ABAP list layout in Unicode Systems: Development Guide

February 17, 2005

Summary:

1 Overview

Preparing ABAP programs for Unicode means eliminating the implicit or explicit assumption that one character consumes one byte in the application server's memory:

1 Character <> 1 Byte

The Unicode preparation requires a more restrictive Unicode syntax check and then the removal of any errors in ABAP programs. All SAP Programs have been checked; for information on how to check your programs, see the ABAP Conversion documentation.

From the point where users could log on to a Unicode system with a 6.20 SAPGUI, it became evident that during the output of Unicode characters in ABAP lists (and in printouts) another implicit assumption is broken: the assumption, that one display column in ABAP lists is equivalent to a field of type C and length 1:

1 Character <> 1 Display Column

(Actually this assumption no longer held with the introduction of Thai in release 3.0D. But the handling for Thai has been hidden.)

Unicode systems normally continue to use the old-fashioned[1] non-proportional fonts. As a result, the former double byte characters are still twice as wide as European characters. But Asian characters can be stored in a character field of length 1 in a Unicode system:

‘ ’	Number of character units in the memory	Number of display columns
Non-Unicode	2	2
Unicode	1	2

As a consequence, the intended layout of existing ABAP lists often will not be preserved if no measures are taken to keep the list layout compatible.

As an example, consider the following table (ZCHNUMBERS) with the fields:

Table 1 DDIC definition of ZCHNUMBERS

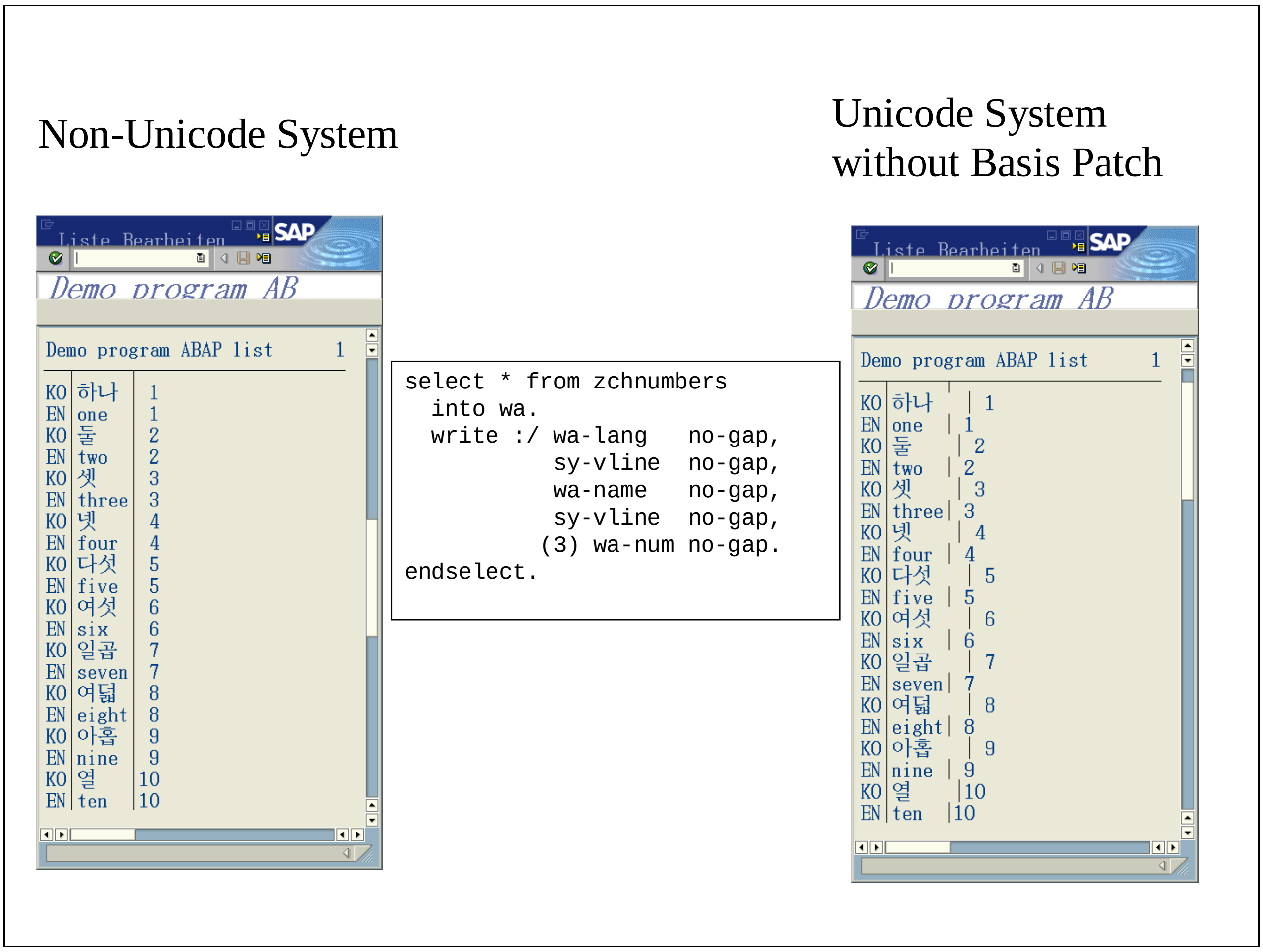
Field	Data type	Length
LANG	LANG	1
NAME	CHAR5	5
NUM	INT4	10

The table content is the numbers from 1 to 10 in Korean and English.

Without improvements, the following write statements result in the displays shown in Figure 1. In the Unicode case additional spaces would show up.

(Note that the LANG field has an conversion exit, which expands the 1 character language field into the 2-character ISO language code).

[1] In fact using ABAP lists itself is old fashioned as they cannot use proportional fonts and lack a couple of other necessary features. Therefore the new ABAP List Viewer has replaced most ABAP lists on the screen. For form printing on the other hand more powerful tools as SAPScript or Smart forms should be used instead of ABAP list printing.



```
Demo program ABAP list      1

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
KO|여섯 | 6
EN|six  | 6
KO|일곱 | 7
EN|seven| 7
KO|여덟 | 8
EN|eight| 8
KO|아홉 | 9
EN|nine | 9
KO|열   |10
EN|ten  |10
```

```
select * from zchnumbers
  into wa.
  write :/ wa-lang   no-gap,
           sy-vline  no-gap,
           wa-name   no-gap,
           sy-vline  no-gap,
        (3) wa-num no-gap.
endselect.
```

```
Demo program ABAP list      1

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
KO|여섯   | 6
EN|six  | 6
KO|일곱   | 7
EN|seven| 7
KO|여덟   | 8
EN|eight| 8
KO|아홉   | 9
EN|nine | 9
KO|열    |10
EN|ten  |10
```

Figure 1 Need for basis patch

The goal of the new ABAP list concept for Unicode is to make the layout of most ABAP lists look the same in Unicode and non-Unicode systems (assuming the content written to the display is the same, of course). Upgrading a non-Unicode system to Unicode does not alter the logical characters in the database, but only their internal representation. This means reports writing ABAP lists after a Unicode upgrade will output the same data to the display, giving the same representation for most ABAP lists within the new ABAP list concept.

1.1 Internal and external representation: Memory vs. Display

A WRITE <field> statement, outputs data in two different steps:

- First, the content of <field> will be put into a character representation (e.g. integers). This character representation is written to the "*list table*", which also stores information about field boundaries and output formats such as color.
  A flat memory copy of all the character representations within one line is available in ABAP programs in the field SY-LISEL.
  We will call the representation in the list table the "*memory representation*".
- Second, everything is output on the screen or printer.
  We will call the representation on the screen or printer the "*display representation*"

In a non-Unicode system the memory and the display representation happen to be the same (see Figure 2). In a Unicode system, however, you have to distinguish between the two different representations. In fact, you have to distinguish between *memory length* and *memory offset* on one hand and *display length* and *display offset, display position* on the other hand.

In the following examples, enclosed number, for example ①, indicate double-width Asian characters.

Example (non-Unicode System):

```
DATA F1(5). F1 = ‘HUGO'.
DATA F2(3). F2 = ‘123'.
DATA F3(6). F3 = '①②③'.
DATA F4(4). F4 = 'ABCD'.
WRITE: AT /1  F1                 NO-GAP
     , AT (9) F2 CENTERD         NO-GAP
     , AT (7) F3 RIGHT JUSTIFIED NO-GAP
     ,        F4.
```

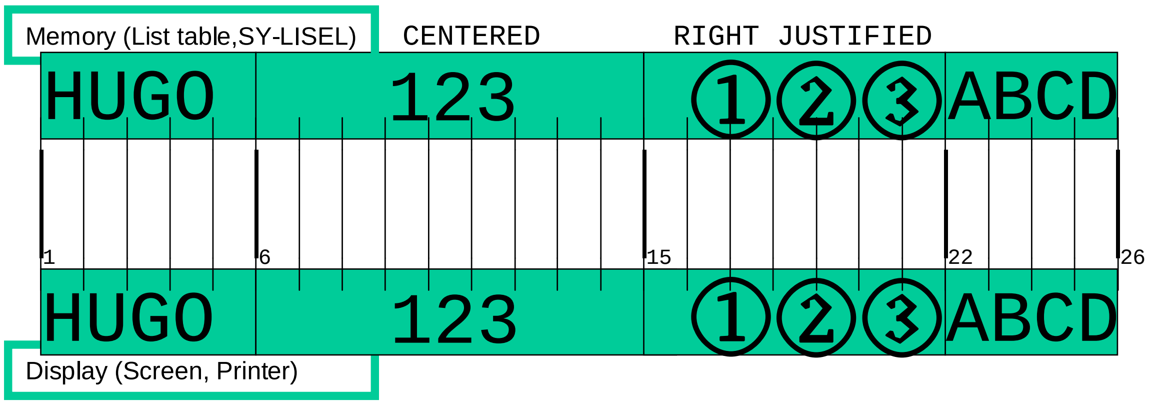


Figure 2 Memory and display representation in a non-Unicode system

2 ABAP list layout in Unicode systems

2.1 General Concept

The main point in the new ABAP list layout in Unicode systems is the implicit determination of display lengths, if no output length has been specified in the WRITE statement. This is explained in section 2.1.1. Sections 2.1.2 and 2.1.3 describe what the end user can do if output of data on the screen had to be truncated.

2.1.1 Display length = Memory length for complete fields

In a Unicode system the memory length and the length necessary to display the whole memory content on the screen is not the same.

Nevertheless, to keep the ABAP list layout and the ABAP list programming techniques as compatible as possible, the new Unicode ABAP list concept ensures that the display length will be the same as the memory length for complete fields.

In order to achieve this, data may be truncated when passed to the display, or data may be padded with spaces when writing it to the list table, depending on the field type.

- The truncation is done during the output of character fields as seen in **Figure 3**.

  → Layout is saved

- The padding during writing in the list table is done for strings as seen in **Figure 4**.

→ Complete content output

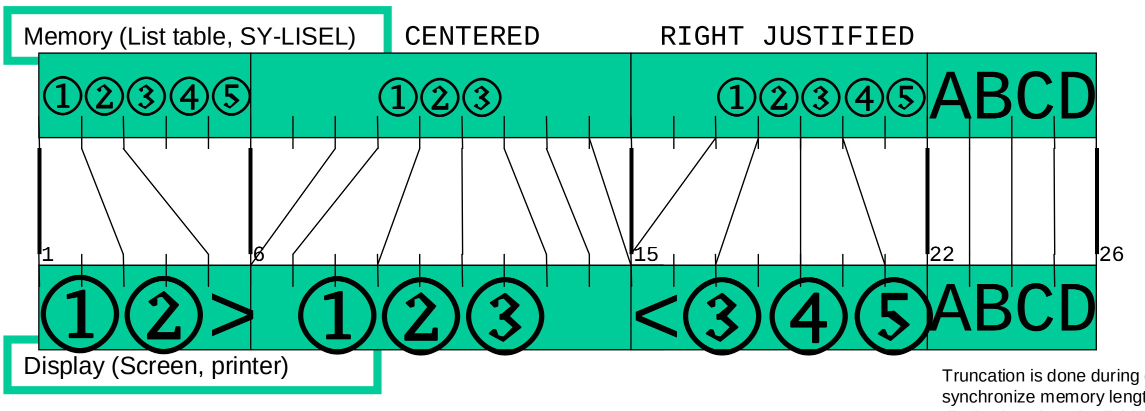
Text elements are treated as character fields of length `mLen`, which is specified in the text element maintenance. Character literals are treated as strings.

The design has two important consequences:

- Display offset and memory offset are ***synchronizing at field boundaries***.
- The list table ***contains all written data*** without any truncation.

Character fields

```
DATA F1(5). F1 = '①②③④⑤'.
DATA F2(3). F2 = '①②③'.
DATA F3(6). F3 = '①②③④⑤ '.   "one space at the end
DATA F4(4). F4 = 'ABCD'.
WRITE: AT /1  F1                 NO-GAP
     , AT (9) F2 CENTERD         NO-GAP
     , AT (7) F3 RIGHT JUSTIFIED NO-GAP
     ,        F4.
```

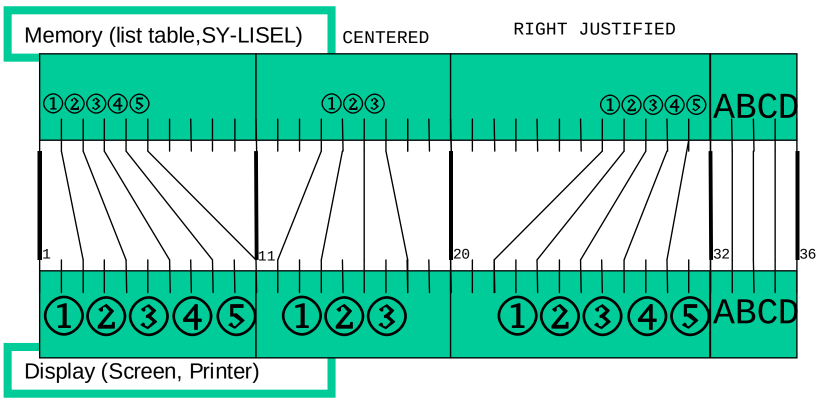


Truncation is done during display to synchronize memory length and display length at the field boundaries.

Figure 3 Memory and display representation in a Unicode system: Character fields

Strings

```
DATA S1 TYPE  STRING. S1 = '①②③④⑤'.
DATA S2 TYPE  STRING. S2 = '①②③'.
DATA S3 TYPE  STRING. S3 = '①②③④⑤'.
DATA S4 TYPE  STRING. S4 = 'ABCD'.
WRITE: AT /1  S1                 NO-GAP
     , (9)    S2 CENTERD         NO-GAP
     , (12)   S3 RIGHT JUSTIFIED NO-GAP
     ,        S4.
```



Padding is done in the list table (for field S1) to synchronize memory length and display length at the field boundaries.

Figure 4 Memory and display representation in a Unicode system: Strings

2.1.2 Truncation indicator

Whenever data has been suppressed due to output length restrictions the “truncation indicator” ‘>' replaces the last visible character.

Example:

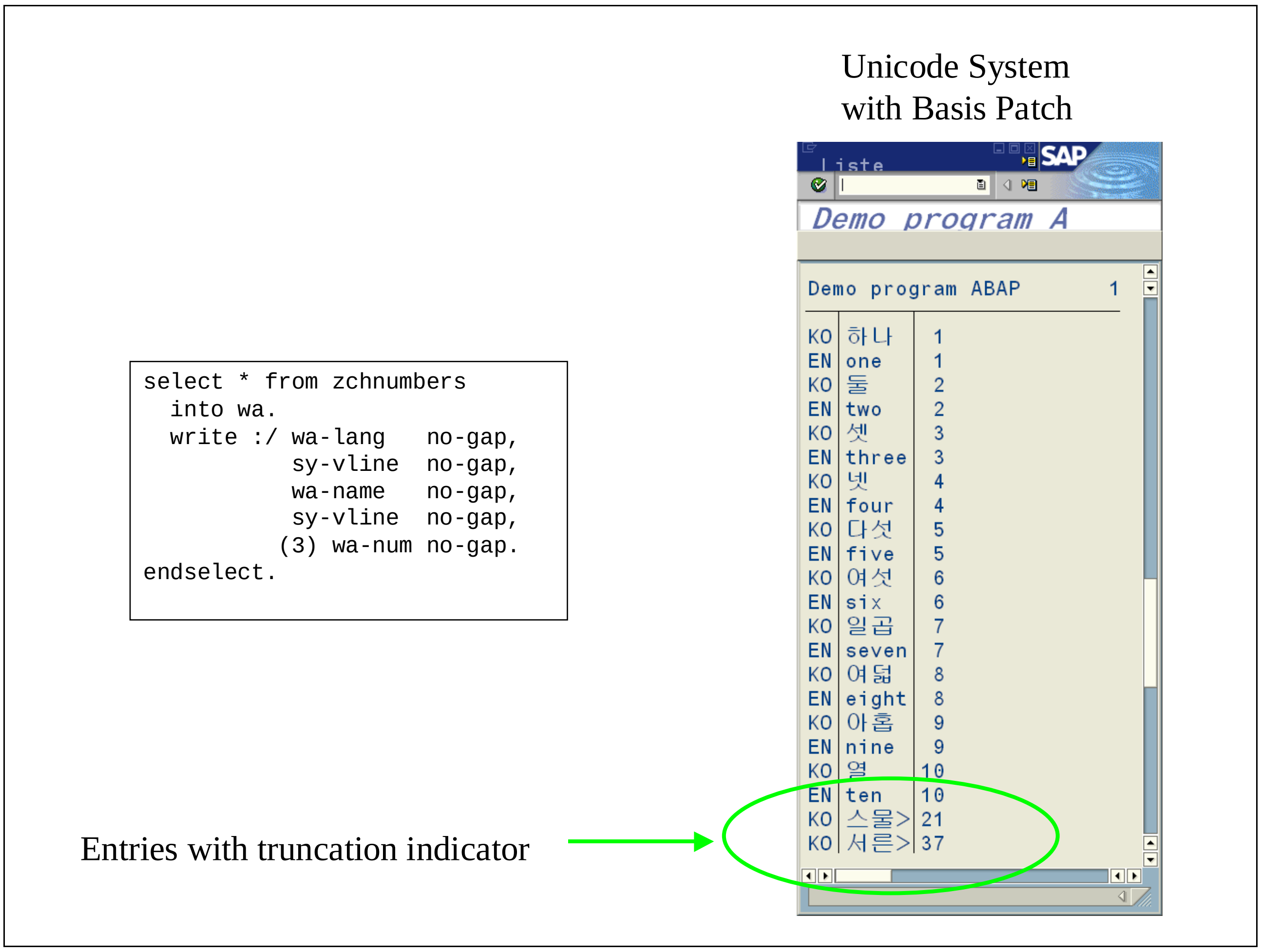
Consider the example table ZCHNUMBERS which contains two more entries in the Unicode system:

Table 2 Additional entries in ZCHNUMBERS

LANG	NAME	NUM
KO	나	21
KO	곱	37

(In the non-Unicode system these entries are not possible because NAME is a character 5 field and in the non-Unicode system a string like ‘ 나’ would occupy 9 characters [4 double-byte Korean characters + one space]).

The resulting screen display is shown in Figure 5.



```
select * from zchnumbers
  into wa.
  write :/ wa-lang   no-gap,
           sy-vline  no-gap,
           wa-name   no-gap,
           sy-vline  no-gap,
          (3) wa-num no-gap.
endselect.
```

```
Demo program ABAP          1

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
KO|여섯   |  6
EN|six   |  6
KO|일곱   |  7
EN|seven |  7
KO|여덟   |  8
EN|eight |  8
KO|아홉   |  9
EN|nine  |  9
KO|열    | 10
EN|ten   | 10
KO|스물>  | 21
KO|서른>  | 37
```

Figure 5 Truncation Indicator

2.1.3 Different list display types

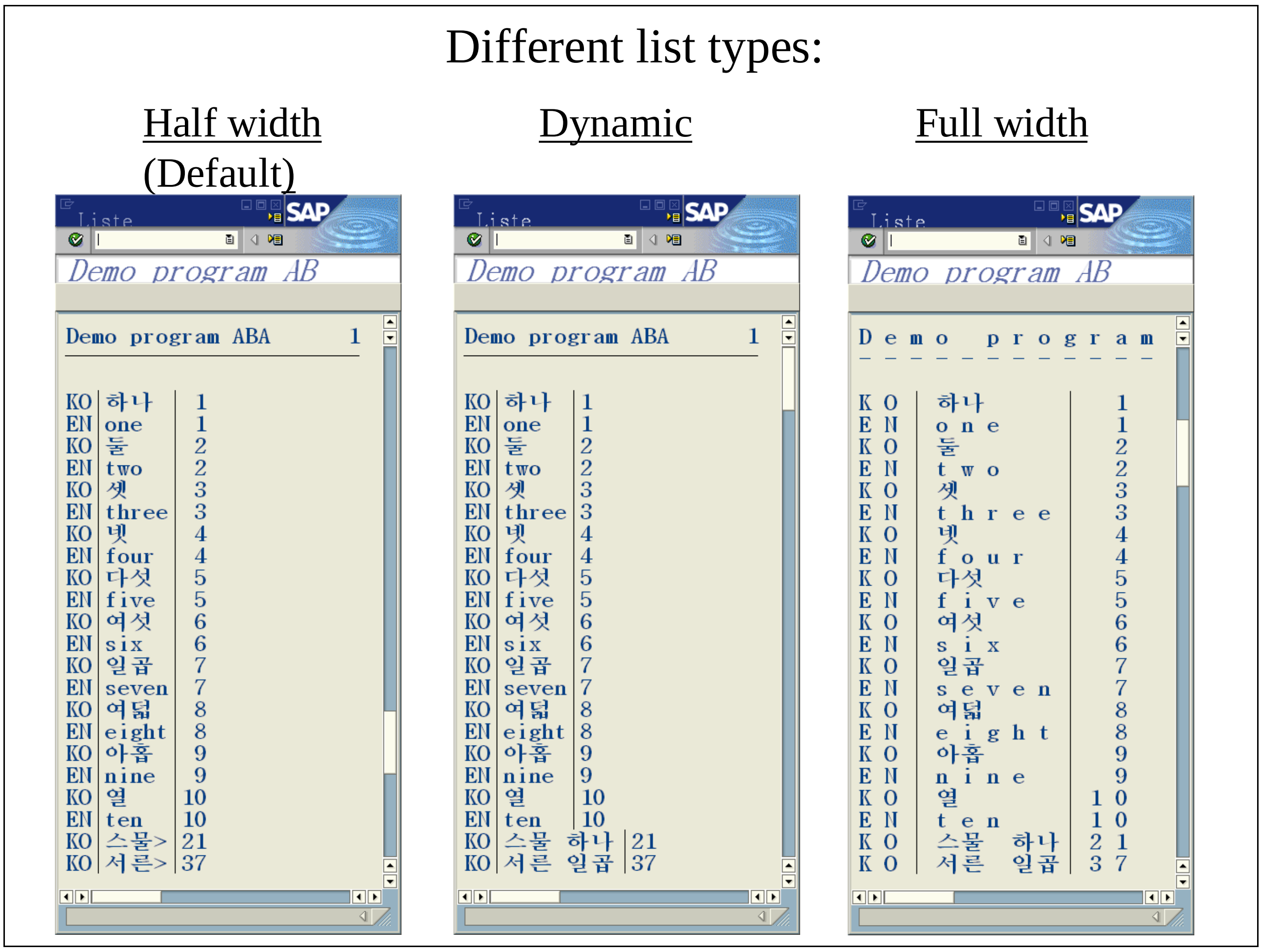
Users have the option to switch to a truncation-free output. Select the menu *System -> List -> Display type* to select a display option.

There will be three different types of “WIDE CHARACTER HANDLING” on the display.

- *Half-width* (= DEFAULT): output with truncation where necessary.
  In this representation the layout is saved, but data truncation may occur.
- *Dynamic*: increase output length where necessary to output all data.
  In this representation the layout may be destroyed, but all data is visible.
- *Full-width*: Output the list table content with the half-width letters spaced to have the same width as double-byte characters (i.e., only letters that occupy one column on the display will be spaced, for all types, including P, F, I or SY-VLINE.)
  In this representation the layout and the data is saved, but the list is very uncomfortable to read.

Examples for the different list types can be seen in Figure 6.

The same types may be used in the ABAP print dialogue. A change in the printing format (e.g. X_65_80 to X_65_200, …) will be suggested to the user in order to get the complete list printed.



```
Demo program ABA         1

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
KO|여섯 |  6
EN|six  |  6
KO|일곱 |  7
EN|seven|  7
KO|여덟 |  8
EN|eight|  8
KO|아홉 |  9
EN|nine |  9
KO|열   | 10
EN|ten  | 10
KO|스물>|21
KO|서른>|37
```

```
Demo program ABA         1

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
KO|여섯 |6
EN|six  |6
KO|일곱 |7
EN|seven|7
KO|여덟 |8
EN|eight|8
KO|아홉 |9
EN|nine |9
KO|열   |10
EN|ten  |10
KO|스물 하나|21
KO|서른 일곱|37
```

```
D e m o   p r o g r a m
- - - - - - - - - - - -

K O | 하나      |   1
E N | o n e     |   1
K O | 둘        |   2
E N | t w o     |   2
K O | 셋        |   3
E N | t h r e e |   3
K O | 넷        |   4
E N | f o u r   |   4
K O | 다섯      |   5
E N | f i v e   |   5
K O | 여섯      |   6
E N | s i x     |   6
K O | 일곱      |   7
E N | s e v e n |   7
K O | 여덟      |   8
E N | e i g h t |   8
K O | 아홉      |   9
E N | n i n e   |   9
K O | 열        | 1 0
E N | t e n     | 1 0
K O | 스물 하나 | 2 1
K O | 서른 일곱 | 3 7
```

Figure 6 Different list types

2.2 Affected ABAP statements and system fields

There are different places where you have to consider the difference between display offset and length and memory offset and length. In the following we give a list of the most important cases. For details, please refer to the online documentation.

2.2.1 SY-* fields

SY-LISEL	is the line as it is stored in the list table.
SY-COLNO	gives the current position in the list table. At the field boundaries this is the same as the display position.
SY-CUCOL	contains the number of the display column on which the cursor was positioned relative to the left scroll boundary. SY-CUCOL does not refer to the position in SY-LISEL.
SY-STACO	is the number of the left-most visible display column. Without scrolling, SY-STACO is 1. In non-Unicode systems, SY-STACO + SY-CUCOL - 3 could be used to get the offset in SY-LISEL. For Unicode systems, this does not work. Please use the statements GET CURSOR FIELD/LINE MEMORY OFFSET instead.
SY-LINSZ	contains the width of the current list table. This is the same as the current display width.

2.2.2 ABAP statements

2.2.2.1 WRITE <charfield>.

The content of <charfield> is written to the list table at memory offset SY-COLNO. The display is restricted to a number of columns equal to the technical size of <charfield> and starts at display offset SY-COLNO.

2.2.2.2 WRITE AT <col> <charfield>.

The content of <charfield> is written to the list table at memory offset <col>. The display is restricted to a number of columns equal to the technical size of <charfield> and starts at display offset <col>.

2.2.2.3 WRITE AT <col>(<len>) <charfield>.

The content of <charfield> is written to the list table at memory offset <col> in the length <len>. The display length is equal to <len> and starts at display offset <col>.

2.2.2.4 WRITE <string>.

The content of <string> is written to the list table at memory offset SY-COLNO. To ensure the complete display of <string> the display length of the whole string is calculated and in the list table trailing spaces are added to the content of <string>. The number of added spaces is determined by the difference of the calculated display length and the memory string length. The length of the resulting list table field is the same as the display length. The display starts at display offset SY-COLNO.

2.2.2.5 WRITE AT <col> <string>.

The content of <string> is written to the list table at memory offset <col>. To ensure the complete display of <string> the display length of the whole string is calculated and in the list table trailing spaces are added to the content of <string>. The number of added spaces is determined by the difference of the calculated display length and the memory string length. The length of the resulting list table field is the same as the display length. The display starts at display offset <col>.

2.2.2.6 WRITE AT <col>(<len>)<string>.

The content of <string> is written to the list table at memory offset <col> in the length <len>. The display length is equal to <len> and starts at display offset <col>.

2.2.2.7 WRITE f USING EDIT MASK mask

Writing fields with edit mask will output the result with dynamic length, i.e. ‘_’ will be replaced by one character, which can have display length 1 or 2.

2.2.2.8 WRITE f INPUT.

The input field has the same display width as it would have without the addition "INPUT". Interactive input is limited by the display width of the field (i.e., there is no scrolling). For STRINGs, the width of the input fields depends on the contents, and interactive input is limited by this width.

Entered data is read form the visible input field as it is, i.e. those parts of f that have not been sent to the screen due to output length restriction are lost.

2.2.2.9 REPORT … LINE-SIZE <col>.

<col> specifies the memory length of the list table. This is the same as the reserved display length.

2.2.2.10 READ/MODIFY LINE

READ/MODIFY LINE FIELD … is working on complete fields and therefore does not change its behavior.

If you use “MODIFY CURRENT LINE” or “MODIFY CURRENT LINE FIELD VALUE FROM wa.” please use one of the new methods (2.3.2.3) in order to handle implicit field boundaries while changing the content of SY-LISEL or wa.

2.2.2.11 GET/SET CURSOR

For all existing variants of GET/SET CURSOR, offsets will refer to the display. There will be new variants with “MEMORY OFFSET” instead of “OFFSET”, which refer to the memory offset rather than the display offset. The addition “DISPLAY OFFSET” will be a synonym for “OFFSET”.

For all existing variants of GET/SET CURSOR, lengths always refer to complete fields and therefore give the display length and the memory length at the same time.

In the statement “SET CURSOR <col> <line>” <col> is the display positions relative to the upper left corner. If a memory position is intended, the programmer should use SET CURSOR FIELD (or SET CURSOR LINE on lists).

2.2.2.12 SCROLL LIST …

In all scroll statements length and position values are counted in display columns. (SCROLL LIST LEFT/RIGHT BY n PLACES; SCROLL LIST TO COLUMN col; SET LEFT SCROLL-BOUNDARY COLUMN col.)

2.2.2.13 DESCRIBE FIELD f OUTPUT-LENGTH len

Same as CL_ABAP_LIST_UTILITIES=>DEFINED_OUTPUT_LENGTH (see 2.3.2.1).

2.2.2.14 PRINT-CONTROL … POSITION col

The control options will be active starting at memory offset col in the list buffer of the specified line.

2.2.2.15 STRLEN

The lengths determined with STRLEN are memory lengths.

2.3 New language elements and utility classes

The following new language elements and the methods of the utility class CL_ABAP_LIST_UTILITIES are needed for Unicode compliant ABAP list programming. The interface of the ABAP class is available with 6.20 basis service package SP8. The new language elements are available only with the appropriate kernel patch level (>=364).

2.3.1 New Language elements

2.3.1.1 Dynamic and maximal length in the WRITE statement:

To reduce the number of keystrokes, short forms for outputting data in dynamic or maximum length are provided:

- ‘WRITE (*) field’ for strings and c fields as a short form for

  ```
  data len type i.
  len = CL_ABAP_LIST_UTILITIES=>DYNAMIC_OUTPUT_LENGTH( field ).
  write (len) field.
  ```

  For P, F and I fields, the output is so long as it is necessary for the full editing of the value. Additions like NO-SIGN, ROUND, DECIMALS, USING EDIT MASK and conversion exits affect the output. Especially for a character field

  ```
  Write (*) field using edit mask mask.
  ```

  the output is not cut.

- ‘WRITE (**) field’ for strings and c fields as a short form for

  ```
  data len type i.
  len = CL_ABAP_LIST_UTILITIES=>MAXIMUM_OUTPUT_LENGTH( field ).
  write (len) field.
  ```

  for P, F and I fields, the output is so long as it is necessary for the full editing of the largest value. Additions like NO-SIGN, ROUND, DECIMALS, USING EDIT MASK and conversion exits affect the output.

2.3.1.2 SET/GET CURSOR: MEMORY OFFSET variants

For all existing variants of GET/SET CURSOR, offsets refer to the display. There will be new variants with “MEMORY OFFSET” instead of “OFFSET”, which refer to the memory offset rather than the display offset. The addition “DISPLAY OFFSET” will be a synonym for “OFFSET”. The MEMORY OFFSETS variants are only available on lists.

2.3.2 Utility Class CL_ABAP_LIST_UTILITIES

This utility class will contain different static methods in order to help programmers calculating display lengths, converting between memory values to display values and declaring the existence of field boundaries within container fields.

2.3.2.1 Calculating display lengths

- **DEFINED_OUTPUT_LENGTH**
  The output length as specified in the data dictionary (or the length as specified in a DATA statement) will be returned. Except for STRINGs the result does not depend on the content of <field>, but on its technical type (the length is part of the type). For type STRING this method will give the same result as DYNAMIC_OUTPUT_LENGTH. For all types the result is the same as in DESCRIBE FIELD <field> OUTPUT-LENGTH <len>.
- **DYNAMIC_OUTPUT_LENGTH**
  The number of display columns necessary to output the whole content of <field> will be returned. This number depends on the content of <field>. For type C the number of display columns necessary to output the content of <field> without trailing spaces will be returned. For type STRING, trailing spaces are included.
- **MAXIMUM_OUTPUT_LENGTH**
  For type STRING this method will return the same result as DYNAMIC_OUTPUT_LENGTH. For type C this methods returns twice the memory length. For the types N, D and T, this method returns the same as DEFINED_OUTPUT_LENGTH. For structures this method returns the sum of the maximum output lengths of the individual fields.

2.3.2.2 Conversions display length ←→ memory length inside fields

- **DISPLAY_OFFSET**
  This method calculates the display length for the beginning part of a field specified by the parameter memory_offset. This is the same as DYNAMIC_OUTPUT_LENGTH(field(memory_offset)).

  MEMORY_OFFSET
  This is the inverse of method DISPLAY_OFFSET.

2.3.2.3 Handling of implicit field boundaries

One of the major problems of ABAP lists in Unicode systems is the occurrence of fields that have a implicit sub-structure that is invisible to the ABAP list processor when writing the field as a whole.

Fields with implicit substructure come in two different flavors: in *memory-layout* and *display-layout* (see Figure 7).

a) A field in memory-layout is well suited for manipulations with memory offsets and lengths but must not be output to the display as it is, because the implicit layout is lost at the display.

b) The display-oriented layout is well suited for direct display output, but manipulations with memory offsets and lengths will destroy the implicit layout or even overwrite content.

Typically you have fields in memory-layout form when you build up a complete line with MOVE or WRITE TO statements before writing it to the screen.
Fields in display-layout form are found in text elements that represent complete lines with implicit sub-structuring or come from database entries that have been formed with a display-oriented editor.

In order to convert a field with implicit sub-structuring from the memory-layout into the display-layout (and vice versa) three different types of methods are provided. Each determines filed boundaries differently. The methods are:

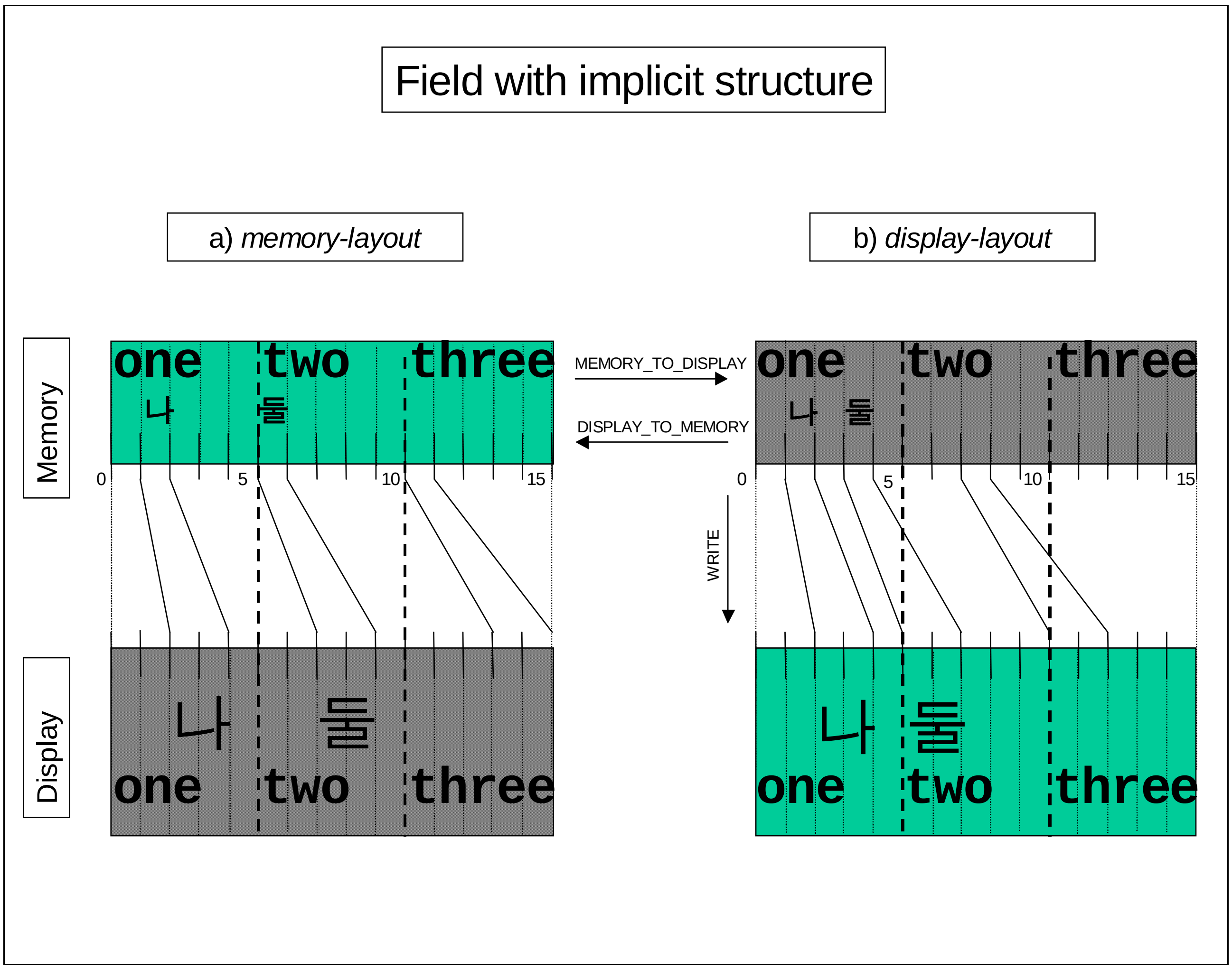


Figure 7 Field with implicit structure

- **MEMORY_TO_DISPLAY / DISPLAY_TO_MEMORY**
  The implicit field boundaries are explicitly supplied via an offset table.
- **STRUCTURE_TO_DISPLAY / DISPLAY_TO_STRUCTURE**
  The implicit field boundaries are supplied indirectly by supplying a structure that has the same field boundaries as the underlying implicit field boundaries in the container field.
- **FRAME_SEPARATED_TO_DISPLAY / DISPLAY_TO_FRAME_SEPARATED**
  The implicit field boundaries are indirectly determined by parsing the field for occurrences of the frame characters SY-VLINE or SY-ULINE.

In addition to the methods above, a method is provided to do replacements in fields in display-layout, if you have display offsets and display lengths for the replacement position:

- **REPLACE_INTO_DISPLAY_LAYOUT**
  This method will do the same as DISPLAY_TO_MEMORY → WRITE … TO +off(len) → MEMORY_TO_DISPLAY

2.3.2.4 Cursor positioning

If the new statement variants for GET/SET CURSOR (2.3.1.2) are not available because the kernel patch level is not sufficient (<364), you may also use the corresponding methods SET/GET_CURSOR_FIELD and

SET/GET_CURSOR_LINE that will do cursor positioning based on memory offsets rather than display offsets.

3 Unicode compliant ABAP lists programming

While writing single fields with output length specification and operations on complete fields are working in the Unicode and in the non-Unicode environment, there are some programming styles previously used that have some problems in the Unicode case.

In order to write ABAP lists that are working both in the Unicode and the non-Unicode system you should obey the following rules:

a) Don't mix up display length and memory length

b) Don't smudge field boundaries

c) Don't overwrite parts of fields

d) Don't do self-programmed right-justified or centered.

e) Don't do self-programmed scrolling (memory based).

f) Don't forget to specify sufficient output length, if all data always needs to be visible

Examples:

1) <u>"SY-VLINE overwriting":</u> (breaks rule b) and c))

Bad coding:

```
write :/ at 2 '       '.
write at 1 sy-vline.
write at 4 sy-vline.
write at 7 sy-vline.
write at 10 sy-vline.
```

intended : `|  |  |  |`

is : `|  | >| |`

Solution:

```
write at 1 sy-vline.
write at 2 '  '.
write at 4 sy-vline.
write at 5 '  '.
write at 7 sy-vline.
write at 8 '  '.
write at 10 sy-vline.
```

2) <u>Right-justified:</u> (breaks rule d))

Bad coding:

```
data: text(10) type c.
text = '     '.
write text to text right-justified.
write text.
```

intended :

is : `        >`

Solution:

```
data: text(10) type c.
```

```
text = ‘     ’.
write text right-justified.
```

3) <u>Display always complete content:</u> (breaks rule f))

Bad coding:

```
data: text(10) type c.
text = ‘OTTOS MOPS‘.
write:/  text.
text = ‘가갸거겨고교구규그기‘.
write:/ text.
```

intended :

```
OTTOS MOPS
가갸거겨고교구규그기
```

is :

```
OTTOS MOPS
가갸거겨>
```

Solution:

```
data: text(10) type c.
text = ‘OTTOS MOPS‘.
write:/ (*)  text.
text = ‘가갸거겨고교구규그기‘.
write:/ (*) text.
```

4) <u>Scrolling:</u> (breaks rule e))

Bad coding:

```
types: t_line(100) type c.
data: line type t_line,
      tab  type table of t_line.
parameters: scrolcol type I default 14.

line = '    1: xxxxxx     2: yyyyyy'. append line to tab.
line = '    1: 남         2:      '. append line to tab.
loop at tab into line.
  write:/ line+scrolcol.
endloop.
```

intended :

```
     2: yyyyyy
     2:
```

is :

```
2: yyyyyy

```

Solution:

```
...
loop at tab into line.
  write:/ line.
endloop.
scroll list to column scrolcol.
```

5) <u>Container, fields in memory-layout or display-layout:</u> (breaks rules a) and b))
Example table zchnumbers as in Figure 1.

Bad coding:

```
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.
```

intended :

```
KO|  나 | 1
EN|one  | 1
KO|둘   | 2
EN|two  | 2
KO|     | 3
EN|three| 3
KO|넷   | 4
EN|four | 4
KO|다   | 5
EN|five | 5
```

is :

```
KO|  나  | 1
EN|one  | 1
KO|둘    | 2
EN|two  | 2
KO|      | 3
EN|three| 3
KO|넷    | 4
EN|four | 4
KO|다     | 5
EN|five | 5
```

Writing single fields is best, but if you have to use the already filled line you can do this:

Solution:

```
data:    offset_tab type abap_offset_tab,
         disp_line like line.

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).
  write 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.
```

6) <u>SY-CUCOL with later usage as memory offset:</u> (breaks rule a))

Bad coding:

```
data: off type i.

AT LINE-SELECTION.
off = SY-STACO + SY-CUCOL – 3.
```

```
SY-LISEL+off(1) = ‘-’.
modify current line.

START-OF-SELECTION.
write:/ ‘  국’
write: at 50(14) ‘            ’.
scroll list to column 20.
```

Placing the cursor on the gaps between the visible Korean characters and double clicking will give:

```
intended   :                                 -  -  -  -
is         :                                   -  - - -
```

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.
...
```

4 Affected Releases

- **BASIS**
  The implementation will be done in the basis release 6.30 and ported to basis release 6.20 SP8 (support package 8). To get all necessary list table modifications the appropriate kernel patch level (>=364) is needed.
- **APPLICATION**
  In order to make the new list functionality available in the application, all application support package systems running on basis release 6.20 need to be upgraded to the appropriate basis support package level SAPKB62008. After reaching the appropriate basis hot package level several corrections have to be done in the application coding (e.g. HR list generator). In order to see the effects of the changes in the ABAP programs the Unicode test system needs to run at least kernel patch level 364.

5 Change history

February 17 2005	Add keyword STRLEN
October 9 2002	Add more details to example 6
September 24 2002	Add appropriate Kernel Patch Level
August 21 2002	PRINT-CONTROL … POSITION col
August 9 2002	Method variants of SET/GET CURSOR, patch levels
August 1 2002	Adjust syntax of some examples
July 31 2002	Minor reformulation in the overview
July 29 2002	Remove Download section
July 23 2002	Examples for most frequent errors
July 10 2002	First version