE      t1 AS grpname[RENAMING WITH SUFFIX suffix].
INCLUDE STRUCTURE s1 AS grpname[RENAMING WITH SUFFIX suffix].
```

By adding a group name in an Include (in the ABAP Dictionary) or with the AS *grpname* addition in an ABAP program, you can then address the area of the structure defined in the Include symbolically (using the group name). The following example groups together parts of an ABAP structure, which is then passed to a subprogram:

```
* Using Includes with Group Names
TYPES: BEGIN OF name,
         first name(20)   TYPE C,
         surname(30)  TYPE C,
       END OF name.

TYPES: BEGIN OF person,
         sex(1) TYPE X.
         INCLUDE TYPE name AS pname.
TYPES:   alter         TYPE I,
         weight       TYPE P,
       END OF person.
DATA:  s2 TYPE person.

PERFORM use_name USING s2-pname.
```

5.5 Creating Data Objects Dynamically

- CREATE DATA allows you to create fields in a pre-defined or user-defined data type. The statement has the following variants:

  ```
  CREATE DATA dref TYPE  typ.
  CREATE DATA dref TYPE (typname).
  CREATE DATA dref LIKE  feld.
  CREATE DATA dref TYPE LINE OF itab.
  CREATE DATA dref LIKE LINE OF itab.
  ```

  In the following example, a specific field is read from database table X031L. Note that neither the field name nor the table name is known until runtime:

```
*  Read a field from the table X031L

PARAMETERS:
  TAB_NAME   LIKE SY-TNAME,             "Table name
  TAB_COMP   LIKE X031L-FIELDNAME,      "Field name
  NUMBER     TYPE I DEFAULT 10.         "Number of lines
FIELD-SYMBOLS: <WA>   TYPE ANY,
                <COMP> TYPE ANY.
DATA: WA_REF TYPE REF TO DATA.

CREATE DATA WA_REF TYPE (TAB_NAME).     "Suitable work area
ASSIGN WA_REF->* TO <WA>.
SELECT * FROM (TAB_NAME) INTO <WA>
  UP TO NUMBER ROWS.
  ASSIGN COMPONENT TAB_COMP OF STRUCTURE <WA> TO <COMP>.
  WRITE: / TAB_COMP, <COMP>.
ENDSELECT.
```

- Another variant of CREATE DATA allows you to create **table objects** at runtime. The line type and table key can be entered statically or dynamically.

```
CREATE DATA dref (TYPE [STANDARD|SORTED|HASHED] TABLE
   OF (LineType | (Name) |REF TO DATA | REF TO Obj))
   | (LIKE [STANDARD|SORTED|HASHED] TABLE OF LineObj )
   [ WITH (UNIQUE|NON-UNIQUE)
   ( KEY (k1 ... kn | (keytab) | TABLE_LINE )
   | DEFAULT KEY )  ]
   [ INITIAL SIZE m ]
```

The following constraints apply to this variant:

1. *m* is a variable or a constant without a sign, whose content at runtime must be of the type NUMLIKE.
2. *keytab* is a table of the type CHARLIKE, which must not be empty, and whose components must not contain any offset, length, or overlapping key entries. You can use the TABLE_LINE addition, if the table contains only one line.
3. The system returns a syntax error if either the type, or line declaration *and* the key declaration are static.
4. If you do not define a key, the system uses the DEFAULT-KEY.

- You can also use the **basic generic types**, C, N, X, and P with the CREATE DATA statement. You can specify the length and number of decimal places (for type P) using additions.

```
CREATE DATA dref TYPE (t | (typeName))
   [ LENGTH   len ]
   [ DECIMALS dec ].
```

You can only use the LENGTH addition for types C, N, X, and P and you must always include it **after** the TYPE keyword. A catchable runtime error occurs if

you do not comply with syntax conventions when entering LENGTH or DECIMALS values.

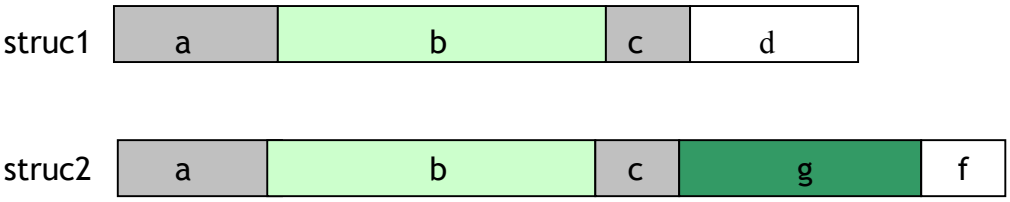
5.6 Assigning Fields Dynamically

In previous releases in the MOVE-CORRESPONDING struc1 TO struc2 statement, the field types of both structures had to be known at the time they generated. This constraint no longer applies under Unicode.

In an NUS until now, there was no problem assigning structures with different Unicode fragment views using a MOVE statement. In a US, such assignments will cause a runtime error, even if both structures start with the same types:

```
BEGIN OF struc1,
  a(2) TYPE C,
  b    TYPE I,
  c(1) TYPE C,
  d(4) TYPE X,
END OF struc1.
```

```
BEGIN OF struc2,
  a(2) TYPE C,
  b    TYPE I,
  c(1) TYPE C,
  g(6) TYPE P,
  f(2) TYPE X,
END OF struc2.
```



For example, the content of *struc2-[a,c]* (which starts with fields of the same type) is assigned to *struc1*. Until now, you could use a simple MOVE statement (although this could mean that the remainder of *struc1* subsequently did not contain meaningful values). However, in Unicode, you must the `MOVE-CORRESPONDING struc2 TO struc1`; otherwise, a runtime error occurs (because you are attempting to MOVE struc2 to a structure with a different fragment view). You obtain the same result in Unicode, if the field names of the start of both structures are identical at runtime.

5.7 Storing Data as a Cluster

New variants of the IMPORT and EXPORT statements are available to support heterogeneous Unicode environments. They allow you to store data as a cluster in an XSTRING in a cross-platform format You can use the following variants:

- `EXPORT {`p_i `=` $dobj_i$ `|` p_i `FROM` $dobj_i$ `} TO DATA BUFFER dbuf.`
  `EXPORT (itab) TO DATA BUFFER dbuf.`

  Stores the objects $dobj_1$ ... $dobj_n$ (fields, flat structures, complex structures, or tables) as a cluster in the data buffer, *dbuf*, which must be of type XSTRING.

```
TYPES:
   BEGIN OF ITAB_TYPE,
     CONT TYPE C LENGTH 4,
   END OF ITAB_TYPE.
```

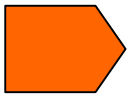
```
DATA:
  XSTR TYPE XSTRING,
  F1 TYPE C LENGTH 4,
  F2 TYPE P,
  ITAB TYPE STANDARD TABLE OF ITAB_TYPE.

EXPORT P1  = F1
       P2  = F2
       TAB = ITAB TO DATA BUFFER XSTR.
```

- New addition: `... CODE PAGE HINT f1`
  You use this addition in conjunction with ambiguous code pages in the `EXPORT obj1 ... objn TO [ DATA BUFFER | DATABASE | DATASET ]` statement. It specifies the code page, f1, which is to be used to interpret the import data.

- `IMPORT {pi = dobji | pi FROM dobji } FROM DATA BUFFER dbuf.`
  `IMPORT (itab) FROM DATA BUFFER dbuf.`

  Imports the data objects *dobj1 ... dobjn* (fields, flat structures, complex structures, or tables) from a data cluster in the data buffer entered, dbuf, which must be of type XSTRING. The system reads all the data that was previously stored in the data buffer dbuf using the EXPORT ... TO DATA BUFFER statement. Again, the system does not check that the structures in the EXPORT and IMPORT statements match.

  If the data objects are specified dynamically, the parameter list in the two-column index table, **itab**, is passed. The columns in this table must be of type C or STRING. The first column of itab contains the name of each parameter, while the second lists the data objects. If the first column of itab is empty, an exception that cannot be handled occurs, with the runtime error DYN_IMEX_OBJ_NAME_EMPTY.

5.8 File Interface

In the file interface, the OPEN DATASET statement has been completely overhauled and the following enhancements added for USs:

- `OPEN DATASET dsn IN TEXT MODE.`
  The file is opened so that it can be read or written to line-by-line. Final space characters are deleted in this mode.

  1. Addition: `ENCODING (DEFAULT | UTF-8 | NON-UNICODE)`

  The keyword ENCODING specifies the character set used to edit the data. In a US the DEFAULT is UTF-8, while in an NUS it is NON-UNICODE. If you specify NON-UNICODE, the system uses the non-Unicode codepage that suits the logon language or the language set using SET LOCALE LANGUAGE in a non-Unicode system.

2. Addition ... `REPLACEMENT CHARACTER rc`

Specifies the replacement character that is used if a character is not available in the target character set when the file is converted. The default replacement character is the hash symbol (#).

- `OPEN DATASET dsn IN BINARY MODE.`

  The file is opened to be read or written to, without any line breaks. In both the US and the NUS, the exact content of memory is copied.

- `OPEN DATASET dsn IN LEGACY TEXT MODE [(LITTLE | BIG) ENDIAN ] [ CODE PAGE cp ].`

  The file is opened so that it can be read or written to line-by-line, in a format compatible with TEXT MODE in the NUS.

  You use the ENDIAN addition to specify the byte order that the system will use to process numbers of type I or type F. If you omit this addition, the operating system of the application server specifies the byte order. If the byte order declared differs from that used by the operating system, the data is converted as appropriate for the statements READ DATASET and TRANSFER. You can also specify a REPLACEMENT CHARACTER, *rc*, in this statement.

  The CODE PAGE addition specifies the code page used to display text from the file *dsn*. If this addition is missing, the system uses the code page used to read or write to the file.

- `OPEN DATASET dsn IN LEGACY BINARY MODE [(LITTLE|BIG) ENDIAN)] [ CODE PAGE cp ].`

  The file is opened to be read or written to without any line breaks, in a format compatible with NUS BINARY MODE. The additions ENDIAN and CODE PAGE are used as described above. You can also specify a REPLACEMENT CHARACTER *rc* in this statement.

For each of the above variants, you can use the IGNORING CONVERSION ERRORS addition to make the system suppress conversion errors at runtime when reading or writing to a file. In general, reading or writing to a file causes a runtime error, unless this file has already been opened using an OPEN DATASET statement.

5.9 Uploading and Downloading Files

WS_UPLOAD and **WS_DOWNLOAD**, the function modules used until now are not part of the standard set of ABAP commands. They are used to display the file interface on the presentation server. WS_UPLOAD and WS_DOWNLOAD are not compatible with USs and have been replaced by **GUI_UPLOAD** and **GUI_DOWNLOAD**.
The new function modules, **GUI_UPLOAD** and **GUI_DOWNLOAD**, have an interface that also allows you to write Unicode format to the local hard drive. For a description of these interfaces, refer to the documentation for each function module, available under *SAP Easy Access -> Development -> Function Builder -> Goto -> Documentation*.

5.10 Generic Types for Field Symbols and Parameters

The following new generic data types are now available for assigning types to parameters and field symbols:

- **SIMPLE** is compatible with all the types that are compatible with CLIKE, XSEQUENCE, or NUMERIC – that is, with all elementary types including STRING and XSTRING. Assigning the generic type SIMPLE ensures that parameters or field symbols can be displayed using WRITE or used in arithmetic operations. However, conversion errors may occur when parameters and field symbols of this type are used in arithmetic operations, depending on the content – for example, a C field is passed as an actual parameter and the field content cannot be interpreted as a number.
- **CLIKE** is compatible with the types C, N, D, T, STRING, and purely character-type structures. In NUSs, CLIKE is also compatible with the elementary types X and XSTRING. Assigning the generic type CLIKE ensures that parameters and field symbols can be used for all operations string processing operations, such as those in the CONCATENATE, FIND, and REPLACE statements. You also guarantee that the system counts in characters when performing offset-based or length-based accesses. These are allowed in the range of the entire field, or within the current string length for STRING-type components.
- **CSEQUENCE** is compatible with the types C and STRING
- **XSEQUENCE** is compatible with the types X and XSTRING. Assigning this generic type guarantees that parameters and field symbols can be used in byte processing operations, such as in the CONCATENATE ... IN BYTE MODE statement.
- **NUMERIC** is compatible with the types I, P, and F. It is also compatible with two types that are only available in the Dictionary - INT1 (single-byte integer) and INT2 (double-byte integer). Note that type N is not compatible with the generic type NUMERIC. Assigning the generic type NUMERIC ensures that parameters and field symbols can be used in arithmetic operations without type or conversion errors occurring.

5.11 Formatting Lists

- **Introduction**

  The WRITE statement writes the content of data objects into a list. When using the WRITE statement during the write process, the output is saved in the list buffer and displayed from there when calling the list. When using WRITE to output a data object, an output length is determined implicitly or explicitly, and the implicit output length depends on the data type. The output length defines:

  - The number of spaces (or the memory space) available for characters in the list buffer.
  - The number of columns (or cells) available in the actual list.

If the output length is shorter than the length of the data object, its content is truncated according to specific rules when writing the data into the list buffer; the value loss for numeric fields is indicated by the * character. When displaying or printing a list, the contents stored in the list buffer are transferred to the list as follows:

- In NUS, each character requires exactly as much space in the list buffer as it does columns in the list. In single-byte systems, a character occupies one byte in the list buffer and one column in the list, while a character in multi byte systems that occupies multiple bytes in the list buffer occupies the same number of columns in the list. Therefore, in NUS, all characters stored in the list buffer are displayed in the list.
- In US, each character generally occupies one space in the list buffer. However, a character can occupy more than one column in the list – this applies especially to East Asian characters. Since the list has only the same number of columns available as there are spaces in the list buffer, in this case, the number of characters that can be displayed in the list is lower than the number of the ones stored in the list buffer. The list output is truncated accordingly, the page is justified, and the indicator > or < is inserted. The complete content can be displayed in a list by choosing *System→ List → Unicode Display*.

These are the reasons why the horizontal position of the list cursor is equal to the output column of a displayed or printed list in NUS only. In US, this is guaranteed only for the minimum and maximum output limits of the individual outputs.

- **Rules for the WRITE Statement**

To avoid inadvertent truncation, the rules for the WRITE statement were adjusted and extended in UP.

1. WRITE Statement with Implicit Output Length

In UP, the WRITE statement without an explicit specification of the output length has the same behavior as in NUP for all data objects to be output with the exception of text field literals and the data objects of the type *string*. Therefore, the number of displayed characters in the list may be lower than the number of characters stored in the list buffer.

In the case of text field literals and data objects of the type *string*, it is assumed that all characters are to be displayed. Therefore, the characters contained in the data object are used to calculate the implicit output length in such a way that it equals the number of columns required for the list. If this output length is longer than the data object length, the superfluous spaces are filled with blank characters when writing the data into the list buffer. When the characters are displayed in the list, these blank characters are truncated because the display of the characters fits the output length exactly.

2. WRITE Statement with Implicit Output Length

When the WRITE statement has a numeric data object specified as an explicit output length after the AT addition, the value of this number is used as the output length in US and NUS. In US, the number of characters displayed in the list can differ from the number of characters stored in the list buffer. As of Release 6.20, the output length can be specified - besides using numeric data objects – in the following way:

1. WRITE AT (*) ...
   - For data objects of the types *c* and *string*, the output length is set to the number of columns that is required for the list to display the complete content; closing blank characters for the type *c* are ignored. For data objects of the *string* type, this setting is the same as the implicit length.
   - For data objects of the types *d* and *t*, the output length is set to 10 and 8.
   - For data objects of the numeric types *i*, *f*, and *p*, the output length is set to the value that is required to output the current value including the thousand separator; the value depends on the possible use of the additions CURRENCY, DECIMALS, NO-SIGN, ROUND, or UNIT.
   - For data objects of the types *n*, *x*, and *xstring*, the implicit output length is used.
2. WRITE AT (**) ...
   - For data objects of the type *c*, the output length is set to double the length of the data object, and for data objects of the type *string*, the output length is set to double the number of the contained characters.
   - For data objects of the types *d* and *t*, the output length is set to 10 and 8.
   - For data objects of the numeric types *i*, *f*, and *p*, the output length is set to the value that is required to output the maximum number of values of this type including the thousand separator; the value depends on the possible additions CURRENCY, DECIMALS, NO-SIGN, ROUND, or UNIT.
   - For data objects of the types *n*, *x*, and *xstring*, the implicit output length is used.

- **New Additions for GET/SET CURSOR FIELD/LINE**

  The additions DISPLAY OFFSET and MEMORY OFFSET take into account that data objects may require different lengths when displayed in a list (display) or when stored in the list buffer (memory) during the intermediate storage. In Unicode pages, there are, for example, characters that take up one space in the list buffer but require two output columns when displayed.

  Accordingly, for the SET CURSOR { FIELD f | LINE l } statement, using the DISPLAY OFFSET addition sets the cursor in the output while using MEMORY OFFSET off sets it in the list buffer.

  Similarly, using the GET CURSOR { FIELD f | LINE l } statement with the DISPLAY OFFSET off addition transfers the cursor position within the displayed field into the data object *off*. On the other hand, when using the MEMORY OFFSET off

addition, the cursor position in the list buffer is transferred into the data object *off*. The DISPLAY addition is the default setting and can therefore be omitted.

- **Class for Formatting Lists**

  The class CL_ABAP_LIST_UTILITIES was introduced for calculating output lengths, converting values from the list buffer, and defining field limits. Using the return values of its methods, the column alignment on ABAP lists can be programmed, even if they contain East Asian characters.

- **List Settings**

  The list objects can be displayed in different output lengths by choosing *System → List → Unicode Display* and setting the desired length. This is especially advantageous for screen lists in US whose outputs were truncated – this is indicated by the indicator > or <.

- **Recommendations**

  To ensure that lists have the same appearance and functions in US and NUS, it is recommended that you adhere to the following rules when programming lists:

  - Specify a sufficient output length
  - Do not overwrite parts of a field
  - Do not use WRITE TO with RIGHT-JUSTIFIED or CENTERED with a subsequent WRITE output
  - For self-programmed horizontal scrolling using the **SCROLL** statement, only the minimum or maximum limit of the data objects that are output should be specified, because in US, it is guaranteed for these limits only that the positions in the list buffer and the displayed list match

6. New Classes for Unicode

6.1 Determining Field Properties

The class **CL_ABAP_CHAR_UTILITIES** provides attributes and methods that affect the properties of single characters. The components of this class are all static and public. The attributes are write-protected and are initialized in the class constructor method.

- **CHARSIZE attribute**

  ```
  CHARSIZE TYPE I.
  ```

  The CHARSIZE attribute declares the length of a C(1) field in bytes – that is, one byte in an NUS, and either two or four bytes in a US.

- **MINCHAR, MAXCHAR attributes**

  ```
  MINCHAR(1) TYPE C.
  ```

  In an NUS, MINCHAR contains the Unicode character X'00'. In a US, it contains the Unicode character U+0000.

  ```
  MAXCHAR(1) TYPE C.
  ```

In an NUS, MAXCHAR contains the Unicode character X'FF'. In a US, it contains the Unicode character U+FFFD.

You can use these values only when performing binary comparisons in the following ABAP statements:

1. SORT without the AS TEXT addition
2. IF with the <, >, <=, and >= operators
3. IF f1 BETWEEN f2 AND f3.
4. IF f in sel.

Bear in mind that the results of binary comparisons are platform-specific. For example, the character sequence in the ISO-8859-1 character set is 1 < A < Z < a < Ä < ü, whereas in EBCDIC it is Ä < a < A < ü < Z < 1.

You can use CLEAR feld WITH CL_ABAP_CHAR_UTILITIES=>MAXCHAR to fill a field with a value that is greater than or equal to all the strings. This also works in a multi-byte NUS.

MINCHAR and MAXCHAR are not usually valid characters in the current character set. In particular, you cannot use TRANSLATE f TO UPPER CASE in conjunction with these attributes, since the result would be undefined. The same constraint applies to all operations that implicitly convert characters to upper case – such as CS, NS, CP, NP or SEARCH. Moreover, you cannot perform character set conversions with the MINCHAR or MAXCHAR attribute.

In addition, MINCHAR may not always be displayed on screens as the number sign (#). In many cases, it is treated as the end of a text field.

- **Byte order**

  Conversion classes, implemented with the file interface, are used to convert numeric data types and Unicode from little-endian to big-endian format and vice versa. The attributes described in this section are used to recognize and set byte order marks in X containers.

  ENDIAN attribute

  ```
  ENDIAN(1) TYPE C.
  ```

  The value ***B*** indicates big-endian format, while ***L*** denotes little-endian, in either a US or an NUS (as in the class-based file interface).

  Constants BYTE_ORDER_MARK_

  ```
  BYTE_ORDER_MARK_LITTLE(2) TYPE X
  BYTE_ORDER_MARK_BIG(2)    TYPE X
  BYTE_ORDER_MARK_UTF8(3)   TYPE X
  ```

  The values for these constants are X'FFFE' for little-endian and X'FEFF' for big-endian, in both Unicode and NUSs. BYTE_ORDER_MARK_UTF8 contains the UTF8 display of U+'FEFF'.

  The class-based file interface writes a byte order mark whenever a UTF16 or UCS2 text is opened to be written to. When a text file is opened to be read or appended, a byte order mark is used to specify the endian format. However, the byte order mark is not written to the target fields when the file is read.

- **NEWLINE, CR_LF, and HORIZONTAL_TAB attributes**

```
NEWLINE(1)              TYPE C
CR_LF(2)                TYPE C
HORIZONTAL_TAB(1)       TYPE C
```

  These attributes contain the appropriate platform-specific control characters.

6.2 Converting Data

These classes are used to convert ABAP data from the system format to external formats and vice versa. During this conversion process, character-type data may be converted to another character set, while numeric-type data may be converted to another byte order (or endian format). You must use a container of type X or XSTRING for data in an external format.

Character sets and endian formats are also converted by the file interface (OPEN DATASET with new additions), RFCs, and the function modules GUI_DOWNLOAD and GUI_UPLOAD. The classes described below are available for those special cases where the possibilities offered by conversion are insufficient. Since these classes work with containers of types X and XSTRING, these containers can be copied unconverted by the file interface (OPEN DATASET with new additions), RFCs, and the function modules GUI_DOWNLOAD and GUI_UPLOAD. These classes replace the following two statements:

```
TRANSLATE c ...FROM CODE PAGE     g1 ... TO CODE PAGE     g2
TRANSLATE f ...FROM NUMBER FORMAT n1 ... TO NUMBER FORMAT n2
```

For a detailed description, see the class documentation in the Class Builder. The following classes are available:

CL_ABAP_CONV_IN_CE:
Reads data from a container and converts it to the system format. You can also fill structures with data.

CL_ABAP_CONV_OUT_CE:
Converts data from the system format to an external format and writes it to a container.

CL_ABAP_CONV_X2X_CE:
Converts data from one external format to another.

7. RFC and Unicode

The Remote Function Call (RFC) interface has also changed as part of the conversion to Unicode. The following remarks give a very brief outline of the general guidelines.

So that the RFC works in both USs and NUSs, an additional attribute has been added to the destination. This attribute specifies whether the RFC is to run in a US or a

NUS. If an ABAP program uses function modules that change the RFC destination, this new attribute must be filled using another required parameter.

When text data is transferred in an RFC, the following new exceptions may occur:

- A text cannot be converted from the source code page to the target code page because the corresponding character in the target code page is not available.
- The field in the target system is shorter than the field in the source system. This may occur if data from a US is transferred to a multibyte system.
- In principle, you can assume that text data has a different length in the sending system than in the receiving system. For this reason, we recommend that you use strings wherever possible. That is, you should try to avoid specifying the length of text data. You can, however, send binary data whose length is specified, without any problems.

8. Further Measures

Until now, when the system checked or converted data types, the class hierarchy and interface implementation relation were only taken into account in the case of single fields with class and interface references. Structures containing references were not checked consistently. For this reason, when deep structures are being converted or when type and comparison checks are being performed on them, the following generalizations are made:

1. *Convertibility and type compatibility for Importing parameters:*
   - Elementary components of structures must match exactly.
   - The class hierarchy and interface implementation relation are also taken into account in the case of interface and class references used as sub-components.
   - Table components are mutually compatible if the line type is type-compatible and convertible. That means that tables can have different access types. *The line types must be convertible – for example, the structure component Table over Integer is compatible with the component Table over Float.*

2. Comparability:
   - Elementary components must match exactly.
   - Comparability can be performed on all classes and interface reference components.
   - Table components are comparable if their line type is comparable.

9. Other Sample Programs for the Conversion to Unicode

9.1 Assignment Between Structures: Conversion Using Includes with Group Names

Before the conversion to Unicode

```
types: begin of T_STRUC,
          F1     type c,
          F2     type c,
          F3     type i,
          F4 type p,
       end of T_STRUC.

data: begin of STRUC1.
        include  type T_STRUC.
data:   F5 type x.
data: end of STRUC1.

data: begin of STRUC2.
        include  type T_STRUC.
data:   F6 type p.
data: end of STRUC2.

STRUC1 = STRUC2.                    ← Unicode error
```

In this case, it is assumed that only the content of the includes is to be assigned - that is the components F1 to F4. Until now, it was tolerated that the component F5 is overwritten with a meaningless value.

After the conversion to Unicode

By introducing group names for the includes, this code segment can be converted easily when assigning structures.

```
types: begin of T_STRUC,
          F1     type c,
          F2     type c,
          F3     type i,
          F4 type p,
       end of T_STRUC.

data: begin of STRUC1.
        include type T_STRUC as PART1.
data:   F5 type x.
data: end of STRUC1.

data: begin of STRUC2.
        include type T_STRUC as PART1.
data:   F6 type p.
data: end of STRUC2.
```

```
STRUC1-PART1 = STRUC2-PART1.        ← ok
```

9.2 Assignment Between Structures: Conversion Using Offset-Based or Length-Based Accesses

Before the conversion to Unicode

```
data: begin of STRUC1,
         F1(10) type c,
         F2(20) type c,
         F3     type i,
         F4 type p,
      end of STRUC1,

      begin of STRUC2,
         C1(10) type c,
         C2(20) type c,
         C3     type x,
         C4     type f,
      end of STRUC2.
```

```
STRUC1 = STRUC2.                    ← Unicode error
```

In this example, it is assumed that only the content of the first two components C1 and C2 is to be passed to F1 and F2, because the following components F3 and F4 are overwritten by meaningless values.

After the conversion to Unicode

Since the initial part of the structures relevant for the assignment is purely character-type, the operands of the assignment can be selected using offset-based or length-based accesses:

```
...
STRUC1(30) = STRUC2(30).            ← ok
```

9.3 Offset-Based or Length-Based Access to Structures for Assignment: Use of Includes

Before the conversion to Unicode

When using offset-based or length-based accesses, areas of structures that are to be assigned to each other are selected. Since the structures do not start with character-type components, the offset-based or length-based accesses in Unicode-enabled programs are no longer allowed.

```
data: begin of STRUC1,
        F0     type x,
        F1(10) type c,
        F2(20) type c,
        F3     type i,
        F4 type p,
      end of STRUC1,

      begin of STRUC2,
        C0     type i,
        C1(10) type c,
        C2(20) type c,
        C3     type x,
        C4     type f,
      end of STRUC2.
```

STRUC1+1(30) = STRUC2+4(30). ← **Unicode error**

After the conversion to Unicode

By introducing includes with group names, the appropriate areas of the structures can be selected and assigned to each other.

```
types: begin of PART1,
         F1(10) type c,
         F2(20) type c,
       end of PART1,

       begin of PART 2,
         C1(10) type c,
         C2(20) type c,
       end of PART 2.

data: begin of STRUC1,
        F0      type x.
        include type PART1 as PART.
data:   f3      type i,
        f4      type p,
      end of STRUC1,

      begin of STRUC2,
        C0      type i.
        include type PART2 as PART.
data:   C3      type x,
        C4     type f,
      end of STRUC2.
```

STRUC1-PART = STRUC2-PART. ← **ok**

9.4 Offset-Based or Length-Based Access to Structures for Assignment: No Solution with Includes

Before the conversion to Unicode

```
data: begin of STRUC1,
        F0(1)  type x,
        F1(10) type c,
        F2(20) type c,
        F3     type i,
        F4 type p,
      end of STRUC1,

      begin of STRUC2,
        C0(1) type c,
        C1(10) type c,
        C2(20) type c,
        C3     type i,
        C4     type f,
      end of STRUC2,

      begin of STRUC3,
        G0(1) type c,
        G1(10) type c,
        G2(20) type c,
      end of STRUC3.
STRUC1+1(35) = STRUC2+1(35).        ← Unicode error
STRUC3       = STRUC2.
```

Conversion attempt: Introduction of INCLUDEs with group names

```
types: begin of PART1,
         F1(10) type c,
         F2(20) type c,
         F3     type i,
       end of PART1,

       begin of PART2,
         C1(10) type c,
         C2(20) type c,
         C3     type i,
       end of PART2.

data: begin of STRUC1,
         F0(1)   type x.
         include type PART1 as PART.
data:    F4      type p,
       end of struc1.

data: begin of STRUC2,
         C0(1)   type c.
```

```
          include type PART2 as PART.
data:     C4      type f,
        end of STRUC2,

        begin of STRUC3,
          G0(1) type c,
          G1(10) type c,
          G2(20) type c,
        end of STRUC3.

STRUC1-PART = STRUC2-PART.
STRUC3      = STRUC2.          ← New Unicode error
```

By introducing the includes, the problematic code segment can be converted to a Unicode-enabled code. However, the subsequent assignment causes a new Unicode problem: The assignment of STRUC2 to STRUC3 is no longer possible because in STRUC2 the include causes an alignment gap before the component C1. Due to this gap, the Unicode fragment views of STRUC2 and STRUC3 no longer match.

After the conversion

The only way of solving this problem is to assign the components of the structures to each other individually:

```
data: begin of STRUC1,
        F0(1)  type x,
        F1(10) type c,
        F2(20) type c,
        F3     type i,
        F4 type p,
      end of STRUC1,

      begin of STRUC2,
         C0(1) type c,
         C1(10) type c,
         C2(20) type c,
         C3     type i,
         C4     type f,
      end of STRUC2,

      begin of STRUC3,
         G0(1) type c,
         G1(10) type c,
         G2(20) type c,
      end of STRUC3.

STRUC1-F1 = STRUC2-C1.
STRUC1-F2 = STRUC2-C2.
STRUC1-F3 = STRUC2-C3.

STRUC3    = STRUC2.
```

9.5 Assignment Between Structure and Single Field of Type N

Assignments between a non-character-type structure and a single field of type N are no longer allowed in Unicode programs.

Before the conversion to Unicode

```
data: begin of STRUC,
        NUMBER(20) type n,
        F2 type p,
      end of STRUC,

      NUMBER(20)   type n.

NUMBER = STRUC.                    ← Unicode error
```

After the conversion to Unicode

Since the first component of the structure is to be assigned to the single field, the code segment can be converted easily by replacing the assignment of the whole structure with an assignment of the first structure component.

```
...
NUMBER = STRUC-NUMBER.
```

9.6 Assignment Between Structure and Single Field of Type D

Before the conversion to Unicode

```
data: begin of STRUC,
        YEAR(4)  type n,
        MONTH(2) type n,
        DAY(2)   type n,
        F4 type p,
      end of STRUC,

      DATE       type d.

DATE = STRUC.              ← Unicode error
```

Assignments between a non-character-type structure and a single field of type D are no longer allowed in Unicode programs.

After the conversion to Unicode

An offset-based or length-based access to the character-type initial part of the structure enables you to convert the problematic code segment to a Unicode-enabled code.

```
DATE = STRUC(8).           ← ok!
```

9.7 Assignment Between Structure and Single Field: ASCII Codes

Previously, assignments between structures with components of type X and single fields were also used to calculate characters for an ASCII code, as shown in the following example.

Before the conversion to Unicode

```
data: begin of TAB,
        X(1) type x value '09', "ASCII horizontal tab
      end of TAB,

      C(10)  type c.

C+5(1) = TAB.              ← Unicode error
```

This does not work in Unicode programs. Now, the new predefined constants in the CL_ABAP_CHAR_UTILITIES class provide a simple and efficient solution for all platforms.

After the conversion to Unicode

```
class cl_abap_char_utilities definition load.
c+5(1) =  cl_abap_char_utilities=>horizontal_tab.
```

Note: If the character in use is not predefined as a constant, you can get the appropriate character for a Unicode code point by using the CL_ABAP_CONV_IN_CE=>UCCP or CL_ABAP_CONV_IN_CE=>UCCPI) method. You can take advantage of the fact that the characters with ASCII codes 00 to 7F) are equivalent to the first 127 Unicode codepoints which are U+0000 to U+007F.

9.8 Assignment Between Structure and Single Field: Container Fields (1)

Long fields of type C were often used as *containers*, in which data of different structures was stored. The assignment between structure and single field could be used directly for storing and reading the data. In Unicode programs, these assignments are only allowed if the structure is purely character-type.

Before the conversion to Unicode

```
data: begin of STRUC,
         F1(3)  type x,
         F2(8) type p,
      end of STRUC,

      CONTAINER(1000) type c.

"Store data in container
```

```
CONTAINER = STRUC.                    ← Unicode error

"read data from container
STRUC     = CONTAINER.                ← Unicode error
```

Now, C fields can only be used as containers for storing non-purely character-type structures if the content is only used temporarily within an application server. Therefore the content of the container field cannot be stored in the database, sent by RFC, or written to a file[1]. It is also not possible to access the container field for the selection of structure components by using offset or length[2].

If these prerequisites are met, the writing and reading of data can be converted by using casts to set field symbols of the type X to the container field and the structure, and then implementing the assignment between the field symbols. Due to the assignment, the remaining content of the container is filled with Hex 00 values instead of blank characters. This should not cause any problems if the C field is only used for data storage.

After the conversion to Unicode

```
field-symbols: <X_CONTAINER> type x,
               <X_STRUC>     type x.

assign CONTAINER to <X_CONTAINER> casting.
assign STRUC     to <X_STRUC>     casting.

"Store data in container
<X_CONTAINER> = <X_STRUC>.          ← ok!

"Read data from container
<X_STRUC>     = <X_CONTAINER>.      ← ok!
```

Note: The class CL_ABAP_CONTAINER_UTILITIES offers generic solutions to store structures in containers of type C or STRING. However, when using these classes, the class wrapping and the generic solution can have a negative effect on the performance.

[1] When reading from the database, writing to it, or using RFC, bytes of the character field might be swapped automatically – depending on the byte order (little endian/big endian) of the application server hardware. If non-character-type data is stored in the character field, this automatic swapping causes a truncation of the content.

[2] An access with offset or length is no longer useful, since the offset and length values might be different on non-Unicode and Unicode platforms. In addition, accesses to C fields are always counted in characters. In the structures, however, components might appear that have odd lengths or offsets on Unicode systems.

```
...
class CL_ABAP_CONTAINER_UTILITIES definition load.

call method CL_ABAP_CONTAINER_UTILITIES =>FILL_CONTAINER_C
  exporting  IM_VALUE               = STRUC
  importing  EX_CONTAINER           = CONTAINER
  exceptions ILLEGAL_PARAMETER_TYPE = 1
             others                 = 2.

call method CL_ABAP_CONTAINER_UTILITIES =>READ_CONTAINER_C
  exporting  IM_CONTAINER           = CONTAINER
  importing  EX_VALUE               = STRUC
  exceptions ILLEGAL_PARAMETER_TYPE = 1
             others                 = 2.
```

9.9 Assignment Between Structure and Single Field; Container Fields (2)

If the total length of the structures stored in the C container is always a multiple of the length of one character - this is the case when the structure contains at least one component of the type C, N, D, T, I, or F - you can use the following solution, which is simpler and only needs one field symbol.

Before the conversion to Unicode

```
data: begin of STRUC,
         F1(3)  type x,
         F2(8)  type p,
         F3(10) type c
      end of STRUC,

      CONTAINER(1000) type c.

* Store data in container
CONTAINER = STRUC.                    ← Unicode error

* Read data from container
STRUC     = CONTAINER.                ← Unicode error
```

After the conversion to Unicode

```
field-symbols: <C_STRUC> type c.

assign STRUC to <C_STRUC> casting.

* Store data in container
CONTAINER = <C_STRUC>.                ← ok
* Read data from container
<C_STRUC>     = CONTAINER.            ← ok
```

9.10 Assignment Between Structure and Single Field: Non-Local Container

In the following example, a table is filled in which each line contains the name of a Dictionary structure and the content of such a structure stored in a container field. Elsewhere in the program the table is further processed: The values of the structure components are output by means of a function module that determines the offsets and lengths of the structure components, and then the components are selected using offset-based or length-based accesses to the container.

Before the conversion to Unicode

```
types: begin of T_VALUE_WA,
         TABNAME(30)     type c,
         DUMMY_ALIGN     type f,
         CONTAINER(1000) type c,
       end of T_VALUE_WA,
       T_VALUE_LIST type table of T_VALUE_WA.

data:  L_VALUES   type T_VALUE_LIST,
       L_VALUE_WA type T_VALUE_WA,
       L_D010SINF type D010SINF,
       L_CO2MAP   type CO2MAP.

* Storage of different structures in a table:
L_VALUE_WA-TABNAME = 'D010SINF'.
L_D010SINF-DATALG  =  41.
L_VALUE_WA-CONTAINER =  L_D010SINF.    ← Unicode error
append L_VALUE_WA to L_VALUES.

L_VALUE_WA-TABNAME = 'CO2MAP'.
L_CO2MAP-ID        =  42.
L_VALUE_WA-CONTAINER = L_CO2MAP.       ← Unicode error
append L_VALUE_WA to L_VALUES.


* Processing of table L_VALUES elsewhere in the
* Program: Generic output of the structure components.

data: L_DFIES_TAB type table of DFIES.

field-symbols: <DT>       type DFIES,
               <VALUE_WA> type T_VALUE_WA,
               <f>.

clear L_VALUES_WA.
loop at L_VALUES assigning <VALUE_WA>.

* Determine components with offset and length
  call function 'DDIF_NAMETAB_GET'
    exporting
```

```
      tabname = <VALUE_WA>-TABNAME
    tables
      DFIES_TAB = L_DFIES_TAB.

* Output of content of components
  loop at L_DFIES_TAB assigning <DT>.
    assign <VALUE_WA>-CONTAINER+<DT>-OFFSET(<DT>-INTLEN)

    to <F> type <DT>-INTTYPE.
    write  <F>.
  endloop.
endloop.
```

The offset-based or length-based accesses do not cause a syntax or runtime error, but provide incorrect data in a Unicode system, because the function module returns the offset and length values in bytes whereas these values are counted in characters when accessing the C container.

As of Release 4.6 the solutions of creating data objects dynamically simplify the conversion to Unicode and make the program significantly more user-friendly.
In the following example, an X string is used as container for the content of the structures that is filled by using EXPORT ... TO DATA BUFFER… and read by using IMPORT … FROM DATA BUFFER…. This ensures that the data can also be used on different application servers.

After the conversion to Unicode

```
types: begin of T_VALUE_WA,
         TABNAME(30)     type c,
         XCONTAINER      type xstring,
       end of T_VALUE_WA,

       T_VALUE_LIST type table of T_VALUE_WA.

data:  L_VALUES   type T_VALUE_LIST,
       L_VALUE_WA type T_VALUE_WA,
       L_D010SINF type D010SINF,
       L_CO2MAP   type CO2MAP.

* Storage of different structures in a table:
L_D010SINF-DATALG  =  41.
L_VALUE_WA-TABNAME = 'D010SINF'.
export    VALUE from L_D010SINF
          to data buffer L_VALUE_WA-XCONTAINER.
append L_VALUE_WA to L_VALUES.

L_CO2MAP-ID        =  42.
L_VALUE_WA-TABNAME = 'CO2MAP'.
export    VALUE from L_CO2MAP
          to data buffer L_VALUE_WA-XCONTAINER.
append L_VALUE_WA to L_VALUES.
```

```
* Processing of table L_VALUES elsewhere in the
* Program: generic output of
* Structure components.
data: DREF     type ref to data,
      L_NUMBER type i,
      L_TYPE   type c.

field-symbols: <STRUC>   type any,
               <COMP> type any.

loop at L_VALUES into L_VALUE_WA.

* Create appropriate data object
  create data DREF type (L_VALUE_WA-TABNAME).
  assign DREF->* to <STRUC>.

* Fill data object with content from container
  import  VALUE to <STRUC>
          from data buffer L_VALUE_WA-XCONTAINER.

* Output of structure components
  describe field <STRUC> type L_TYPE components L_NUMBER.
  do L_NUMBER times.
    assign component SY-INDEX
           of structure <STRUC> to <COMP>.
    write <COMP>.
  enddo.

endloop.
```

9.11 Character processing

In the following example, components of type X are used for storing characters. This is no longer possible in Unicode programs.

Before the conversion to Unicode

```
data: begin of L_LINE,
        TEXT1(10)  type c,
        MARK1(1)   type x value '00',
        TEXT2(10)  type c,
        MARK2(1)   type x value '00',
        TEXT3(10)  type c,
        MARK3(1)   type x value '00',
        BLANK(100) type c,
      end of L_LINE,

      HEX0(1) type x value '00',
      CRLF(2) type x value '0D0A'.
```

```
L_LINE-TEXT 1 = 'SYSTEM:   '.
L_LINE-TEXT 2 = 'USER:     '.
L_LINE-TEXT 3 = 'CLIENT:   '.

replace: HEX0 with SY-SYSID into L_LINE, ← Unicode error
         HEX0 with SY-UNAME into L_LINE, ← Unicode error
         HEX0 with SY-MANDT into L_LINE. ← Unicode error

condense L_LINE.                         ← Unicode error
concatenate L_LINE CRLF into L_LINE.     ← Unicode error

* Further processing of L_LINE.
```

After the conversion to Unicode

When using the constants of the CL_ABAP_CHAR_UTILITIES class, the use of X fields can be avoided entirely.

```
class CL_ABAP_CHAR_UTILITIES definition load.

data:
  begin of L_LINE,
    TEXT1(10)  type c,
    MARK1(1)   type c
      value CL_ABAP_CHAR_UTILITIES=>MINCHAR,
    TEXT2(10)  type c,
    MARK2(1)   type c
      value CL_ABAP_CHAR_UTILITIES=>MINCHAR,
    TEXT3(10)  type c,
    MARK3(1)   type c
      value CL_ABAP_CHAR_UTILITIES=>MINCHAR,
    BLANK(100) type c,
   end of L_LINE,

   HEX0(1) type c,
   CRLF(2) type c.

HEX0 = CL_ABAP_CHAR_UTILITIES=>MINCHAR.
CRLF = CL_ABAP_CHAR_UTILITIES=>CR_LF.

L_LINE-TEXT1 = 'SYSTEM:   '.
L_LINE-TEXT2 = 'USER:     '.
L_LINE-TEXT3 = 'CLIENT:   '.

replace: HEX0 with SY-SYSID into L_LINE,
         HEX0 with SY-UNAME into L_LINE,
         HEX0 with SY-MANDT into L_LINE.

condense L_LINE.
concatenate L_LINE CRLF into L_LINE.
```

```
* Further processing of L_LINE.
```

Note: A similar version of the sample program above was found in a *real* application. It is more complex than necessary and not an example of good programming. A better solution is the following:

```
class CL_ABAP_CHAR_UTILITIES definition load.

data: L_LINE(133) type c.

concatenate `SYSTEM: `  SY-SYSID
            ` USER: `   SY-UNAME
            ` CLIENT: ` SY-MANDT
           CL_ABAP_CHAR_UTILITIES=>CR_LF
into L_LINE.
```

9.12 Opening Files

Before the conversion to Unicode

```
data: begin of STRUC,
        F1     type c,
        F2 type p,
      end of STRUC,

      DSN(30) type c value 'TEMPFILE'.

STRUC-F1 = 'X'.
STRUC-F2 =  42.

* write data into file
open dataset DSN in text mode.     ← Unicode error
transfer STRUC to DSN.
close dataset DSN.
Read from file

* read data from file
clear STRUC.
open dataset DSN in text mode.     ← Unicode error
read  dataset DSN into STRUC.
close dataset DSN.
write: / STRUC-F1, STRUC-F2.
```

There are two reasons why the sample program is not executable in Unicode. OPEN DATASET in Unicode programs requires a more detailed file format specification, and only purely character-type structures can be written into text files.
Depending on whether or not the old file format still needs to be read or whether or not it is possible to store the data in a different and new format, different conversion solutions are suitable. Two of them are shown below.

After the Unicode conversion - Case 1: New storage text in UTF-8 format

```
...
data: begin of STRUC2,
        F1     type c,
        F2(20) type c,
      end of STRUC2.

* convert data to text
move-corresponding STRUC to STRUC2.

* write data into file
open dataset DSN in text mode for output encoding utf-8.
transfer STRUC2 to DSN.
close dataset DSN.

* read data from file
clear STRUC.
open  dataset DSN in text mode for input encoding utf-8.
read  dataset DSN into STRUC2.
close dataset DSN.

move-corresponding STRUC2 to STRUC.
write: / STRUC-F1, STRUC-F2.
```

Storing the text in UTF-8 format ensures that the created files are platform-independent.

After the Unicode conversion - Case 2: Old non-Unicode format must be maintained

```
...
* write data into file
open dataset DSN in legacy text mode for output.
transfer STRUC to DSN.
close dataset DSN.

* read from file
clear STRUC.
open  dataset DSN in legacy text mode for input.
read  dataset DSN into STRUC.
close dataset DSN.
write: / STRUC-F1, STRUC-F2.
```

The use of the LEGACY TEXT MODE ensures that the data in old non-Unicode format is stored and read. In this mode, it is also possible to read and write non-character-type structures. However, you have to take into account that loss of data and conversion errors might occur in real Unicode systems if the structure contains characters that cannot be displayed in the non-Unicode codepage.

9.13 Formatting Lists

Later Overwriting Positioning of SY-VLINE

Bad code:

```
WRITE / AT 2 '일이삼'.
WRITE   AT 1  sy-vline.
WRITE   AT 4  sy-vline.
WRITE   AT 7  sy-vline.
WRITE   AT 10 sy-vline.
```

Should be: |일|이|삼|

Is: |일| >| |

Recommended solution:

```
WRITE AT 1 sy-vline.
WRITE AT 2 '일'.
WRITE AT 4 sy-vline.
WRITE AT 5 '이'.
WRITE AT 7 sy-vline.
WRITE AT 8 '삼'.
WRITE AT 10 sy-vline.
```

Output Using RIGHT-JUSTIFIED

```
Bad code:
DATA text(10) TYPE c.
text = '일이삼'.
WRITE text TO text RIGHT-JUSTIFIED.
WRITE text.
```

Should be: 일이삼

Is: 일>

Recommended solution:

```
DATA text(10) type c.
text = '일이삼'.
WRITE text RIGHT-JUSTIFIED.
```

Output of Complete Content
Bad code:

```
DATA text(10) TYPE c.
text = 'OTTOS MOPS'.
WRITE / text.
text = '가갸거겨고교구규그기'.
WRITE / text.
```

Should be: OTTOS MOPS
가갸거겨고교구규그기
Is: OTTOS MOPS
가갸거겨>

Recommended solution:

```
DATA text(10) TYPE c.
text = 'OTTOS MOPS'.
WRITE / (*) text.
text = '가갸거겨고교구규그기'.
WRITE / (*) text.
```

Scrolling:
Bad code:

```
TYPES t_line(100)   TYPE c.
DATA: line          TYPE t_line,
      tab           TYPE table of t_line.
PARAMETERS scrolcol TYPE i DEFAULT 14.

line = '이름1: xxxxxx 이름2: yyyyyy'.
APPEND line TO tab.
line = '이름1: 남명희 이름2: 조홍우'.
APPEND line TO tab.
LOOP AT tab INTO line.
  WRITE / line+scrolcol.
ENDLOOP.
```

Should be: 이름2: yyyyyy
이름2: 조홍우
Is: 2: yyyyyy
조홍우

Recommended solution:

```
TYPES t_line(100)   TYPE c.
DATA: line          TYPE t_line,
      tab           TYPE table of t_line.
PARAMETERS scrolcol TYPE i DEFAULT 14.
```

```
line = '이름1: xxxxxx 이름2: yyyyyy'.
APPEND line TO tab.
line = '이름1: 남명희 이름2: 조홍우'.
APPEND line TO tab.
LOOP AT tab INTO line.
  WRITE / line.
ENDLOOP.
SCROLL LIST TO COLUMN scrolcol.
```

Mixing Output and Buffer Length
The example refers to the database table ZCHNUMBERS in the ABAP Dictionary, whose content is to be descriptions of the numbers 1 to 5 in English and Korean.

Bad code:

```
SELECT * FROM zchnumbers INTO wa ORDER BY num lang.
  WRITE wa-lang  TO line(2).
  WRITE sy-vline TO line+2(1).
  WRITE wa-name  TO line+3(5).
  WRITE sy-vline TO line+8(1).
  WRITE wa-num   TO line+9(3) RIGHT-JUSTIFIED.
  WRITE / line.
ENDSELECT.
```

Should be:

```
KO|하나    | 1
EN|one     | 1
KO|둘      | 2
EN|two     | 2
KO|셋      | 3
EN|three   | 3
KO|넷      | 4
EN|four    | 4
KO|다섯    | 5
EN|five    | 5
```

Is:

```
KO|하나  | 1
EN|one  | 1
KO|둘   | 2
EN|two  | 2
EN|two  | 2
KO|셋   | 3
EN|three| 3
KO|넷   | 4
EN|four | 4
KO|다섯  | 5
```

Recommended solution:

```
DATA: offset_tab TYPE abap_offset_tab.

APPEND 3 TO offset_tab.
APPEND 8 TO offset_tab.
SELECT * FROM zchnumbers INTO wa.
  WRITE wa-lang  TO line(2).
  WRITE sy-vline TO line+2(1).
  WRITE wa-name  TO line+3(5).
  WRITE sy-vline TO line+8(1).
  WRIE  wa-num   TO line+9(3) RIGHT-JUSTIFIED.
```

```
  CALL METHOD cl_abap_list_utilities=>memory_to_display
    EXPORTING memory_data  = line
              offset_tab   = offset_tab
    IMPORTING display_data = disp_line.
  WRITE / disp_line.
ENDSELECT.
```

Misuse of the System Field SY-CUCOL as a Buffer Offset
Bad code:

```
DATA: off type i.

AT LINE-SELECTION.
  off = sy-staco + sy-cucol - 3.
  sy-lisel+off(1) = '-'.
  MODIFY CURRENT LINE.

START-OF-SELECTION.
  WRITE / '한국'
  WRITE at 50(14) '일-이-삼-사-오'.
  SCROLL LIST TO COLUMN 20.
```

If you double-click the gaps between the visible Korean characters, the following is displayed:

Should be: 일-이-삼-사-오

Is: 일 이 -사 - - -

Recommended solution:

```
DATA: mem_off TYPE i,
      f       TYPE string.
AT LINE-SELECTION.
  GET CURSOR FIELD f MEMORY OFFSET mem_off.
  sy-lisel+mem_off(1) = '-'.
  MODIFY CURRENT LINE.

START-OF-SELECTION.
  WRITE / '한국'.
  WRITE at 50(14) '일-이-삼-사-오'.
  SCROLL LIST TO COLUMN 20.
```

10. Glossary

byte type
The ABAP data types X and XSTRING
code
Binary encoding of letters, digits, and special characters
codepage
Set of encoded characters for the environment selected
data cluster
Grouping of several data objects (fields, structures, tables)
Dictionary
ABAP Data Dictionary
endian
The byte order of a number. Numbers are stored in memory with decreasing place value either from left to right (big-endian format) or from right to left (little-endian format)
front end
Presentation server in the SAP System
kernel
The Basis functions of the SAP System (written in C and C++)
NUP
Non-Unicode program: ABAP program for which the Unicode flag has not been set
NUS
Non-Unicode system: SAP System, in which each character is encoded in binary form such that it occupies one byte
remaining length
Pertaining to a field or other character-type structure, the total length minus the offset
surrogate area
Character supplement for characters that cannot be contained in standard Unicode (which can only contain 65,536 characters). In addition to the standard Unicode bit pattern, the system reads two further bytes from this area.
Unicode Fragment View
View that splits structures into similar data areas
UP
Unicode program: ABAP program for which the Unicode flag has been set
US
Unicode system: SAP System, in which each character is encoded in binary form, such that it occupies either two or four bytes
UTF-8
Data format for communication and data exchange
XML
Extensible Markup Language: Language used to display documents on the Internet
character
Letter, digit, or special character
character-type
The ABAP data types C, N, D, T, and STRING

11. Index of Key Concepts