# HTMLBusiness Language Reference

**Release 4.6C**

## Copyright

## Icons

| Icon | Meaning |
|---|---|
| ⚠ | Caution |
| 💬 | Example |
| ➡ | Note |
| 🧭 | Recommendation |
| SYN | Syntax |
| 💡 | Tip |

# Contents

# HTML$^{Business}$ Language Reference

## Purpose

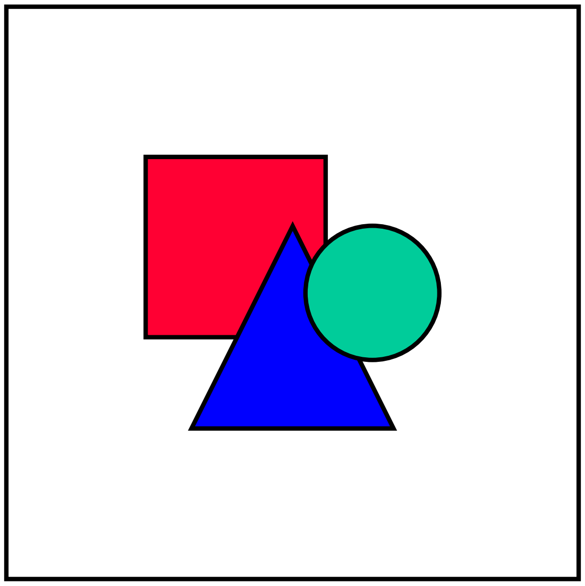
HTML$^{Business}$ is an SAP-specific macro language used by the Internet Transaction Server (ITS) to merge data from transaction screens dynamically into the HTML templates of Internet Application Components (IACs).

When a user starts an IAC from a Web browser, this triggers an ITS service, which corresponds to an R/3 transaction. For each R/3 transaction screen, there an HTML template, which contains standard HTML code and HTMLBusiness statements. At runtime, the R/3 screen data is passed to the ITS. Then, the HTML$^{Business}$ interpreter evaluates the HTML$^{Business}$ statements and the screen data is merged into the HTML templates, which are them displayed in the user‘s Web browser.

## Features

To integrate HTML$^{Business}$ with standard HTML, the ITS employs tags such as `<server>` `</server>` or the equivalent back quotes.

These tags enclose HTML$^{Business}$ expressions and thus avoid the need to implement multiple tags or to mark methods in HTML pages. It also means you can use any authoring tool when writing HTML code to design Web pages.



This document assumes that you have a basic knowledge and understanding of standard HTML, R/3 screen processing, and ITS architecture.

# HTML$^{Business}$ General Rules

The following topics describe general syntax and how to reference variables:

## General Syntax

## Referencing Variables

# Embedding HTML$^{Business}$ in Standard HTML

To embed HTML$^{Business}$ expressions in standard HTML code, you use the **`<server>`** … **`</server>`** tag or the equivalent back quotes.

The following code includes the `vbcom-kunde` screen field in the HTML page:

```
<h1> Order Status </h1>
<p> Customer Number: <server> VBCOM-KUNDE </server>
...
```

## Embedding HTML$^{Business}$ Tags in Standard HTML Tags

Since HTML does not allow tags within tags, you cannot embed HTML$^{Business}$ tags in standard HTML tags.

The following code is not possible:

```
<a href="<server> screenURL </server>"> Link </a>
```

To handle this, HTML$^{Business}$ provides the back quote (`) as an additional way of marking statements.

You can use back quotes in the same way as the **`<server>`** … **`</server>`** tag, and also inside HTML tags.

The hyperlink in the previous example could be written as:

```
<a href="`screenURL`"> Link </a>
```

## Leaving Back Quotes Uninterpreted

If you want to leave back quotes in your HTML page uninterpreted, insert the equivalent code "`&#96`".

In the code

```
<p> This is an HTML Business expression: &#96VBCOM-KUNDE&#96
</p>
```

the `vbcom-kunde` field is left uninterpreted, so the browser output is:

```
The following is an HTML Business expression: `VBCOM-KUNDE`
```

## Using Comments in HTML$^{Business}$

You can use standard HTML comments to comment on HTML$^{Business}$ expressions:

- HTML comments start with `<!--` and end with `-->`
- HTML$^{Business}$ expressions that occur within HTML comments appear as uninterpreted output on the HTML page:

  For example, the line

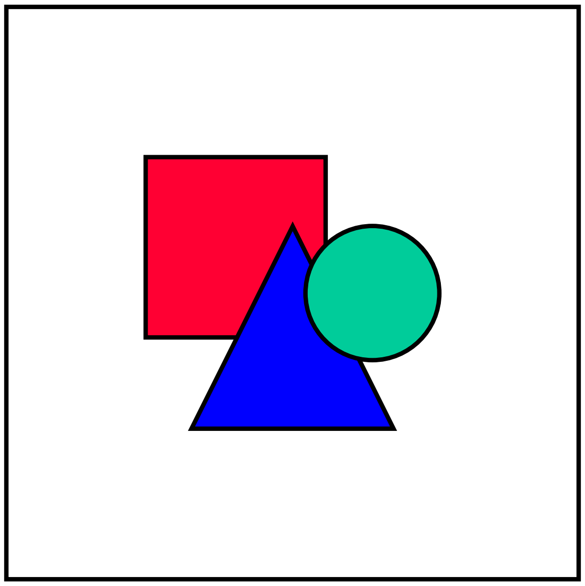
  ```
  <!-- `VBCOM-KUNDE` -->
  ```

  is ignored.

- You can also add comments to parts of HTML$^{Business}$ expressions:

  The following line contains a valid HTML$^{Business}$ expression:

  ```
  `repeat with j from 1 to <!--stepLoop.dim --> 10`
  ```



HTML comments always end with `-->`.

The character > alone is insufficient to indicate the end of a comment.

# Embedding Multiple HTML$^{Business}$ Statements

When using multiple HTML$^{Business}$ statements, you can:

- Enclose them in separate **<server>...</server>** blocks or sets of back quotes.
- Enclose them in a single server command or between a single pair of back quotes.

  In this case (just like in C or JavaScript), you separate the HTML$^{Business}$ statements with semi-colons, as in the following statement:

  ```
  <p> `for (j=0; j <= array.dim; j++)
      j; write("="); array[j]; write (", ")
      end`
  </p>
  ```

## Delimiters Between Statements in Back Quotes

In ITS versions earlier than 2.2, you had to insert at least one separator between multiple HTML$^{Business}$ statements enclosed in back quotes:

```
`if (1)` `write("Hello world!"` `end` (be aware of the blanks!)
```

This also applied to statements enclosed between the **</server>** and **<server>** tags.)

ITS 2.2 does not require a separator between each statement, so you can now write:

```
`if (1)``write("Hello world")``end`.
```

## Delimiters Between Successive HTML$^{Business}$ Statements

In ITS versions earlier than 2.2, you had to insert a semicolon or at least a single separator between successive HTML$^{Business}$ statements.

ITS 2.2 does requires neither semi-colons nor other separators. Therefore, you do not have to include a semicolon:

- Before the keywords **end**, **else**, **elsif** and **elseif**
- After statements that introduce a block.

  Examples are **for (…)**, **if (…)**, **elsif (…)**, **elseif (…)** and **repeat (    )**

# Referencing R/3 Screen Fields

The main task of HTML$^{Business}$ is to tell the Internet Transaction Server (ITS) how to transfer data between R/3 screens and the HTML templates.

At runtime, there are two types of R/3 field known to the ITS:

- Fields passed from an R/3 screen

  Only screen fields are visible to the ITS. Fields known to the module pool but not present in a screen are not visible.

- Fields passed via the RFC channel

  These are fields transferred by the ABAP program using the ITS macros FIELD_SET [Ext.] or FIELD_TRANSPORT [Ext.].

## Referencing Fields

Field replacement is the simplest form of HTML$^{Business}$ statement. With the exception of reserved keywords, every HTML$^{Business}$ expression is treated as the name of an R/3 field.

To display R/3 screen field data in your HTML page, simply enclose the R/3 field name in **`<server>...</server>`** tags or back quotes.

There is normally a 1:1 relationship between screen fields and HTML fields. This applies both when reading from and writing to the screen. Large text areas are an exception to this rule.

- If the ITS finds a match for a field name, it copies the contents of the R/3 field to the HTML page.
- If the ITS finds no match for a field name, it replaces the variable with an empty string.

You can also use field contents in simple expressions.

## Field Identifier Syntax

The following table shows the syntax for referencing R/3 screen fields in HTML$^{Business}$. The attributes provide additional information about the field.

| Name | Syntax |
|---|---|
| Field | `{ ^ } identifier [ [ expression ] ] [. attribute ]` |
| Attribute | `label \|` |
| | `visSize \|` |
| | `maxSize \|` |
| | `dim \|` |
| | `disabled \|` |
| | `name \|` |
| | `value` |

**Referencing R/3 Screen Fields**

In HTML$^{Business}$, field identifiers should conform to the same conventions as in other programming languages such as C or JavaScript.

The conventions are summarized in the following table:

| Field Identifier Component | Possible Characters | Values |
|---|---|---|
| First character | A letter | `a…z, A…Z` |
| | An underscore | `_` |
| | A tilde | `~` |
| Subequent characters | Letters | `a…z, A…Z` |
| | An underscore | `_` |
| | A tilde | `~` |
| | A digit | 0…9 |
| | A hyphen | - |

The following are valid field identifiers:

```
nCustomers
_foo_bar
~frame
date1
ordered_items
vbcom-kunde
```

In the following cases, you must enclose the entire field identifier in single quotation marks:

- If you use other characters.
- If you use the same name as a keyword.

```
'*vbcom-kunde'
'$@#identifier()***'
'from'
```

Identifiers such as **`vbcom-kunde`** do not need single quotes, because the hyphen is legal. Simply write `` `vbcom-kunde` `` instead of `` `'vbcom-kunde'` ``.

However, if you want to use arithmetic expressions that contain minus signs, there must be at least one space before and after the minus sign:

```
`vbcom - kunde`
```

## Limiting the Number of Characters Transferred

You can limit the number of characters per field that are transferred from the HTML page to the screen. To do this, enter a colon after the field name and then specify the number of characters desired. You must specify the length between the field name and the index.

If you limit the number of characters in this way, extra characters are lost. This does not apply when the field is an array and you have not specified an explicit index with `[]`. In this case, characters beyond your specified limit are included in a new line of the step loop. This is necessary when processing large text areas.

## Syntax Conventions

The following syntax conventions are observed in tables:

- Optional derivations are enclosed in angle brackets ( `[]` ).
- Curly brackets ( `{}` ) indicate zero or repetitions of the expression within the brackets.
- A vertical line ( `|` ) links alternative derivatives (“OR”).
- Parentheses ( `()` ) logically combine components in cases where uniqueness cannot be guaranteed.

## Referencing R/3 Arrays

In addition to simple fields, the Internet Transaction Server (ITS) also processes arrays.

On the R/3 side, arrays are either constructed as step loops in the screen or sent from the screen to the ITS via RFC. You can access the individual elements in an array by specifying an index in angle brackets after the field name. This assignment is valid for step loop fields in either direction.

If the angle brackets following the field name do not contain a value, the data transferred from the HTML document to the R/3 System is appended to the existing fields. The array cannot be longer than the associated step loop on the screen. If it is, a runtime error results.

## Using Pointers to R/3 Fields

To define identifiers as pointers to R/3 fields, you can use the caret character `^`.

If, for example, **`Field_Name`** has the value **`vbcom-kunde`**, the following statement reports on the value in the field **`vbcom-kunde`**.

```
<p> The value of the field `Field_Name` is `^Field_Name` </p>
```

This statement generates the following in the HTML page:

```
<p> The value of the field VBCOM-KUNDE is 3100 </p>
```

Multiple indirections such as

```
^^^scarcely_practical[j]
```

are also possible

# Getting R/3 Field Attributes

HTML$^{Business}$ provides several attributes for determining the attributes of screen fields in R/3.

The currently available attributes are listed in the following table:

**Screen Field Attributes**

| Attribute | Determines |
|---|---|
| .dim [Page 18] | Dimension of an array or muptiple value field. |
| .disabled [Page 19] | Whether a field is ready for input or read-only. |
| .exists [Page 20] | Whether or not a field exists on the screen. |
| .label [Page 21] | Text description of an input field. |
| .maxSize [Page 22] | Maximum input length of a field. |
| .name [Page 23] | Name and index of a field. |
| .type [Page 24] | Type of a field. |
| .value [Page 25] | Current value of a field. |
| .visSize [Page 26] | Maximum visible length of a field. |

## .dim

You use the **`.dim`** attribute to determine the dimension of arrays or multiple value fields. This is important for step loops, which are represented as multiple-value fields.

Suppose a step loop consists of a column based on the field **`vbcom-kunde`**:

- If the column has 20 rows, **`vbcom-kunde.dim`** returns the value 20.
- If the column has only one row, **`vbcom-kunde.dim`** returns the value 1.
- If no field is defined, the value returned is 0.

You can access specific values of such an array by specifying an index in angle brackets. The value range of the index is from 1 to `.dim`.

The following code processes an array based on the **`vbcom-kunde`** field:

```
`repeat with I from 0 to vbcom-kunde.dim`
  <input type=text name="`vbcom-kunde[I].name`" value="`vbcom-
kunde[I].value`">
`end`
```

To access the second value of the field `vbcom-kunde`, you can use the expression:

```
Customer no. 2: `vbcom-kunde[2]`
```

The value of **`.dim`** must be within the defined dimension of the array:

- If you supply a negative index, a runtime error occurs.
- If you supply an index greater than the value defined by **`.dim`**, the result is an empty string.

## .disabled

You use the **.disabled** attribute to determine whether a screen field is ready for input or read-only.

The following code determines whether the **vbcom-kunde** field exists on the screen and whether it is ready for input:

```
`if ( vbcom-kunde.exists )`
  `if ( vbcom-kunde.disabled )`
     <! Value only -->
     `vbcom-kunde`
  `else`
     <! ready for input -->
     <input type="text" name="vbcom-kunde" value "`vbcom-
kunde`">
  `endif`
`endif`
```

# .exists

You use the **`.exists`** attribute to determine whether or not a screen field exists on the current screen.

The following code determines whether the **`vbcom-kunde`** field exists on the screen and whether it is ready for input:

```
`if ( vbcom-kunde.exists )`
  `if ( vbcom-kunde.disabled )`
     <! Value only -->
     `vbcom-kunde`
  `else`
     <! ready for input -->
     <input type=”text” name="vbcom-kunde" value “`vbcom-
kunde`”>
  `endif`
`endif`
```

## .label

You use the **`.label`** attribute to determine the text description of an input field:

The following code determines the text description of the **`vbcom-kunde`** field:

```
`vbcom-kunde.label`: <input type=text name="vbcom-kunde">
```

Like language resource files, this attribute allows language independence in HTML pages. When you use the `.label` attribute, the R/3 logon language determines the texts that are displayed on the HTML page.

For further information, see Using Language Resource Files [Page 87].

## .maxSize

You use the **`.maxSize`** attribute to determine the maximum input length of a screen field:

The following code determines the maximum input length of the **`vbcom-kunde`** field:

```
<p> Please enter your customer number
   <input type=text name="vbcom-kunde"
          maxSize=`vbcom-kunde.maxSize` </p>
```

## .name

You use the **`.name`** attribute together with the **`.value`** attribute to construct valid field names.

The **`.name`** attribute outputs the name and index of a field.

To create an input tag for the multiple value field **`vbcom-kunde`**, you could write

```
`repeat with I from 0 to vbcom-kunde.dim`
  <input type=text name="vbcom-kunde[`I`]" value="`vbcom-
kunde[I]`">
`end`
```

This results in the following output:

```
<input type=text name="vbcom-kunde[1]" value="4711">
<input type=text name="vbcom-kunde[2]" value="8523">
<input type=text name="vbcom-kunde[3]" value="1234">
```

The sequence of back quotes and quotes can be confusing. Instead, you can use the `.name` and `.value` attributes to write:

```
`repeat with I from 0 to vbcom-kunde.dim`
  <input type=text name="`vbcom-kunde[I].name`" value="`vbcom-
kunde[I].value`">
`end`
```

The output is identical to above:

```
<input type=text name="vbcom-kunde[1]" value="4711">
<input type=text name="vbcom-kunde[2]" value="8523">
<input type=text name="vbcom-kunde[3]" value="1234">
```

## .type

You use the `.type` attribute to determine the type of a screen field.

To determine whether a particular field is a combo box, use the following syntax:

```
`if <field>.type==ComboBox`
    (action)
`endif`
```

The possible values for the `.type` attribute are listed in the following table:

| Field Attribute | Possible Values |
|---|---|
| `.type` | `TableView` |
| | `TableColumn` |
| | `TableSelector` |
| | `TableColTitle` |
| | `Label` |
| | `Frame` |
| | `PushButton` |
| | `RadioButton` |
| | `CheckButton` |
| | `Password` |
| | `ComboBox` |
| | `Edit` |
| | `Unknown` |

## .value

You use the **`.value`** attribute together with the **`.name`** attribute to construct valid field names.

The **`.value`** attribute outputs the value of a field.

To create an input tag for the multiple value field **`vbcom-kunde`**, you could write

```
`repeat with I from 0 to vbcom-kunde.dim`
  <input type=text name="vbcom-kunde[`I`]" value="`vbcom-
kunde[I]`">
`end`
```

This results in the following output:

```
<input type=text name="vbcom-kunde[1]" value="4711">
<input type=text name="vbcom-kunde[2]" value="8523">
<input type=text name="vbcom-kunde[3]" value="1234">
```

The sequence of back quotes and quotes can be confusing. Instead, you can use the **`.name`** and **`.value`** attributes to write:

```
`repeat with I from 0 to vbcom-kunde.dim`
  <input type=text name="`vbcom-kunde[I].name`" value="`vbcom-
kunde[I].value`">
`end`
```

The output is identical to above:

```
<input type=text name="vbcom-kunde[1]" value="4711">
<input type=text name="vbcom-kunde[2]" value="8523">
<input type=text name="vbcom-kunde[3]" value="1234">
```

## .visSize

You use the **`.visSize`** attribute to determine the maximum visible length of a screen field:

The following code determines the maximum visible length of the **`vbcom-kunde`** field:

```
<p> Please enter your customer number
   <input type=text name="vbcom-kunde"
          size=`vbcom-kunde.visSize` </p>
```

# HTML$^{Business}$ Language Description

The following topics describe individual HTML$^{Business}$ language components.

# HTML$^{\text{Business}}$ Keywords

The following tokens are reserved keywords in HTML$^{\text{Business}}$.

**HTML$^{\text{Business}}$ Keywords**

| | | | |
|---|---|---|---|
| `archiveURL` | `assert` | `by` | `declare` |
| `define` | `else` | `elseif` | `elsif` |
| `end` | `for` | `from` | `if` |
| `imageURL` | `in` | `include` | `mimeURL` |
| `repeat` | `secure` | `times` | `to` |
| `wgateURL` | `with` | `write` | `writeEnc` |

You cannot use HTML$^{\text{Business}}$ keywords as identifiers. If you want to use an identifier that has the same name as a keyword, you must enclose the identifier in single quotation marks.

For example: `` `repeat with i from 'from' to 'to'`. ``

# HTML$^{Business}$ Expressions

HTML$^{Business}$ expressions are similar to those used in C, C++ or Java.

From ITS 2.2, there are new rules for

- Operator priority
- Operator associativity

## Operator Priority

In ITS versions earlier than 2.2, operator priority rules in HTML$^{Business}$ were not always intuitive, and differed from C, C++, or Java.

For instance, the `||` operator had priority over the == operator, so ITS developers had to use parentheses unnecessarily to clarify priority.

Instead of writing

```
`if (a==1 && b==2)`,
```

it was necessary to write

```
`if ((a==1) && (b==2))`.
```

Otherwise, the HTML$^{Business}$ interpreter would interpret the expression as

```
`if (a==(1 && b)== 2)`
```

To overcome this problem in ITS 2.2, operator priority has been redefined to match that used by the programming languages C, C++, and Java:

Currently, HTML$^{Business}$ provides the following operators, listed in decreasing order of priority:

**Operator Priority**

| Operator | Priority |
|---|---|
| `++, --` | 1 |
| `*, /, %` | 2 |
| `+, -, &` | 3 |
| `==, !=, >, <, >=, <=` | 4 |
| `&&, ||` | 5 |

If you need a different evaluation sequence, you must use parentheses. Operators with the same priority are evaluated from left to right.

## Operator Associativity

In ITS versions earlier than 2.2, operator associativity also differed from C, C++, or Java in some (but not all) cases, since terms were evaluated from right to left (rather than from left to right).

In ITS 2.2, operator associativity has been changed to evaluate from left to right, in order to achieve consistency with other programming languages.

Therefore, the expression

```
`8/2*4`
```

is now evaluated as

```
`(8/2)*4`  (==16)
```

instead of

```
`8/(2*4)`  (==1).
```

## Expression Syntax

The syntax summarized in the table below specifies the currently allowed forms of expression:

- Operators with identical priority are grouped and evaluated from left to right.
- If you want to enforce a different evaluation sequence, you must use parentheses.

| Nonterminal | Derivation |
|---|---|
| `expression` | `simpleexpr [compop simpleexpr]` |
| `simpleexpr` | `term { addopr  simpleexpr}` |
| term | `factor { mulopr factor}` |
| factor | `(! \| ++ \| --) factor`<br>`( expression) \|`<br>`function call \|`<br>`assignment \|`<br>`lvalue [++ \| --] \|`<br>`constant` |
| `function call` | `internalfn ( argument {, argument}) \|`<br>`externalfn ( expression {, expression})` |
| `internalfn` | `write \| writeEnc \| wgateURL \| archiveURL \| imageURL \| mimeURL \| assert` |
| `mulopr` | `* / % &&` |
| `addopr` | `+ - & \|\|` |
| `compop` | `== \| != \| > \| < \| >= \| <=` |

For information on the notation used in the table, see Syntax Conventions [Page 14].

The following are examples of correct expressions:

```
vbcom-kunde
nCustomers % 10
!fExists
a > b*2+1
```

```
name != "Walt"&" "&"Whitman"
(x -y) * (a+b) & " US$"
cond1 && (cond2 || cond3) && cond4
```

## HTML$^{Business}$ Functions

HTML$^{Business}$ provides a set of standard functions that generate HTML fragments. These functions can only be developed within the HTML$^{Business}$ interpreter itself, so the set of HTML generation macros cannot be extended.

To increase flexibility, HTML$^{Business}$ also allows you to write your own functions, but this does require some knowledge of the underlying concepts behind procedural languages.

The standard functions provided are:

archiveURL Function [Page 33]

assert Function [Page 34]

imageURL Function [Page 35]

mimeURL Function [Page 38]

printf Function [Page 41]

strCat Function [Page 43]

strCmp Function [Page 44]

strCpy Function [Page 45]

striCmp Function [Page 46]

strLen Function [Page 47]

strLwr Function [Page 48]

strnCmp Function [Page 49]

strniCmp Function [Page 50]

strStr Function [Page 51]

strSub Function [Page 52]

strUpr Function [Page 53]

toLower Function [Page 54]

toUpper Function [Page 55]

wgateURL Function [Page 56]

write Function [Page 59]

writeEnc Function [Page 60]

HTMLBusiness Functions [Page 61]

For further information about writing your own HTML$^{Business}$ functions, see:

HTMLBusiness Function Specification [Page 61]

# archiveURL Function

## Purpose

Accesses functions in the iXOS archive.

## Syntax

```
archiveURL(command, archiveID=<expression>, docID=expression)
```

## Description

Use the **`archiveURL`** function in the same way as **`wgateURL`** to access the iXOS archive system. You specify the appropriate iXOS function using the **`command`**, **`archiveID`** and **`docID`** parameters. Further information about the iXOS archive is available in the appropriate product documentation.

The individual parameters for the function you are calling should be taken from the product description of the accompanying Archive Web DLL. The relevant Archive Web DLL must be installed correctly on the Web server.

The URL used by **`archiveURL`** is specified in the global service description **`global.srvc`** under the **`~URLarchive`** parameter.

# assert Function

## Purpose

Generates a standard message when an HTML field value is in error.

## Syntax

**`assert(<field name>)`**

## Description

Use the **`assert`** function to generate field-specific error messages in an HTML page.

In an R/3 transaction, a screen field generates an error, if the field contents are found to be invalid. In this case, R/3 uses the cursor position to determine the field responsible for the error. On the ITS side, a standard error message is placed in the system field ~**`MessageLine`**.

In an HTML template, if you use the **`assert`** function for a field, and that field generates an error in R/3, the ITS replaces your **`assert`** statement with the standard error message. With the **`assert`** function, you can have a message (or graphic) displayed next to a field whenever it contains error data.

Normally, the Internet Transaction Server (ITS) replaces the **`assert`** with the **`~ErrorMarker`** value from the **`global.srvc`** service description. This parameter may contain any HTML statement (for example, a hyperlink to a graphic). If you don't set **`~ErrorMarker`**, the default is: "@@ Error @@".

```
<form …>
…
Material no <input type="text" name="matnr">`assert(matnr)`
Quantity    <input type="text"
                   name="quantity">`assert(quantity)`
   <p>`~MessageLine`</p>
</form>
```

Please note that since only a straight replacement occurs, HTML$^{Business}$ statements in this parameter are not analyzed.

## imageURL Function

### Purpose

Generates dynamic links to graphics files, based on language and theme.

### Syntax

```
imageURL(~type=<expression>, ~name=<expression> {,~theme=<expression>,}
{,~language=<expression>})
```

### Parameters

| Parameter | Meaning |
|---|---|
| `~name` | Graphic file name. |
| `~type` | Graphic file type. |
| `~language` | Current logon language. |
| `~theme` | Theme. |

### Description

As of Internet Transaction Server (ITS) version 1.1 (R/3 Release 3.1H), the **`imageURL`** function is obsolete. For later versions, use the mimeURL Function [Page 38] function instead. Currently, the **`imageURL`** function is supported only to ensure compatibility with existing HTML templates.

The **`imageURL`** function specifies links to graphics files. These files are not stored in the ITS directory but in the directory of the HTTP server.

When specifying links to graphics files, you cannot specify a simple relative URL. The URL must contain parameters that tell the type of graphic file and as well as the service and theme to which it belongs. **`imageURL`** also lets you access images according to their language. This allows the management of multi-lingual images.

Except for the name, specify all parameters relative to the sub-directory within the ITS directory system. The URL generated depends on these values and the **`~URLimage`** parameter defined in the global service file **`global.srvc`**. The **`imageURL`** function creates the basic URL as follows:

```
<~URLimage>/<~type>/<~language>/<~theme>/<~name>
```

or specifically:

```
/<HTTP server root directory>/SAP/ITS/GRAPHICS/<type>/<language>/<theme>/<name>
```

If the **`~Language`** and **`~Theme`** parameters are not specified, the values from the service description or the current logon language are used. The values assigned to the parameters are taken without checks from the ITS to make up the URL. In this way, you can create further sub-directories (linked to the graphics directory) and they can be addressed via the **`imageURL`** function.

Values that are not defined or are empty are removed from the directory structure.

Some examples (where `~language` is **EN** and `~theme` is not defined):

1. `<img src="`imageURL(~type="backgrounds", ~name="marmor.gif")`">`

   results in:

   `<img src="/sap/its/graphics/backgrounds/EN/marmor.gif">`

2. `<img src="`imageURL(~language="", ~name="marmor.gif")`">`

   results in:

   `<img src="/sap/its/graphics/marmor.gif">`

3. `<img src="`imageURL(~theme="ides", ~name="marmor.gif")`">`

   results in:

   `<img src="/sap/its/graphics/EN/ides/marmor.gif">`

4. `<img src="`imageURL(~type="backgrounds", ~theme="ides", ~language="DE", ~name="marmor.gif")`">`

   results in:

   `<img src="/sap/its/graphics/backgrounds/DE/ides/marmor.gif">`

# includeFrame Function

## Purpose

Converts multi-frame applications to single pages.

## Syntax

```
includeFrame(~framename=<frame name>)
```

## Parameters

| Parameter | Meaning |
|---|---|
| `~framename` | Name of frame to be included. |

## Description

If you have a multi-frame application, you can use this function to convert the multiple frames to a single page

For example, you could implement a simple two-frame application with the **`<frameset>`** … **`</frameset>`** tag as follows:

```
<frameset rows=80,* FRAMEBORDER=0 BORDER=0 FRAMESPACING=0>
<frame name="FRAME_1" SRC="`wgateURL(~FrameName="FRAME_1")`"
scrolling="no">
<frame name="FRAME_2" SRC="`wgateURL(~FrameName="FRAME_2")`">
</frameset>
```

To convert this multiple frame application to a single frame HTML page, you remove the **`<frameset>`** … **`</frameset>`** structure and use the **`includeFrame`** function with the **`<table>`** … **`</table>`** tag instead:

```
<table>
  <TR><TD>`includeFrame(~framename="FRAME_1")`</TD></TR>
  <TR><TD>`includeFrame(~framename="FRAME_2")`</TD></TR>
</table>
```

When you execute this code, **`FRAME_1`** and **`FRAME_2`** are still displayed in 2 rows, but as a single page rather than two separate frames.

# mimeURL Function

## Purpose

Generates dynamic links to graphics files, based on ITS language and theme.

## Syntax

```
mimeURL ( [ [~service=] expression, ] [ [~theme=] expression,] [ [~language=]
expression,] [~secure=]expression,] [~name=] expression)
```

## Parameters

| Parameter | Meaning |
|---|---|
| **`~service`** | Service.<br>If omitted, the name of the current service is used.<br>Special service names are **`global`** (which covers all files that used by several servers) and **`system`** (which is used for system messages). |
| **`~theme`** | Theme.<br>If omitted, the current theme (defined in the relevant service file) is used. To omit the whole theme part of the URL path, specify **`~theme=""`**. |
| **`~language`** | Language.<br>If omitted, the current language (defined in the relevant service file or using the logon page) is used. To omit the language part of the URL path, specify **`~language=""`**. |
| **`~secure`** | Specifies whether a relative or absolute URL should be used.<br>• **`~secure`** not used at all:<br>**`mimeURL()`** creates a relative URL, as in ITS versions prior to 2.2.<br>• **`~secure="on"`**:<br>**`mimeURL()`** creates an absolute URL and uses the protocol type HTTPS. Use this value to switch from an HTTP (unsecure) communication to an HTTPS (secure) communication within a running session.<br>• **`~secure="off"`**:<br>**`mimeURL()`** creates an absolute URL and uses the protocol type HTTP. Use this value to switch from an HTTPS (secure) communication to an HTTP (unsecure) communication within a running session.<br>For examples of using this parameter, see the wgateURL [Page 56] function. |
| **`~name`** | Name (and optional subdirectories) of the file to be referenced.<br>Examples are **`ok.gif`** or **`buttons/roundones/cancel.gif`**. |

You can omit parameter names. This results in shorter but less comprehensible function calls. If you omit the parameter names, the default order is **`~service`**, **`~theme`**, **`~language`**, **`~secure`**, **`~name`**.

All parameters except **`~name`** are optional and are derived from the current session context if not already defined as an argument. Apart from the file name, specify all other parameters relative to the subdirectory within the ITS directory system. **`mimeURL`** forms the path name according to the following pattern:

```
/<HTTP server root directory>/SAP/ITS/MIMES/<service>/<theme>/<language>/<name>
```

If the **`~service`**, **`~theme`** and **`~language`** parameters are not specified, the corresponding values from the service description are used. If you specify the **`~language`** parameter without a value (that is, **`~language=""`**), **`mimeURL`** uses the files in the parent **`Theme`** directory. This allows you to access language independent files.

## Description

Use the **`mimeURL`** function to specify links to static files, for example graphics and help files that are to be incorporated into the HTML templates at runtime. These files are stored in the directory of the HTTP server, not in the ITS directory.

You cannot use a simple relative URL for this purpose. You must use a URL containing parameters that assign the static file to a specific service, theme and language.

The **`mimeURL`** function is intended to replace the **`imageURL`** function. **`mimeURL`** provides more flexible and easier access to service-, language-, and theme-dependent files like images, sounds and other multimedia data (hence the name **`mimeURL`**).

Apart from addressing specific service files, you can use the **`mimeURL`** function to address global service files using the **`~service="global"`** parameter.

The values assigned to the parameters are taken without checks from the ITS to make up the URL. You can thus create further subdirectories linked to the **`MIMES`** directory which are addressed via the **`mimeURL`** function.

The following example illustrates the use of **`mimeURL`**.

If **`~URLmime`** is set to `/sap/its/mimes`, the current service is `vw01`, the current language is `EN` and the current theme is `99`, calling **`mimeURL()`** will result in the following output:

1. **`<img src="`mimeURL(~name="ok.gif")`">`**

   `results in:`

   **`<img src="/sap/its/mimes/vw01/99/EN/ok.gif">`**

2. **`<img src="`mimeURL(~service="global", ~name="ok.gif")`">`**

   `results in:`

   **`<img src="/sap/its/mimes/global/99/EN/ok.gif">`**

3. **`<img src="`mimeURL("", ~name="buttons/ok.gif")`">`**

   `results in:`

   **`<img src="/sap/its/mimes/99/EN/buttons/ok.gif">`**

## mimeURL Function

4. `<img src="`mimeURL("","buttons/roundones/ok.gif")`">`

   results in:

   `<img src="/sap/its/mimes/99/EN/buttons/roundones/ok.gif">`

5. `<img src="`mimeURL("", 1, "buttons/roundones/ok.gif")`">`

   results in:

   `<img src="/sap/its/mimes/1/EN/buttons/roundones/ok.gif">`

6.
   ```
   <img src="`mimeURL("global", 1, "DE",
   "buttons/roundones/ok.gif")`">
   ```

   results in:

   `<img src="/sap/its/mimes/global/1/DE/buttons/roundones/ok.gif">`

7. `<img src="`mimeURL(~service="system", ~name="ok.gif", 1, "")`">`

   results in:

   `<img src="/sap/its/mimes/system/1/ok.gif">`

# printf Function

## Purpose

Returns a string using the specified formatting.

## Syntax

```
string  printf  (in string  format,...)
```

## Parameters

| | |
|---|---|
| Format | Format string (template) from which to create the final string. The format string contains "%" format specifiers that are replaced by parameters passed as the...-arguments. |
| … | A variable number of arguments. |

## Description

Returns a string built from the given format string by resolving format specifiers from a given variable argument list (like the C **`printf()`** function). Format specifiers have the following form:

**`%<flags><width><.pre><conversion>`** where:

| | | |
|---|---|---|
| **`<flags>`** can contain zero or more of: | - | Left justify within field. |
| | + | Display leading sign. |
| | &lt;space&gt; | Prefix space to result instead of sign. |
| | 0 | Leading zeros. |
| **`<width>`** | | Number of characters to expand to (optional). |
| **`<pre>`** | | Precision to use for (optional after.). |
| diouxX | | Minimum number of digits. |
| s | | Maximum number of chars. |
| **`<conversion>`** contains one of: | % | "%" character |
| | d | Signed decimal integer |
| | I | Signed integer |
| | u | Unsigned decimal integer |
| | x, X | Unsigned hexadecimal integer |
| | o | Unsigned octal character |
| | c | Display as character |
| | s | String |

## Return Value

Returns a string constructed from the format string by replacing all "%" format specifiers.

I
c
o
n
s

Icon
Meaning

## Example

```
`printf ("Lawrence Smith", format="%-20s will be opening %s in %s on %s
   %02d, %04d.",
   "a menswear store", "Billingsley", "January", 1, 1999)`
```

# strCat Function

## Purpose

Appends one string onto another.

## Syntax

```
string  strCat  (inout string  destString,
                  in    string  sourceString)
```

## Parameters

| | |
|---|---|
| **`destString`** | Variable specifying destination for append operation. |
| **`source String`** | Expression specifying source for append operation. |

## Description

This function concatenates strings and places the result in the destination string.

Writing **`` `a = a & b` ``** is equivalent to calling **`` `strCat (a, b)` ``**.

## Return Value

Returns the content of **`destString`**.

## Example

```
`if (toLower(strCat(~language, "glish")) == "english")`
English Version:
`end`
```

# strCmp Function

## Purpose

Compares two strings for equality.

## Syntax

```
int  strCmp  (in string  string1,
              in string  string2)
```

## Parameters

| | |
|---|---|
| **string1** | Expression evaluating to a string. |
| **string2** | Expression evaluating to a string. |

## Description

This function compares two strings.

Writing `` `a == b` `` is equivalent to calling `` `strCmp (a, b)` ``.

## Return Value

This function returns:

- 0, if **string1** is equal to **string2**
- <0, if **string1** is less than **string2**
- >0, if **string2** is less than **string1**

## Example

```
`if (strCmp(toLower(~language), "en") == 0)`
English Version:
`end`
```

# strCpy Function

## Purpose

Creates a copy of a given string.

## Syntax

```
string  strCpy  (inout string  destString,
                in    string  sourceString)
```

## Parameters

| | |
|---|---|
| **destString** | Variable specifying the destination for the copy operation. |
| **sourceString** | Expression used as the source for the copy operation. |

## Description

This function copies a string value into a second location.

Writing **`a = b`** is equivalent to calling **`strCpy (a, b)`** .

## Return Value

Returns the content of **destString**.

## Example

```
`if (toLower(strCpy(l, ~language)) == "en")`
English Version:
`end`
```

## striCmp Function

### Purpose

Compares two strings for equality in a case-insensitive manner.

### Syntax

```
int  striCmp  (in string  string1,
               in string  string2)
```

### Parameters

| | |
|---|---|
| **string1** | Expression evaluating to a string. |
| **string2** | Expression evaluating to a string. |

### Description

Writing `` `toLower(a) == toLower(b)` `` is equivalent to calling `` `striCmp (a, b)` ``.

### Return Value

This function returns:

- 0, if **string1** is equal to **string2** disregarding case
- <0, if **string1** is less than **string2** disregarding case
- >0, if **string2** is less than **string1** disregarding case

### Example

```
`if (striCmp(~language, "en") == 0)`
English Version:
`end`
```

# strLen Function

## Purpose

Returns the number of characters in a given string.

## Syntax

```
int  strLen  (in string  string)
```

## Parameters

| | |
|---|---|
| `string` | String whose characters are to be counted. |

## Return Value

Returns the number of characters in the string.

## Example

```
`if (strniCmp(~language, "en", strLen("en")) == 0)`
English Version:
`end`
```

## strLwr Function

### Purpose

Returns a lowercase copy of the given string.

### Syntax

```
string  strLwr  (in string  string)
```

### Parameters

| | |
|---|---|
| `string` | String for which a lowercase copy is desired. |

### Return Value

Returns a lowercase copy of the given string

### Example

```
`if (strnCmp(strLwr(~language), "en", 1) == 0)`
English Version:
`end`
```

## strnCmp Function

### Purpose

Compares the first n characters of two strings for equality.

### Syntax

```
int  strnCmp  (in string  string1,
               in string  string2,
               in int     numberOfChars)
```

### Parameters

| | |
|---|---|
| **string1** | Expression evaluating to a string. |
| **string2** | Expression evaluating to a string. |
| **numberOfChars** | Number of characters to compare. |

### Return Value

This function returns:

- 0, if the first **numberOfChars** characters of **string1** are equal to the first **numberOfChars** characters of **string2**.
- <0, if the first **numberOfChars** characters of **string1** are less to the first **numberOfChars** characters of **string2**.
- >0, if the first **numberOfChars** characters of **string2** are less to the first **numberOfChars** characters of **string1**.

### Example

```
`if (strnCmp(toLower(~language), "en", 1) == 0)`
English Version:
`end`
```

# strniCmp Function

## Purpose

Compares the first n characters of two strings for equality in a case-insensitive manner.

## Syntax

```
int  strniCmp  (in string  string1,
                in string  string2,
                in int     numberOfChars)
```

## Parameters

| | |
|---|---|
| **string1** | Expression evaluating to a string. |
| **string2** | Expression evaluating to a string. |
| **numberOfChars** | Number of characters to compare. |

## Return Value

This function returns:

- 0, if the first **numberOfChars** characters of **string1** and **string2** are equal, regardless of case.
- <0, if the first **numberOfChars** characters of **string1** are less than those of **string2**, regardless of case.
- >0, if the first **numberOfChars** characters of **string2** are less than those of **string1**, regardless of case.

## Example

```
`if (strniCmp(~language, "en", 1) == 0)`
English Version:
`end`
```

# strStr Function

## Purpose

Searches for a given substring within a string.

## Syntax

```
int  strStr  (in string  string,
              in string  stringPattern)
```

## Parameters

| | |
|---|---|
| **string** | The string in which to search. |
| **stringPattern** | The string to search for. |

## Return Value

This function returns:

- 0, if **string** does not contain **stringPattern**.
- >0, if **string** contains **stringPattern**. In this case, the non-zero value tells the starting position where the substring was found.

## Example

The expression `` `strStr("Hello world", "world")` `` returns the position (position 7) at which "world" occurs in "Hello world".

# strSub Function

## Purpose

Returns a substring from the given string.

## Syntax

```
string  strSub  (in string  string,
                 in int     position,
                 in int     length)
```

## Parameters

| | |
|---|---|
| `string` | Expression evaluating to a string. |
| `position` | Starting position of the substring within string [1..n]. |
| `length` | Number of characters to put into the substring starting at `position`. |

## Description

This function returns a substring of a given string. You can specify the starting position and length, and `strSub` will return `length` characters, starting at the position you request.

If you specify a length of 0, this function returns all characters from the starting position to the end of the string. If `length` is larger than the available characters in the source string, all characters up to the end of the string are taken.

## Return Value

Returns a substring (a copy) of the given string.

## Example

```
`name = "Walt Whitman"`
The last name of `name` is
`if (0 != (j = strstr (name, " "))`

    `strSub (name, j + 1, 0)`
`else`
    undefined.
`end`
```

# strUpr Function

## Purpose

Returns an uppercase copy of the given string.

## Syntax

```
string  strUpr  (in string  string)
```

## Parameters

| | |
|---|---|
| `string` | String for which an uppercase copy is desired. |

## Return Value

Returns an uppercase copy of the given string

## Example

```
`if (strnCmp(strUpr(~language), "EN", 1) == 0)`
English Version:
`end`
```

# toLower Function

## Purpose

Returns an all lowercase copy of the given string.

## Syntax

```
string  toLower  (in string  theString)
```

## Parameters

| | |
|---|---|
| `theString` | Expression evaluating to a string. |

## Description

This function returns a lowercase copy of the input string. `toLower` does not change the input string itself.

## Return Value

Returns a lowercase copy of the string passed as `theString`.

## Example

```
`if (toLower(~language) == "en")`
English Version:
`end`
```

## toUpper Function

### Purpose

Returns an uppercase copy of a string.

### Syntax

```
string  toUpper  (in string  theString)
```

### Parameters

| | |
|---|---|
| `theString` | Expression evaluating to a string. |

### Description

This function creates an al uppercase version of the input string and returns. `toUpper` does not change the original string.

### Return Value

Returns an all uppercase copy of the string passed as `theString`.

### Example

```
`if (toUpper(~language) == "EN")`
English Version:
`end`
```

# wgateURL Function

## Purpose

Generates URLs dynamically that are compatible with the current Web server.

## Syntax

```
wgateURL(parameter = expression {, parameter = expression})
```

## Parameters

The following table shows the possible parameters of the **wgateURL** function.

| Parameter | Meaning |
|---|---|
| **~OkCode** | Function code to be triggered in R/3 transaction. |
| **~target** | Name of target frame for a **<form>** command.<br><br>The response to this request will appear in the target frame. For full details on using **~target**, see Browser Independence [Page 89].<br><br>(From ITS version 2.2, **~target** replaces **~forceTarget**.) |
| **~forceTarget** | No longer supported<br><br>From ITS version 2.2, use **~target** for all browsers. |
| **~FrameName** | Determines URL of a frame in a frame set document. |
| **~secure** | Specifies whether a relative or absolute URL should be used:<br>• **~secure** not used at all:<br>**wgateURL()** creates a relative URL as in pre-2.2 versions of ITS.<br>• **~secure="on"**:<br>**wgateURL()** creates an absolute URL and uses the protocol type HTTPS. Use this value to switch from HTTP (unsecure) communication to HTTPS (secure) communication within a session.<br>• **~secure="off"**:<br>**wgateURL()** creates an absolute URL and uses the protocol type HTTP. Use this value to switch from an HTTPS (secure) communication to an HTTP (unsecure) communication within a running session. |
| **Screen field name** | Fills screen fields in target frame. |

## Description

This function generates URLs dynamically that agree with the system information for the current HTTP server.

In order to let you communicate with the ITS software safely and simply, HTML$^{Business}$ provides the **wgateURL** function. Use this function to produce templates that are easily portable between servers. At runtime, **wgateURL** dynamically generates URLs that agree with the system information for the current HTTP server. In this way, you can avoid hard-coding URLs in a template.

In addition to including system information (such as the **~State** field), the **wgateURL** function also encodes all parameters according to URL requirements. (For details, see writeEnc Function [Page 60]).

Use **wgateURL** in the following three cases:

- In **<form>** commands, you need to specify the URL of the program that will process the contents of the form. This path depends on your ITS installation type and therefore needs to be determined dynamically.
- When you are using a transaction with several frames, this function gives the target frame. In this case, the URL of the ITS needs to be generated and additional parameters merged with it.
- Within a frame set document, this function gives the URL for a frame.

## Creating Anchors in the URL

Use the **~anchor** parameter to generate an HTML anchor in the **wgate URL**:

```
<form method="POST" action="`wgateURL(~anchor="footer")`">
```

This is expanded to:

```
<form method="POST" action="/scripts/wgate.dll/vw01#footer">
```

The following example shows how **wgateURL** works:

```
<ul>
  `repeat with j from 1 to xlist-matnr.dim`
  <li>
    <a href="`wgateURL(matnr=xlist-matnr[j], quantity=xlist-
                                                kwmeng[j])`">
    `xlist-matbez[j]` </a> </li>
  `end`
</ul>
```

This HTML template would be expanded as follows:

```
<ul>
  <li>
    <a
href="http://pn0208/scripts/wgate.dll/vw01?~State=4711&matnr=9
132&quantity=2">
    Microsoft Word Update </a> </li>
  <li>
```

```
    <a
href="http://pn0208/scripts/wgate.dll/vw01?~State=4711&matnr=9
133&quantity=1">
    Microsoft Excel Update </a> </li>
</ul>
```

Always make sure that the value of the attribute **`href`** is enclosed in double quotes (").

## Using the ~secure Parameter

You can use the system variable **`~http_https`** to query whether the current request is using HTTP or HTTPS. (For HTTP, **`~http_https`** = "off", and for HTTPS, **`~http_https`** = "on"). The following are examples of how to use the **`~secure`** parameter:

- **`<a href="`wgateURL(~okcode="back")`">`** expands to
  **`<a href="/scripts/wgate/vw01/~sd..f?~okcode=back">`**
- **`<a href="`wgateURL(~okcode="back", ~secure="on")`">`** expands to
  **`<a href="https://myHost:444/scripts/wgate/vw01/~sd..f?~okcode=back">`**
- **`<a href="`wgateURL(~okcode="back", ~secure="off")`">`** expands to
  **`<a href="http://myHost:1080/scripts/wgate/vw01/~sd..f?~okcode=back">`**
- **`<a href="`wgateURL(~okcode="back", ~secure=~http_https)`">`** expands to either the HTTP or the HTTPS variant, depending on the protocol type of the current request.

The **`~secure`** parameter requires additional information in the service file **`(<service>.srvc`**), usually the global service file (**`global.srvc`**):

| System Parameter | Description |
|---|---|
| **`~hostSecure`** | Host name and domain of the Web server to be used for secure requests (HTTPS).<br>If not specified, this parameter defaults to **`~hostUnsecure`**. |
| **`~portSecure`** | Port number to be used for secure requests (HTTPS).<br>If not specified, this parameter defaults to 443 (standard HTTPS port). |
| **`~hostUnsecure`** | Host name and domain of the Web server to be used for unsecure requests (HTTP).<br>If not specified, this parameter defaults to **`~http_host`** from the current request HTTP header. |
| **`~portUnsecure`** | Port number to be used for unsecure requests (HTTPS).<br>If not specified, this parameter defaults to 80 (standard HTTP port). |

As of ITS version 2.2, these system parameters are added to the global service file.

# write Function

## Purpose

Writes output to the HTML page without inserting spaces or other separators.

## Syntax

```
write ( <expression> {, <expression>})
```

## Description

The **write** function in HTML$^{Business}$ is similar to the same function in JavaScript. Both functions write field contents and other information to the HTML document.The only difference between **write** and simple replacement is that **write** outputs individual arguments without inserting spaces or other separators between the values.

Examples:

```
<p> `write (i, ". ", xlist-matnr)` </p>
<p> `write (j * 2, 123, j > 3)` </p>
```

# writeEnc Function

## Purpose

Writes output to the HTML page, converting all non-alphanumeric characters to hexadecimal.

## Syntax

```
writeEnc ( <expression> {, <expression>})
```

## Description

The **`writeENC`** function works like **`write`**, except that the output is encoded according to URL requirements. All non-alphanumeric characters (spaces and special characters) are converted into their hexadecimal equivalents. This function lets you construct URLs correctly from fields that contain spaces and special characters.

The difference between **`write`** and **`writeEnc`** can be illustrated using the following example:

```
<a href="http://pn0208/scripts/any.dll?matbez=`write
(matbez[j])`&quantity=1">
```

The least favorable expansion of the above code example is:

```
<a href="http://pn0208/scripts/any.dll?matbez=Large
Chairs&quantity=1">
```

Spaces like the one in **`Large Chairs`** are not permitted in URLs. However, if **`writeEnc`** is used:

```
<a
href="http://pn0208/scripts/any.dll?matbez=`writeEnc(matbez[j]
)`&quantity=1">
```

the following is completely correct. (The string “Large” and the string “Chairs” have been separated by the code for a space.)

```
<a
href="http://pn0208/scripts/any.dll?matbez=Large%20Chairs&quan
tity=1">
```

# HTML$^{Business}$ Function Specification

## General Principles

To support the design of HTML templates when implementing Internet Application Components (IACs), you can write your own HTML$^{Business}$ functions, which are syntactically similar to those used in programming languages like ABAP or C.

For example, a simple HTML$^{Business}$ function **concatenate_string** that concatenates two strings could have the following form:

```
`function concatenate_string (string1,string2)
   result = string1 & string2;
   return (result);
 end;`
```

You pass the two strings to be concatenated in the parameters **string1** and **string2**, and then return the **result**.

You could call the function **concatenate_string** as follows:

```
…
`write (concatenate_string("A","B"));`
…
```

This would produce the output **AB**.

The advantage of defining HTML$^{Business}$ functions is that you can reuse them as required in as many templates as you like. This saves you from having to redefine the code every time.

SAP does not deliver a standard library of reusable HTML$^{Business}$ functions. Rather, it is up to your development team to define functions that meet your specific application's requirements.

## Defining HTML$^{Business}$ Functions

The function definition specifies:

- The name of the function
- The number and the names of the parameters the function can expect to receive
- A function body containing the statements that determine what the function does
- A return value

HTML$^{Business}$ functions must be defined before they can be used. You can place a function definition anywhere in the code, but **not** within any other HTML$^{Business}$ statement. For example, you cannot embed a function definition in an **if** statement.

## HTML$^{Business}$ Function Syntax

The basic syntax of an HTML$^{Business}$ function definition is:

```
function <function name> (<parameter list>)
   <function body>
   return ( <return value> );
end;
```

The syntax components are summarized in the following table:

**HTMLBusiness Function Specification**

| Component | Description |
|---|---|
| `<function name>` | Unique identifier, which consists of an unbroken sequence of alphanumeric characters and permitted special characters.<br>HTML$^{Business}$ function names **must** begin with a letter. After this letter, you can use any combination of upper case letters from **A** to **Z**, lower case letters from **a** to **z**, and digits from **0** to **9**.<br>Blanks are not allowed, but you can use the underscore character **_** to join separate words that make up the function name. |
| `<parameter list>` | The names of the parameters you want to pass.<br>The default expression is used whenever the caller does not provide a parameter. |
| `<function body>` | Any valid HTML$^{Business}$ statement. |
| `<return value>` | Any valid HTML$^{Business}$ statement. |

## Calling HTML$^{Business}$ Functions

You can call HTML$^{Business}$ functions as part of any valid HTML$^{Business}$ statement. The syntax is:

```
`
…
<function name> (<parameter expression>)
…
`
```

The **`<parameter expression>`** consists of a list of parameter names, each separated by a comma. For each parameter name, the syntax is:

**`<parameter name>=<any valid HTML`$^{Business}$ `expression>`**

When an HTML$^{Business}$ function is called, each parameter expression is evaluated and the resulting value is assigned to the matching parameter identifier.

- If parameters are identified with parameter names, a matching parameter name is looked up in the function definition. If no matching name is found, a runtime error occurs.
- If parameters are not identified with parameter names, they are regarded as positional parameters and are copied one by one into the matching parameter identifier by copying the value of the n$^{th}$ parameter expression into the n$^{th}$ parameter identifier.

This means that you can call the function **`concatenate_string`** in two ways:

- With parameter names, as in:

  ```
  concatenate_string(string1="prefix",string2="suffix") ;
  ```

- With positional parameters, as in:

  ```
  concatenate_string("prefix","suffix") ;
  ```

A function can call another function, provided the other function has been declared. In the following example, a function **`func_1`** calls a function **`func_2`**, but this would result in a compile time error, because the function **`func_2`** has not been declared when it is called.

```
`
function func_1 (x)
   write(func_2(2*x));   <!—compile error -->
end;

function func_2 (y)
   write(y);
end;

func_1(10);
`
```

## Calling HTML$^{Business}$ Functions Recursively

An HTML$^{Business}$ function can call itself. The following example function calculates factorials:

```
`
function factorial(f,x=1)
   if (x==f)
      return (x);
   else
      return (x*factorial(f,x+1));
   end;
end;

repeat with i from 1 to 10;
   factorial(i);
end;
`
```

## Return Values in HTML$^{Business}$ Functions

HTML$^{Business}$ functions can return values to the caller.

If an HTML$^{Business}$ function returns a string value, which is then used in an expression that results in a numerical value, automatic type conversion is performed.

In the following function **dup_string**, which duplicates strings, all the statements specified below are possible:

```
`
function dup_string (s)
  return (s&s);
end;

dup_string(„a“);                              <-- outputs “aa“ -->

write(dup_string(„b“));                  <-- outputs “bb“ -->

dup_string(dup _string(“c”));       <-- outputs “cccc“ -->

2*dup_string(“1”);                            <-- outputs “22“ -->
`
```

## Variable Scope in HTML$^{Business}$ Functions

If a parameter of an HTML$^{Business}$ function has the same name as a global variable, that parameter contains the value supplied by the caller within the function, but the value of the global variable is restored when processing returns from the function to caller, as shown in the following example:

```
`
…
function write_numbers ( x1,x2,x3 )
   global_variable = x1;
   write(„global_variable is „,global_variable,"\n");
   write(„x1 is „,x1,"\n");
   write(„x2 is „,x2,"\n");
   write(„x3 is „,x3,"\n");
end;

global_variable = 1000;
x1 = 100;
x2 = 200;
x3 = 300;
write(„Before the call\n");
write("global_variable is ",global_variable,"\n");
write("x1 is ",x1,"\n");
write("x2 is ",x2,"\n");
write("x3 is ",x3,"\n");
write("Calling the function . . .\n”);
write_numbers(10,20,30);
write("After the call\n”);
write("global_variable is ",global_variable,"\n");
write("x1 is ",x1,"\n");
write("x2 is ",x2,"\n");
write("x3 is ",x3,"\n");
…
`
```

This produces the following output:

```
Before the call
global_variable is 1000
x1 is 100
x2 is 200
x3 is 300
Calling the function . . .
global_variable is 10
x1 is 10
x2 is 20
x3 is 30
After the call
global_variable is 10
x1 is 100
x2 is 200
x3 is 300
```

## Standard HTML in HTML$^{Business}$ Functions

HTML$^{Business}$ functions can contain standard HTML code.

In the following example, an HTML$^{Business}$ function called **`write_html_text`** outputs the HTML text **`<H1> This is standard HTML </H1>`** ten times in the Web browser.

```
`
function write_html_text()
`  <!--Now entering HTML mode -->
   <H1> This is standard HTML </H1>
`end;`

`repeat with I from 1 to 10;
   write_html_text();
end;
`
```

## Field References

You cannot pass parameters to fields by reference. Consider the following example:

```
`
…
function dump_table(t)
   `<TABLE>`
   repeat with i from 1 to t.dim;
      `<TR><TD>`t[i]`</TD></TR>`
   end;
   `</TABLE>`
end;

mytable[1] = „Line 1";
mytable[2] = „Line 2";
mytable[3] = „Line 3";
dump_table(mytable);
…
`
```

In this case, HTML$^{Business}$ outputs only the first line of the table because **`mytable`**, passed as a parameter to the function **`dump_table`**, is evaluated as the value of the first line of the multi-value field **`mytable`** which is **`Line 1`**.

To resolve the name at runtime, you must pass the name of the variable as a string instead and use the HTML$^{Business}$ **^** operator, as shown below:

```
`
…
function dump_table(t)
   `<TABLE>`
   repeat with i from 1 to ^t.dim;
      `<TR><TD>`^t[I]`</TD></TR>`
   end;
   `</TABLE>`
end;

mytable[1] = „Line 1";
mytable[2] = „Line 2";
mytable[3] = „Line 3";
dump_table(mytable);
…
`
```

## Application Example

One useful application is to write HTML$^{Business}$ functions that generate HTML code. For example, you could define a function **screenfield** as follows:

```
`
function ( ~screenfield )
   if (^~screenfield.disabled)
      <!--- Screenfield is not ready for input -->
      write(^screenfield.label);
   else
      write("<INPUT TYPE=\"");
      if (^screenfield.type=="RadioButton")
         write("RADIO\"");
         write(" NAME=\"",^screenfield.group,"\"");
         write(" VALUE=\"",^screenfield.name,"\"");
      else
         if (^screenfield.type=="CheckButton")
            html_type = "CHECKBOX";
         else
            html_type = "TEXT";
         end;
         write(html_type,"\"");
         write(" NAME=\"",^screenfield.name,"\"");
         write(" VALUE=\"",^screenfield.value,"\"");
      end;
   end;
end;
`
```

You could call the function **screenfield** in two ways:

- With parameter names, as in:

```
screenfield (~screenfield="VBAK-VBELN");
```

- With positional parameters, as in:

```
screenfield ("VBAK-VBELN");
```

Function calls can evaluate expressions, so you can also say:

```
this_field = „VBAK-VBELN";
dnprofield (this_field);
```

## Including HTML$^{Business}$ Functions

You can define HTML$^{Business}$ functions in one template, and then include them in any other template where you want to use them.

In the following fragment example, it is assumed that the function **screenfield** is defined in the template **util.html**. In order to use **screenfield**, another HTML$^{Business}$ template must therefore include **util.html**:

```
`
include(~service="util",~theme="",~name="util.html");

screenfield(~screenfield="VBAK-VBELN");
screenfield(~screenfield="VBAK-BELNR");
screenfield(~screenfield="VBAK-DATUM");
```

`

Although the **`include`** statement assigns default values to the parameters and does not generate an error message if parameters are missing, you **must** list the parameters **`~service`**, **`~theme`** and **`~name`** explicitly, because the HTML$^{\text{Business}}$ interpreter differentiates between compile time includes and runtime includes.

Compile time includes are already included at compile time of the including template. This is necessary so that the functions defined in the included template are known.

## HTML$^{Business}$ Statements

HTML$^{Business}$ provides the following statements:

# for Statement

## Purpose

Repeats a substitution in a **for** loop.

## Syntax

```
for ( expression ; expression ; expression) statement end
```

## Description

You can use the **for** statement just as in C or JavaScript. However, in contrast to C, you cannot list several expressions separated by commas.

The use of the **for** loop can be illustrated by the following examples:

```
`for (j = 10; j > 0; j--)` <td> `end`
`for (j = 1; j <= array.dim ; j++)`
  Array[`j`]=`array[j]`
`end`
`for (a = "a"; a != "aaaa"; a = a & "a") a end`
```

For information on how HTML$^{\text{Business}}$ evaluates expressions, see Expressions [Page 29].

For information on using **for** to code repeat loops, see repeat Statement [Page 77].

# if Statement

## Purpose

Performs conditional substitutions on an HTML$^{\text{Business}}$ expression.

## Syntax

```
if ( expression) statement
{ [elsif | elseif] ( expression) statement }
[else statement ]
end
```

## Description

The `if` statement allows you to perform conditional substitution in your HTML document. `if` forces the ITS to test some condition before performing a substitution.

The `if` statement in HTML$^{\text{Business}}$ is similar to the syntax and semantics of other common programming languages such as C and JavaScript. Since nesting is possible, `if` statements can contain other `if` statements. The same embedding is also allowed for the `repeat` statement.

The key words `elsif` and `elseif` can be used as alternatives.

In the syntax description, the term `expression` refers to any expression that evaluates to a truth value, that is, to 0 (FALSE) for conditions not met, or not equal to 0 (TRUE) for conditions met.

The following are examples of this kind of expression:

```
VBCOM-KUNDE >= 1000
VBCOM-KUNDE
j % 3 == 0
j / (4 - 1) != a * (b + (c + 2))
s == "Walter" & " " & "Weissmann"
(x > 2 && x <99) || (s > "abc")
```

The following are examples of `if` statements:

Example 1

```
`if (VBCOM-KUNDE) VBCOM-KUNDE else` Undefined Customer Number
`end`
```

Example 2

```
`if (j % 3 == 0)` <TR> `else` <TD> `end`
```

Example 3

```
`if (1)    write("This branch is always true!")
 elsif (0) write("This branch is never true!")
 else      write("The impossible occurred!")
end`
```

Example 4

```
`if (x > 0 && x < 100)`
   x is greater than 0 and smaller than 100
   `if (y > 0 && y < 100)`
      y is greater than 0 and smaller than 100
   `elsif (Y > 100)`
      y is greater than 100
`end
  elsif (x <= 0)`
    x is smaller than 0!
  `else`
    x is greater than 99!
`end`
```

The operators <, <=, > and >= are not defined for character strings. If you use them with character strings, the strings are automatically converted into numerical values.

# include Statement

## Purpose

Includes code from other HTML templates in the current HTML template.

## Syntax

```
include([[~service=]expression,] [[~theme=]expression,]
[[~language=]expression,] [~name=]expression)
```

## Parameters

| Parameter | Meaning |
|---|---|
| `~service` | Service name.<br>If omitted, the current service is used. |
| `~theme` | Theme.<br>If omitted, the current theme (defined in the `.srvc` file) is used. To omit the theme part of the URL path, specify `~theme=""`. |
| `~language` | Language.<br>If omitted, **no** language is used. If you want to use the current language, specify `~language=~language`. This behavior contrasts with the `~language` parameter of the `mimeURL` statement. |
| `~name` | File name.<br>The file to be included is taken from a path constructed as follows:<br>`<itsRootDir>\<virtual ITS>\templates\<~service>\<~theme>\<filename(~name)>[_<~language>].<extension(~name)>\|html)` |

## Description

The `include` statement allows you to include code from other HTML templates in the current HTML template.

### Advantages

The `include` statement has the following advantages:

- Consistency of style for Internet Application Components (IACs)

  IACs often use a shared set of style elements such as title bars or copyright footers. You can store these elements in a single file and include it in all the relevant HTML templates. This approach allows for centralized updating.

- Reuse of common code

  You can reuse Java or Visual Basic scripts in multiple templates. This is useful for implementing commonly used routines such as validity checking of user input.

- Reuse of function declarations

Included files can also contain function interface declarations. This is especially useful for external function libraries, because you can declare all library functions in one file, and include the file in each template that uses the library.

## Named Parameters or Positional Parameters

You can use either named parameters or positional parameters.

If you use positional parameters (for example, omit `~language=` before you specify the language value), the following order is assumed:

- `~name` is the first unnamed parameter from the right
- `~service` is the first unnamed parameter from the left
- `~theme` is the second unnamed parameter from the left
- `~language` is the third unnamed parameter from the left

To include code from the same location as the including template:

```
`include ("purchasing_titlebar")`
```

To include code from the global template directory (using the same theme):

```
`include ("global", "purchasing_titlebar")`
```

To include code used for external function declarations:

```
`include ("system", "", "sapxjstring.html")`
```

## Compile Time Evaluation and Runtime Evaluation

The Internet Transaction Server (ITS) resolves `include` statements at compile time or at runtime.

If the parameters evaluate to constants, or are implicitly defined by the current context, the ITS can resolve them at compile time. Since there are performance advantages to compile time evaluation, you should prefer this option whenever possible.

Examples of include statements resolved at compile time are:

- `` `include ("purchasing_titlebar.html")` ``

  Here, the service, theme and language are defined by the including template itself.

- `` `include ("purchasing"&"_titlebar"&".html")` ``

  Here, the service, theme and language are implicitly defined, and the expression `"purchasing"&"_titlebar"&".html"` evaluates to a constant.

- `` `include ("system", "", "sapxjstring.html")` ``

  Here, all parameters get constant values explicitly assigned.

Examples of include statements that can only be resolved at runtime are:

- `` `include ("purchasing" & screen_element & ".html")` ``

  Here, `screen_element` is a non-constant that can only be defined at runtime.

- `` `include (~service=~service, "purchasing_titlebar.html")` ``

When evaluating expressions, the HTML$^{Business}$ interpreter does not differentiate between variables defined at compile time and variables defined at runtime. For this reason, **`~service`** is not considered a constant expression. However, if you omit the **`~service`** parameter completely, it is handled as a constant and implicitly defined.

## Including Language Resources With Included Templates

If you include a template from a different service in your current service, any language resources associated with the included template are also taken into account.

- If an included template references a language resource file called **`<resource file>_<language>.htrc`**, the HTML$^{Business}$ interpreter attempts to resolve the name from within the included template.
- If an included template references a language resource file called **`<resource file>_<language>.htrc`**, and the name cannot be resolved from within the included template, the HTML$^{Business}$ interpreter must attempt to resolve the name from within the including template.

Suppose the template **`templateA_html`** of service A includes the template **`templateB_html`** of service B:

- If the template **`templateB_html`** of service B references a language resource file called **`resource1_en.htrc`**, the HTML$^{Business}$ interpreter should resolve the name from within the template **`templateB.html`** of service B.
- If the name of the language resource file cannot be resolved from within the included template **`templateB.html`**, it must be resolved from within the including **`templateA.html`** of service A.

A compile time include example could be written as:

```
`include(~service="paw1",~theme="99",
~name="sapmpw01_100")`
```

Suppose **`templateA.html`** of service A includes **`templateB.html`** of service B and **`templateC.html`** of service C:

**`templateA.html`** contains the following HTML$^{Business}$ code:

```
`write (#resource1)`
`include(~service="ServiceB", ~theme="99", ~name= "B" )`
`include (~service="ServiceC", ~theme="99",~name="C")`
```

**`templateB.html`** contains the following HTML$^{Business}$ code:

```
`write (#resource1)`
`write (#resource2)`
```

**`templateC.html`** contains the following HTML$^{Business}$ code:

```
`write (#resource1)`
`write (#resource2)`
```

The contents of the respective language resource files are as follows:

| Resource File | Variable | Value |
|---|---|---|

| | | |
|---|---|---|
| templateA_en.htrc | #resource1 | A1 |
| | #resource2 | A2 |
| templateB_en.htrc | #resource1 | B1 |
| templateC_en.htrc | #resource1 | C1 |

If the logon language is English (**en**), and you execute **templateA.html**, you get the following output :

```
A1
B1
A2
C1
A2
```

A runtime include example could be written as:

```
mytheme=99`
`include(~service="paw1",~theme=mytheme,~name="sapmpw01_100
")`
```

Suppose **templateA.html** of service A includes **templateB.html** of service B, and **templateB.html** of service B includes **templateC.html** of service C:

**templateA.html** contains the following HTML$^{Business}$ code:

```
`write (#resource1)`
`mytheme=99`
`include (~service="ServiceB", ~theme=mytheme,~name="B" )`
```

**templateB.html** contains the following HTML$^{Business}$ code:

```
`mytheme=99`
`include (~service="ServiceC", ~theme=mytheme, ~name="C")`
`write (#resource1)`
`write (#resource2)`
```

**templateC.html** contains the following HTML$^{Business}$ code:

```
`write (#resource1)`
`write (#resource2)`
```

The contents of the respective language resource files are again as follows:

| Resource File | Variable | Value |
|---|---|---|
| templateA_en.htrc | #resource1 | A1 |
| | #resource2 | A2 |
| templateB_en.htrc | #resource1 | B1 |
| templateC_en.htrc | #resource1 | C1 |

If the logon language is English (**en**), and you execute **templateA.html**, you get the following output:

```
A1
C1
A2
```

```
B1
A2
```

## Restrictions

- You cannot use the `include` statement in a language resource file.
- The file name you include must have the extension `.html`.

## repeat Statement

The **repeat** and **for** statements allow you to perform repeated substitution in HTML$^{Business}$ expressions. They force the Internet Transaction Server (ITS) to perform a substitution multiple times. These statements allow you to include R/3 step loops in HTML pages and structure them as you like.

The following table shows some example uses:

| **repeat** | **for** |
|---|---|
| ``` `repeat 10 times` <p></p> `end` ``` | ``` `for(i=1;i<=10;i++)` <p></p> `end` ``` |
| ``` `repeat with i in FIELD_OP` <option value="`i`"> `end` ``` | (No equivalent **for** statement.) |
| ``` `repeat with i from 1 to FIELD_OP.dim` <option value="`FIELD_OP[i]`"> `FIELD_BZ[i]` `end` ``` | ``` `for(i=1; i<=FIELD_OP.dim; i++)` <option value="`FIELD_OP[i]`"> `FIELD_BZ[i]` `end` ``` |

For details on syntax notation in the table, see:

Syntax Conventions [Page 14]

For further information on these statements, see:

repeat [Page 78]

repeat with <reg> in <field> [Page 79]

repeat with <reg> from <expn> to <expn> [Page 80]

For information on the **for** statement, see:

for Statement [Page 69]

# repeat

## Purpose

Repeats a substitution a given number of times.

## Syntax

**repeat** expression **times** statement **end**

## Description

If a certain area in the HTML page is to be repeated several times, and the loop index is not involved, this can be done simply by using this statement.

For information on how HTML Business evaluates expressions, see Expressions [Page 29].

```
`repeat 10 * c times` <td> `end`
```

# repeat with <reg> in <field>

## Purpose

Repeats a substitution for each value in a array.

## Syntax

**`repeat with <register> in <field statement> end`**

## Description

If all values in a multiple value field (array) are displayed in a list, the loop construction is the obvious choice. The **`repeat with <register> in`** variant places all current values of the multi-value field in the register in succession.

For information on how HTML<sup>Business</sup> evaluates expressions, see Expressions [Page 29].

```
<ol>
  `repeat with value in array`
    <li>`value`</li>
  `end`
</ol>
```

# repeat with <reg> from <expn> to <expn>

## Purpose

Repeats a substitution using a loop index.

## Syntax

```
repeat with <register> from <expression> to <expression> [by
<expression>] statement end
```

## Description

With repeat with <reg> in <field> [Page 79], iteration is only possible over a single column of a step loop.

If you want to iterate over several columns in parallel, the individual field values should be activated using an index. To get this running index, use the **`repeat with... from`** variant. This variant sets the register to the **`from`** value and iterates over all values up to the **`to`** value. Using **`by`**, you can specify an increment other than 1.

For information on how HTML$^{Business}$ evaluates expressions, see Expressions [Page 29].

```
<table>
  `repeat with index from 1 to xlist-posnr.dim`
      <tr> <td> `xlist-posnr[i]`  </td>
          <td> `xlist-matnr[i]`  </td>
          <td> `xlist-arktx[i]`  </td>
          <td> `xlist-kwmeng[i]` </td> </tr>
  `end`
</table>
```

To output the above table in reverse sequence, do the following:

```
<table>
  `repeat with index from xlist-posnr.dim to 1 by "-1" `
      <tr> <td> `xlist-posnr[i]`  </td>
          <td> `xlist-matnr[i]`  </td>
          <td> `xlist-arktx[i]`  </td>
          <td> `xlist-kwmeng[i]` </td> </tr>
  `end`
</table>
```

# HTML$^{Business}$ Grammar Summary

The following table summarizes the correct formulation of HTML$^{Business}$ expressions:

| Nonterminal | Derivation |
|---|---|
| htmlbusiness | ([ html \| script[;] ] htmlbusiness)\| **eof** |
| html | **bytestream** |
| script | (declaration \|<br>expression \|<br>conditional \|<br>loop) |
| declaration | **declare** externalfn {, externalfn } **in** module |
| externalfn | identifier |
| module | constant |
| function | funcname **(** argument {**,** argument}) |
| argument | [identifier **=**] expression |
| expression | simpleexpr [compop simpleexpr] |
| simpleexpr | term { addopr  simpleexpr} |
| term | factor { mulopr factor} |
| factor | (**!** \| **++** \| **--**) factor<br>**(** expression) \|<br>assignment \|<br>lvalue [**++** \| **--**] \|<br>constant |
| function call | internalfn **(** argument {**,** argument}) \|<br>externalfn **(** expression {**,** expression}) |
| internalfn | **write \| writeEnc \| wgateURL \| archiveURL \| imageURL \| mimeURL \| assert** |
| mulopr | *** / % &&** |
| addopr | **+ - & \|\|** |
| compop | **== \| != \| > \| < \| >= \| <=** |
| lvalue | field \| register |
| field | { **^** } identifier [ **[** expression **]** ] [. attribute ] |

| | |
|---|---|
| attribute | **label** \| <br> **visSize** \| <br> **maxSize** \| <br> **dim** \| <br> **disabled** \| <br> **name** \| <br> **value** |
| assignment | lvalue **=** expression |
| conditional | **if (** expression**)** htmlbusiness <br> { (**elsif** \| **elseif**) **(** expression) htmlbusiness } <br> [**else** htmlbusiness ] |
| loop | **repeat** expression **times** htmlbusiness **end** \| <br> **repeat with** register **in** field htmlbusiness **end** \| <br> **repeat with** register **from** expression **to** expression **by** expression htmlbusiness **end** \| <br> **for (** expression **;** expression **;** expression**)** htmlbusiness **end** |
| register | identifier |
| identifier | { **~** \| _ \| - } char { char \| **digit** \| _ \| **~** \| **-** } \| |
| constant | **digit** {**digit**} \| <br> **" bytestream "** <br> **#**identifier |
| char | **a..z** \| **A..Z** |

For information on notation used in the table, see:

Syntax Conventions [Page 14]

## External Factors

When using HTML$^{Business}$, there are certain external factors you need to take into account:

Language Independence [Page 84]

Browser Independence [Page 89]

Clientside Caching [Page 92]

Java [Page 94]

## Language Independence

There are three ways of generating language-independent HTML templates:

Getting Texts from the R/3 System [Page 85]

Using Language-Specific Templates [Page 86]

Using Language Resource Files [Page 87]

## Getting Texts from the R/3 System

The easiest way to retrieve texts for an HTML page is to take advantage of language information in the R/3 System. When you log on to the R/3 System, language-specific texts are displayed on the screens. You can evaluate these texts in an HTML template and include them in the resulting HTML page by using the.`label` attribute.

```
<table>
  <tr> <th> `text-posnr.label`  </th>
       <th> `text-matnr.label`  </th>
       <th> `text-arktx.label`  </th>
       <th> `text-kwmeng.label` </th> </tr>
  `repeat with index from 1 to xlist-posnr.dim`
      <tr> <td> `xlist-posnr[i]`  </td>
         <td> `xlist-matnr[i]`  </td>
         <td> `xlist-arktx[i]`  </td>
         <td> `xlist-kwmeng[i]` </td> </tr>
  `end`
</table>
```

The above example is adequate only if all the texts fit on the HTML page. If this is not the case, use one of the methods described in:

Using Language-Specific Templates [Page 86]

Using Language Resource Files [Page 87]

## Using Language-Specific Templates

With language-specific templates, there is a separate set of templates for each language supported. The names of these templates have an language indicator that matches the logon language at runtime.

The naming convention for language-specific HTML templates is:

```
<module pool>_<screen number>_<language>.html
```

For example, `SAPLEC30_1000_D.html`.

The HTML templates for a service are stored in a directory with the same name as the service:

```
…\2.0\<virtual-ITS>\Templates\Service
```

For example, `C:\Program Files\SAP\ITS\2.0\<virtual-ITS>\Templates\ECS3`.

A typical template directory for the languages German and English might contain:

```
saplec30_1000_d.html
saplec30_1000_e.html
saplec30_2000_d.html
saplec30_2000_e.html
```

You can store different sets of HTML templates for the same R/3 transaction, and avoid having to define a separate transaction for each variant. For example, you could create HTML templates in one design, other templates in a second design, and so forth.

It is sufficient to define a separate service for each variant, for example, ECS3_SAP and ECS3_UpToDate. Each service then calls the same transaction (`~transaction` statement in the `<service>.srvc` file). Since each service has its own template directory, you can store different HTML template variants.

The disadvantage of the variant scheme is that HTML pages are stored several times. If structural changes are made to the HTML templates, the different language variants must be adapted one by one.

# Using Language Resource Files

If you want to use a single set of HTML templates for all supported languages, you can use language resource files.

In this case, you create a separate language resource file for each language. Language resource files are stored with the HTML templates, and named according to the following convention:

```
<service>_<language>.htrc
```

For example, `ECS3_EN.htrc`

The language resource files for a service must be stored in the same directory as the HTML templates:

```
…\2.0\<virtual-ITS>\Templates\Service
```

For example, `C:\Program Files\SAP\ITS\2.0\<virtual-ITS>\Templates\ECS3`.

If you use language resource files, the naming conventions for HTML templates are slightly different. In this case, you must omit the language indicator from the file name, otherwise the HTML templates are not recognized. The naming convention is thus:

```
<module pool>_<screen number>.html
```

Instead of using the language-specific templates

```
saplec30_1000_d.html
saplec30_1000_e.html
saplec30_2000_d.html
saplec30_2000_e.html
```

you could implement just two templates and two language resource files:

```
saplec30_1000.html
saplec30_2000.html
ecs3_d.htrc
ecs3_e.htrc
```

## Language Resource File Structure

A language resource file consists of a number of resource keys, each with a name and value. The name is a placeholder string, and the value is the translation for the key into the given language.

You use the resource key as a placeholder in an HTML template to specify where the translated text (the resource value) should be inserted at runtime.

A language resource file for German could contain a number of keys, as shown in the following table:

| Key | Text |
|---|---|
| Title | Statusabfrage fuer Bestellungen [Ask for order status] |
| Ok | OK |
| Cancel | Abbrechen [Cancel] |
| Exit | Beenden [Exit] |
| 4711 | Sind Sie sicher? [Are you sure?] |
| #4712 | Diesen Key gibt es nicht! [This key does not exist!] |
| frontcolor | 0x000000 |
| backcolor | 0xffffff |

## Using Resource Keys

In an HTML template, you can use resource keys wherever an HTML$^{\text{Business}}$ constant can be used. Each resource key must be preceded by the hash symbol:

```
<html>
  <head>
    <title>`#title`</title>
  </head>
  <body bgcolor="`#backcolor`" text="`#frontcolor`">
    <form action="`wgateURL()`" method="post">
      …
      <input type=submit name="~OkCode=/00"  value=" `#ok` ">
      <input type=submit name="~OkCode=/NEX" value=" `#exit`">
    </form>
  </body>
</html>
```

## Browser Independence

To be able to use the same HTML templates in different browsers, you need to specify URLs using the **`wgateURL`** function.

For instance, Netscape Communicator and Microsoft Internet Explorer behave in different ways with respect to the HTTP header "Window-Target":

- When using Netscape Communicator, you can set the destination frame of a request dynamically from the server.
- Microsoft Internet Explorer ignores this entry. A destination frame can only be defined statically using the `<form>` or `<a>` tag, but this coding is not sufficient in all cases.

To handle these differences, the **`wgateURL`** function selects the appropriate variant for the browser making the request (client sensing).

When implementing browser-independent HTML templates, you must decide which requests are to be targetted on which frames, and specify the relevant destination frames to **`wgateURL`** in the `~`**`target`** parameter.

This can be explained in the following examples:

Requests That Change Only One Frame [Page 90]

Requests That Change Multiple Frames [Page 91]

# Requests That Change Only One Frame

If a request triggered by a hyperlink or a submit button (**get** or **post**) changes only one frame, the **~target** parameter must be set to the name of this frame.

Two frames are used in service ECS3.

The left frame shows a selection of purchase requisitions, each one in the form of a hyperlink. When the user selects a hyperlink, the right frame is changed and becomes `Subscreen_210`. It then displays the items belonging to the chosen purchase requisition.

The hyperlink in the left frame should be expressed as follows:

```
<ul>
`for (j=1; j <= xorder-vbeln.dim; j++)`
    <li> <a href="`wgateURL(xlist-vbeln=xorder-vbeln[i],
              ~OkCode="SLCT",
              ~target="subscreen_210")`"> `xorder-vbeln[i]`
`end`
</ul>
```

This construction is triggered in a way that suits the individual browser.

Use the same procedure for URLs in <**form**> tags.

The values of the href attribute in the <**a**> tag and the action attribute in the <**form**> tag must be enclosed in double quotation marks.

You cannot specify a target attribute explicitly.

# Requests That Change Multiple Frames

Although the Internet allows transmission of only one HTML page at a time for a particular request, a user action on one page often requires the contents of several frames to be refreshed dynamically at the same time.

To achieve this, you send the current frameset document back to the browser. The frameset document then requests all the relevant frame pages from the server. In this way you can refresh either one frame, or all frames at once.

If you do not know how many frames need to be refreshed in a transaction as a result of a user action, or if you definitely need to refresh more than one frame, you must request all frames again from the server.

To implement this behavior in an HTML page independently from the browser, the **~target** parameter should be set to **_parent** or **_top**. The server then resubmits the frameset document to the frame superior to the one making the request (**_parent**), or to the topmost frame (**_top**).

Three frames are used in the service CreateSO - a purchasing application.

One frame shows the product details and has a submit button which allows the currently displayed product to be added to a basket of goods displayed in another frame. When the product has been added to the basket of goods, the product overview is again displayed in the product detail frame. Therefore, more than one frame must be renewed because of an action/a request.

The HTML code in the product detail page is as follows:

```
…
<form action="`wgateURL(~target = "_parent")`" method="post">
  …
  …
  <input type=submit name="~OkCode="SLCT" value=" Insert ">
</form>
```

## Clientside Caching

Depending on the browser settings, you can store Web pages in the browser's cache. To a limited extent, this caching behavior can be influenced by users, so you must assume that the user has configured a given behavior.

To handle this, you need to influence the browser's caching behavior from the server side to ensure that when the user performs an action in the browser, a requested page is not taken from the browser's local cache but is displayed after communication with the R/3 System.

Undesired caching causes problems when the request method **`get`** is used. This method has the semantics of a pure read operation and is used with hyperlinks. It is not absolutely necessary to communicate with the server if the browser can fall back on a result which exists for an earlier request. The worst case scenario is that the server is not activated by the user action.

Suppose the service ECS3 is to be activated via the hyperlink

**`"http://pn0208:1080/scripts/wgate.dll?~Service=ECS3"`**

The first time this hyperlink is selected, the request from the browser is sent to the server, which returns the relevant page and starts the appropriate service. The browser stores this page in the local cache. If the configuration is appropriate, the next time the user selects this hyperlink, the browser will load this page directly from the cache without communicating with the server. This means that no service is started on the server side. When the user has finished with the cached page and sends it to the server, the server responds to the request with the error message

`"Session not found!"`

This response is correct, since no service was started.

If you use a form (**`<form>`** tag) and a request is sent to the server via the **`post`** method, this problem does not arise. The **`post`** method has update semantics, causing the browser to communicate with the server when a request is made.

In Web transactions, you must use **`post`** to trigger actions with updates. You cannot use hyperlinks.

### Using an Expiration Date

To be able to use both request methods equally, pages given to the browser by the server should be allocated an expiration date according to their type. This allows you to control when the browser should consider a page to be out-of-date and no longer access it in the cache. When such a page is activated, the browser will re-establish contact with the server.

Use the HTTP header "**`Expires`**" to define the expiration date and give it the value "0" to instruct the browser not to cache this page.

This HTTP header is only used for pages which are addressed using the "`get`" method, for example, via a hyperlink. Pages which are read by "`post`" are not given an expiration date.

To ensure that automatic cache control functions correctly, the caching behavior of HTML templates must not be influenced by subsequently adding <**`meta http-equiv`**…> tags.

## Frame Set Documents

To start a service indirectly via a frame set document you should assign an expiration date to this page statically. This guarantees that the service will actually be started.

The following example shows an indirect service start using a static frame set page:

```
<a href="http://pn0208:1080/ecs1/index.html"> Service ECS1
</a>
```

Contents of index.html:

```
<html><head><title> Order Status </TITLE>
      <meta http-equiv="Expires" content="0">  <!--
IMPORTANT!! -->
      </head>
  <frameset rows=90,*>
    <frame name="PRISMA" src="/ecs1/uptodate.html"
scrolling="no" >
    <frame name="R3" src="/scripts/wgate.dll?~Service=ECS1">
  </frameset>
</html>
```

If the initial pages are static and they are activated using the **`get`** method, the tag **`<meta http-equiv="Expires" content="0">`** must be specified in the <**`head`**> area of the HTML page. Please note the uppercase and lowercase characters in "**`Expires`**".

# Using Java Applets

For display purposes, you can integrate Java applets and ActiveX controls in HTML templates.

The following example shows how to call a Java applet which displays a pie chart:

```
<applet code="PieChart.class" codebase="/ECS3" width=350 height=90>
  <param name=columns   value="3">
  <param name=scale     value="1">
  <param name=bgcolor   value="white">
  <param name=c1_color  value="red">
  <param name=c2_color  value="blue">
  <param name=c3_color  value="green">
  <param name=c1_label  value="Unconfirmed and Undelivered
           `write(100 * (ordered - confirmed) / ordered)`%">
  <param name=c2_label  value="Confirmed but Undelivered
           `write(100 * (confirmed - delivered) / ordered)`%">
  <param name=c3_label  value="Delivered
           `write(100 * delivered / ordered)`%">
  <param name=c1_value  value="`write(100 * (ordered -
confirmed)
            / ordered)`">
  <param name=c2_value  value="`write(100 * (confirmed -
            delivered) / ordered)`">
  <param name=c3_value  value="`write(100 * delivered /
            ordered)`">
</applet>
```

In this example, the code base of the applet is set to `"/ECS3"`. In the case of the Internet Information Server, `"PieChart.class"` must be located in the `"inetsrv/wwwroot/ECS3"` directory.

You can integrate ActiveX controls in the same way.

## Mapping Internet Input Onto the R/3 System

When submitting data from a Web browser to the R/3 System, the concept of name/value pairs supported by the Internet is insufficient. Therefore, the Internet Transaction Server (ITS) uses extended complex syntax to map input fields from the Internet to R/3. This involves the use of angle brackets to allow for step loop or array data input.

The parameters submitted by the Web browser still use the name/value format and they are still URL-encoded (that is, they have the **`name=value&name=value&`**… format. The only difference is that **`name`** can be something like **`mail-text:80[]`**.

Five main cases of input control must be supported:

- Single-value input

  In this case, no changes to the **`name=value`** syntax are necessary. The **`name`** part is mapped onto the corresponding R/3 screen field without any changes. A typical example is the entry field **`firstName`**.

- Multi-value input

  This allows you to fill step loops on screens with input from the Internet. The number of the step loop row is specified together with the name.

- Limitations on input length

  Sometimes, it is necessary to limit the number of characters submitted to the R/3 screen field or to wrap input automatically, e.g. for <**`textarea`**> input controls.

- Submission of several input elements from a single Internet input control
- Support of image maps

For further information, see:

Syntax and Semantics [Page 96]

Passing Multiple Fields From HTML Controls [Page 99]

Using <textarea> Controls [Page 100]

# Syntax and Semantics

The following table summarizes the syntax used to describe all cases of Internet input mapping.

| Nonterminal | Derivation |
|---|---|
| name/value | Field name [ length] [index] "=" value [","  name/value] |
| field name | { "~" \|" _" \| "-" } char { char \| digit \| "_" \| "~" \| "-" } |
| length | ":" numconst |
| Index | "[" numconst "]" |
| value | URL-encoded string |
| numconst | digit { digit } |
| digit | "0".. "9" |
| char | "A".. "Z" \| "a".. "z" |

Please note that angle brackets indicate an optional attribute.

The following table contains the corresponding semantics:

| | |
|---|---|
| length | Limits the number of characters taken from the value part and placed in the corresponding R/3 screen field to numconst characters.<br>Any additional characters are lost, except if the field is a multi-value field, corresponds to a step loop, **and** is filled in append mode (see "index" below). In this case any additional characters are automatically "word-wrapped" to the next step loop row. |
| index | If specified, this makes the field a multiple value field or a step loop. A numconst defines the row to which the given value belongs. No numconst (that is, empty angle brackets) means that the value is added in append mode (that is, appended to the values already set). |

Below are some examples to clarify this:

`firstName=Walt`

sets the field named "`firstName`" on the screen to the value "`Walt`".

`mail-text[12]=Best+regards`

sets the step loop row 12 to the value "`Best regards`". Note that + is the URL-encoding for space.

`mail-text[]="Walt+Whitman"`

appends the text "**`Walt Whitman`**" to the end of the rows currently defined in the field "**`mail-text`**".

If this field already contains

```
mail-text[1] = "Hello Friend"
mail-text[2] = "Hope to see you soon"
mail-text[3] = "Best regards"
```

the result will be

```
mail-text[1] = "Hello Friend"
mail-text[2] = "Hope to see you soon"
mail-text[3] = "Best regards"
mail-text[4] = "Walt Whitman"
```

**`lastName:4=Whitman`**

sets the screen field "**`lastName`**" to the value "**`Whit`**"

**`mail-text:10[2]=Hope+to+see+you+soon`**

sets the step loop row number 2 to the value "**`Hope to see `**" and discards the "**`you soon`**" part.
The line is truncated after the last space if the following word does not fit completely into the given boundaries. Due to the specified index, the "**`you soon`**" part is not wrapped to the next row as shown in the next example.

**`mail-text:10[]=Hope+to+see+you+soon`**

If mail-text currently contains **`mail-text[1]= "Hello! "`**, this will result in:

```
mail-text[1] = "Hello! "
mail-text[2] = "Hope to see "
mail-text[3] = "you soon"
```

The automatic word wrapping takes places because of the unspecified index ("[ ]").

**`mail-text:10[]`**

**`=Hope+to+see+you+soon%d%a%d%aBest+regards%d%aWalt+Whitman`**

If **`mail-text`** currently contains **`mail-text[1]= "Hello! "`**, this will result in:

```
mail-text[1] = "Hello! "
mail-text[2] = "Hope to see "
mail-text[3] = "you soon"
mail-text[4] = ""
mail-text[5] = "Best "
mail-text[6] = "regards"
mail-text[7] = "Walt "
mail-text[8] = "Whitman"
```

The carriage return/new line sequence submitted by <**`textarea`**> controls has the same truncation/word wrap effect as reaching the length limitations on input.

**`firstName=Walt,LastName=Whitman`**

sets the field "**`firstName`**" to the value "**`Walt`**" and the field "**`lastName`**" to the value "**`Whitman`**".

## Passing Multiple Fields From HTML Controls

Suppose you have an R/3 screen with two entry fields, one for the unit of measurement and another one for the amount. On the HTML page, there is just one dropdown list, from which both values must be selected in one step:

```
...
<select name="unit=oz,amount=">
  <option value="1">1 ounce
  <option value="5">5 ounces
  <option value="50">50 ounces
</select>
...
```

You cannot to use the following code to pass two fields at once, because the concatenation of name/value pairs is valid only if applied in the name part of a control:

```
...
<select name="productno=4711,amount=">
  <option value="1,unit=oz">1 ounce
  <option value="5,unit=oz">5 ounces
  <option value="50,unit=l">50 ounces
</select>
...
```

## Using <textarea> Controls

To use a `<textarea>` control to obtain input for an R/3 step loop, you can include the following statement in your HTML template:

```
<textarea name="mail-text:80[]" cols="80"></textarea>
```

This statement ensures that no more than 80 characters per step loop row are placed on the screen.

To place the current content of the step loop on the screen as a default for the <**textarea**> control, use the following code:

```
<textarea name="mail-text:80[]" cols="80">`repeat with r in mail-
text;
write (r, "\r\n"); end`</textarea>
```

Do not include additional line breaks, since this may lead to unintended results in some browsers.

Make sure that there are sufficient step loop rows on your screen to hold the values of the multiple value field. Otherwise an error will occur.