IBM® Workplace Forms™

Version 2.6.1

Introduction to the Viewer Functions

S325-2609-00

Note

Before using this information and the product it supports, read the information in "Notices," on page 25.

First Edition (September 2006)

This edition applies to version 1, release 2.6.1 of Workplace Forms and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces version 1, release 2.6 of Workplace Forms.

© Copyright International Business Machines Corporation 2003, 2006. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Introduction to the Viewer Functions . . 1
Calling a Function in XFDL 1
Document Conventions 1
About Parameters 1
addressBook 2
env 3
fileOpen 4
fileSave 5
getDefaultFilename 6
getHeight 7
getHelpMode 8
getWidth 8
getX 9
getY 10
Header/Footer Functions 11
measureHeight 15
messageBox 17
param 18
setCursor 20
setDefaultFilename 21
setHelpMode 22
showCalendar 23

Appendix. Notices 25
Trademarks 26

© Copyright IBM Corp. 2003, 2006

Introduction to the Viewer Functions

The Viewer functions enable form developers to trigger actions in the Viewer from within a form. The Viewer functions are compiled into a package called *viewer*, similar to the XFDL *system* package. Please refer to the *XFDL Specification* for more details regarding the *system* functions.

Calling a Function in XFDL

The syntax used to call a Viewer function is as follows:

```
viewer.functionName(parameter_1, parameter_2, ... parameter_n)
```

Expression	Description
viewer	The name of the package that the function belongs to.
functionName	The name of the function.
parameter	A string representing the value of the parameter. Functions take zero or more parameters.

All parameters should appear in quotations, as in the following example:

```
viewer.fileOpen("C:\\My Documents","Forms.xfd")
```

Use empty quotes to represent null values.

Document Conventions

Optional parameters in function calls are indicated with brackets ([ ]). For example:

```
fileOpen(startdir, [extfilter])
```

In this function call, the *extfilter* parameter is optional.

About Parameters

In general, parameters are enclosed in single quotes, as shown:

```
function('param1', 'param2')
```

However, in some cases you may want to copy a value from another element in the form. For example, you may want to use the value of a user-set field as the parameter in a function. To do this, you would use a reference to that value with no quotations as a parameter, as shown:

```
function('param1', reference)
```

In this case, the reference will be evaluated, and the value retrieved will be substituted for the reference, resulting in the following:

```
function('param1', 'retrieved value')
```

The function will then be computed.

© Copyright IBM Corp. 2003, 2006

Reference Strings

In some cases, a function may require a *reference string* as a parameter. For example, the second parameter of the *measureHeight* function allows you to specify which item should be measured by providing a reference to that item.

In the normal case, you would provide a reference that is enclosed in quotation marks, as shown:

```
measureHeight('pixels', 'descriptionField')
```

The quotation marks indicate that the function should use the reference as the final value. So in this case, the function will measure the height of the *descriptionField*.

However, if a different element in the form is storing the reference you want to use, you can provide a reference to that element that is not in quotations. For example:

```
getHeight('pixels', storageField.value)
```

In this case, the function will first retrieve the value of the *storageField.value* option, and will use that value to compute the function. For example, if the *value* option of *storageField* contained "descriptionField", then the function would be evaluated as though it was:

```
getHeight('pixels', 'descriptionField')
```

addressBook

Sets the value of one or more form options based on user selection of email addresses. The function opens the user's email client address book and allows them to select email addresses.

This function uses extended MAPI when available, but otherwise uses simple MAPI.

Note: If MS Exchange users have not configured their SMTP addresses, the MS Exchange common name will be returned.

Call

addressBook(*to_field*, *cc_field*, *bcc_field*)

Parameters

Expression	Setting	Description
to_field	**string**	References a form option that is set when a the user selects the To information
cc_field	**string**	References a form option that is set when a the user selects the Cc information (the to_field parameter must be present for this parameter to receive input)
bcc_field	**string**	References a form option that is set when a the user selects the Bcc information (the to_field and cc_field parameters must be present for this parameter to receive input)

Returns

Returns **1** on success and an empty string on failure. Errors are logged and an error form launches on failure.

Example

In the following example, when the user clicks the form's Address Book button, an address book dialog box appears, allowing the user to select email addresses using the To, Cc, or Bcc fields:

```
<button sid="ADDRESS_BOOK">
   <type>select</type>
   <value>Address Book</value>
   <custom:opt xfdl:compute="(toggle(activated, 'off', &#xA;
      'on') == '1' ? viewer.addressbook('TO_FIELD.value', &#xA;
      'CC_FIELD.value', 'BCC_FIELD.value') : '')"></custom:opt>
   <visible compute="(isAvailable('function', &#xA;
      'viewer.addressbook') > '0' ? 'on' : 'off')">on</visible>
   <itemlocation>
      <x>337</x>
      <y>47</y>
   </itemlocation>
</button>
```

env

The *env* function returns a string that contains the details of the environment in which the Viewer is operating. This is useful for determining whether the Viewer is standalone or embedded in a browser, embedded in Eclipse, or embedded in an HTML page.

Call

env()

Parameters

None.

Returns

A string containing:

- **standalone —** If the Viewer is operating as a standalone Viewer.
- **eclipse —** If the Viewer is operating inside Eclipse.
- **browser —** If the Viewer is operating inside a browser.
- **html-object —** If the Viewer is operating inside an HTML page.

Example

The following example uses the *env* function to determine which environment in which the Viewer is operating. The URL of the submit button changes depending upon the Viewer environment. In other words, if the Viewer is operating in standalone mode, it is submitted to server1. If it is operating inside a browser, it is submitted to server2 and so on. As you'll see, two custom options are used, one

containing a compute which calculates the Viewer environment, the other specifying which URL to use for each environment.

```
<button sid = "Submit">
   <value>Submit</value>
   <type>submit</type>
   <custom:enviro xfdl:compute="toggle(global.global.activated, &#xA;
      'off', 'on') == '1' ? viewer.env() : ''"></custom:enviro>
   <custom:envString xfdl:compute="custom:enviro == 'standalone'&#xA;
      ? ('http://server1/cgi-bin/submit') &#xA;
      : custom:enviro == 'browser' &#xA;
      ? ('http://server2/cgi-bin/submit') &#xA;
      : custom:enviro == 'html-object' &#xA;
      ? ('http://server3/cgi-bin/submit') &#xA;
      : custom:enviro == 'eclipse' &#xA;
      ? ('http://server4/cgi-bin/submit') : ''"></custom:envString>
   <url compute="custom:envString"></url>
</button>
```

fileOpen

Displays an **Open File** dialog box and allows the user to select a file. Returns the filename and path of the selected file, but does not actually open the file. This is useful for allowing the user to select a specific file that will be accessed at another time.

For example, the Workplace Forms™ Viewer uses this function in the preferences form to allow the user to set the location of the web browser.

Call

fileOpen(*startdir*, [*extfilter*])

Parameters

Expression	Setting	Description
startdir	**directory path**	Default directory where the file browser will look for files.
extfilters	**string**	A list of one or more strings, which specify the file extensions that can be selected in the Open File dialog. The list must be comma delimited, and the strings should follow this format: <text description> *.<ext> . For example, "HTML document *.html" would represent HTML files. The exact text of the string will appear in the Save File dialog. **Note:** A filter of *.* represents any file type. Also, a list of file extensions, separated by commas, specifies the types of files to display in the dialog box.

Returns

A string containing the path of the file to be opened.

Example

The following example uses the *fileOpen* function to set the URL option in a link button. When the button is selected, an Open File dialog box appears, and the user can select a specific file. The path and filename are returned and used as the URL

option for the link button. Because it is a link button, the selected file is opened by the browser. For more information on link buttons, refer to the *XFDL Specification*.

```
<button sid = "BUTTON1">
   <value>Open File...</value>
   <type>link</type>
   <url compute="toggle(activated, 'off', 'on') == '1' ? &#xA;
      viewer.fileOpen('C:\\My Documents', &#xA;
      'XFDL Document *.xfd, HTML Documents *.htm') : ''"></url>
</button>
```

The parameters in this example specify that the Open File dialog box will default to the My Documents folder, and that it will display both XFDL (.xfd) and HTML (.htm) files.

fileSave

Displays a Save File dialog box and allows the user to select or type a filename. The path and filename selected are returned, but the file is not actually saved. This function is intended for use with other applications, which can perform the actual save action based on the function return value.

For example, a configuration form (like the Viewer's preferences form) might allow the user to set a default location to which all files should be saved. The application in question would then check the preferences form, and save all files in the specified location.

Call

fileSave(*startdir*, [*default_text*, [*extfilter_1*, *extfilter_2*, ... *extfilter_n*]])

Parameters

Expression	Setting	Description
startdir	**directory path**	Default directory where files will be saved.
default_text	**file extension**	Default file extension used when files are saved.
extfilters	**string**	A list of one or more strings, which specify the file extensions that can be selected in the Save File dialog. The list must be comma delimited, and the strings should follow this format: <text description> *.<ext> . For example, "HTML document *.html" would represent HTML files. The exact text of the string will appear in the Save File dialog. Note that a filter of *.* represents any file type.

Returns

A string containing the file path name of the file to be saved.

Example

The following example uses the *fileSave* function to set the value of a label. When the button on the form is clicked, a Save File dialog opens, and the user selects a folder and filename. The *fileSave* function returns the path to this location, and the

set function is used to assign the path to the value option of the label (see the *XFDL Specification* for more information on the *set* function).

```
<label sid = "savelabel">
   <size>
      <width>50</width>
      <height>1</height>
   </size>
   <value></value>
</label>
<button sid = "savebutton">
   <type>select</type>
   <value>Select file to be saved</value>
   <custom:save_opt xfdl:compute="toggle(activated, 'off', &#xA;
      'on') == '1' ? set('savelabel.value', &#xA;
      viewer.fileSave('C:\My Documents', 'frm', 'Forms &#xA;
      *.xfd, HTML Forms .htm,   *.html, *.doc')) : ''"></custom:save_opt>
</button>
```

The parameters in this example specify that the Save File dialog box will default to the My Documents folder, that the filename will default to a ".xfd" extension. The filters specify that XFDL (.xfd), HTML documents (.html and .htm), and Word documents (.doc) are acceptable file extensions.

getDefaultFilename

The Viewer maintains a default filename for all open forms, unless they are temporary files. In general, a form will be considered temporary if it is passed to the Viewer by the web browser. For example, a form passed to the Viewer in response to a web transaction would be a temporary file. Temporary files have no default filenames.

Calling **getDefaultFilename** will return the default filename.

Call

getDefaultFilename()

Parameters

None.

Returns

Returns a string containing the default filename of the form. This string does not include any path information.

Usage Details

You must use the event model to trigger the **getDefaultFilename** function. This means you must use the *toggle* function, *keypress*, *mouseover*, or some other event. If you want **getDefaultFilename** to run when the form opens, *toggle* the function off of the value of the *global.global.activated* option. This option will switch to *on* when the form is opened.

Example

The following example creates a label in the form that displays the default filename when the user clicks the *getFilename* button.

```
<label sid = "filename_LABEL">
   <value compute="toggle(getFilenameButton.activated, &#xA;
      'off', 'on') == '1' ? viewer.getDefaultFilename() :
      value"></value>
</label>
```

getHeight

Measures an item's height in either pixels or characters.

Call

getHeight(*units*, [*item*])

Parameters

Expression	Setting	Description
units	**string**	Determines whether height is returned in **chars** or **pixels**.
item	**reference string**	*Optional*. References the SID of the item you want to measure. If no item is specified, the current item is measured.

Returns

Returns a string containing the height of the current or specified item in either pixels or characters.

Usage Details

You must use the event model to trigger the **getHeight** function. This means you must use the *toggle* function, *keypress*, *mouseover*, or some other event. If you want **getHeight** to run when the form opens, *toggle* the function off of the value of the *global.global.activated* option. This option will switch to *on* when the form is opened.

Example

In the following example, when a user clicks BUTTON1, **getHeight** calculates the height in pixels of FIELD3:

```
<field sid="FIELD3">
   <itemlocation>
      <x>17</x>
      <y>25</y>
      <width>48</width>
      <height>397</height>
   </itemlocation>
   <value compute="toggle(BUTTON1.activated, 'off', 'on') == &#xA;
      '1' ? viewer.getHeight('pixels') : value"></value>
   <scrollhoriz>always</scrollhoriz>
</field>
```

getHelpMode

The Viewer has a help mode that is entered when the user clicks the appropriate icon on the Viewer's toolbar. While the help mode is active, help messages that have been added to the form are displayed for the user as tool tips.

Calling **getHelpMode** will return the Viewer's help mode status: *on* or *off*.

Call

getHelpMode()

Parameters

None.

Returns

Returns either **on** or **off**.

Example

The following example creates a button in the form that will turn the help mode on and off (just like the button in the Viewer's toolbar). When the button is clicked, the form uses **getHelpMode** to determine whether the help mode is currently *on* or *off*. Based on that value, the form uses **setHelpMode** to change the setting of the help mode. So, if the help mode is *on*, the form sets it to *off*, and if the help mode is *off*, the form sets it to *on*.

Note: **getHelpMode** should be used in conjunction with the toggle function (see the XFDL Specification for more information about the toggle function).

```
<button sid="toggleHelp_BUTTON">
   <value>Toggle Help Mode</value>
   <custom:toggle_OPTION xfdl:compute="toggle(activated, &#xA;
      'off', 'on')=='1' ? viewer.getHelpMode()=='on' ? &#xA;
      viewer.setHelpMode("off") : viewer.setHelpMode &#xA;
      ('on') : ''"></custom:toggle_OPTION>
</button>
```

getWidth

Measures an item's width in either pixels or characters.

Call

getWidth(*units*, [*item*])

Parameters

Expression	Setting	Description
units	**string**	Determines whether width is returned in **chars** or **pixels**.
item	**reference string**	*Optional*. References the SID of the item you want to measure. If no item is specified, the current item is measured.

Returns

Returns a string containing the width of the current or specified item in either pixels or characters.

Usage Details

You must use the event model to trigger the **getWidth** function. This means you must use the *toggle* function, *keypress*, *mouseover*, or some other event. If you want **getWidth** to run when the form opens, toggle the function off of the value of the *global.global.activated* option. This option will switch to *on* when the form is opened.

Example

In the following example, when a user selects BUTTON1, **getWidth** calculates the width in pixels of FIELD3:

```
<field sid="FIELD3">
   <itemlocation>
      <x>17</x>
      <y>25</y>
      <width>48</width>
      <height>397</height>
   </itemlocation>
   <value compute="toggle(BUTTON1.activated, 'off', 'on') == &#xA;
      '1' ? viewer.getWidth('pixels') : value"></value>
   <scrollhoriz>always</scrollhoriz>
</field>
```

getX

Calculates the distance from the left edge of the form to the left edge of the item (in other words, the x coordinate of the item) in pixels.

Call

getX([*item*])

Parameters

Expression	Setting	Description
item	reference string	*Optional*. References the SID of the item you want to use. If no item is specified, the current item is used.

Returns

Returns a string containing the x coordinate of the item in pixels.

Usage Details

You must use the event model to trigger the **getX** function. This means you must use the *toggle* function, *keypress*, *mouseover*, or some other event. If you want **getX** to run when the form opens, *toggle* the function off of the value of the *global.global.activated* option. This option will switch to *on* when the form is opened.

Example

In the following example, when a user selects BUTTON1, **getX** calculates the x coordinate of FIELD3:

```
<field sid="FIELD3">
   <itemlocation>
      <x>17</x>
      <y>25</y>
      <width>48</width>
      <height>397</height>
   </itemlocation>
   <value compute="toggle(BUTTON1.activated, 'off', 'on') == &#xA;
      '1' ? viewer.getX() : '' "></value>
   <scrollhoriz>always</scrollhoriz>
</field>
```

getY

Calculates the distance from the top edge of the form to the top edge of the item (in other words, the y coordinate of the item) in pixels.

Call

getY([*item*])

Parameters

Expression	Setting	Description
item	**reference string**	*Optional*. References the SID of the item you want to use. If no item is specified, the current item is used.

Returns

Returns a string containing the y coordinate of the item in pixels.

Usage Details

You must use the event model to trigger the **getY** function. This means you must use the *toggle* function, *keypress*, *mouseover*, or some other event. If you want **getY** to run when the form opens, *toggle* the function off of the value of the *global.global.activated* option. This option will switch to *on* when the form is opened.

Example

In the following example, when a user selects BUTTON1, **getY** calculates the y coordinate of FIELD3:

```
<field sid="FIELD3">
   <itemlocation>
      <x>17</x>
      <y>25</y>
      <width>48</width>
      <height>397</height>
   </itemlocation>
   <value compute="toggle(BUTTON1.activated, 'off', 'on') == &#xA;
      '1' ? viewer.getY() : ''"></value>
   <scrollhoriz>always</scrollhoriz>
</field>
```

Header/Footer Functions

This is a collection of functions that can be used within the *printsettings* option to add header and footer information to a form. These headers and footers do not appear on the screen, but do appear when the form is printed.

Each header and footer can be one or more lines in height. However, they can be no larger than 1/3 of the page size. Each header and footer is also divided into three separate sections - the left, the middle, and the right. By placing text in a particular section, you control where the text is positioned, as follows:

- **Left —** The text begins at the left edge of the form.
- **Middle —** The text is centered in the middle of the form.
- **Right —** The text is positioned so that it ends at the right edge of the form.

Each section can contain different text. For example, you might put a date in the left section, a title in the middle section, and a page number in the right section.

If you place a long string of text in a header or footer, it will overlap the other sections of that header or footer. For example, suppose you put the following text in the left section of your header:

```
This form is for demonstration purposes only. Do not distribute.
```

This text would start at the left edge of the form, but would continue to overlap the middle portion of the header. Futhermore, a longer string would also overlap the right portion of the header.

Any hard returns placed in a string are respected. For example, you could avoid overlapping the other sections of the header by using the same string with hard returns, as shown:

```
This form is for
demonstration purposes
only. Do not distribute.
```

If a string is wider than the form, it is truncated appropriately. For example, a string that starts on the left edge of the form is truncated once it reaches the right edge of the form, and vice versa. If a string starts in the middle of the form, it is truncated on both the left and right edges.

Setting the PrintSettings Option

When using the Header/Footer functions, you must include two additional arrays in the *printsettings* option for the form. The *printsettings* option should be configured as follows:

```
<printsettings>
   <pages>page list</pages>
   <dialog>dialog settings</dialog>
   <header>header information</header>
   <footer>footer information</footer>
</printsettings>
```

The header and footer information are themselves arrays, and should look like this:

```
<header>
   <left>left text</left>
   <center>center text</center>
   <right>right text</right>
</header>
```

All text can be set as normal, using strings, computes, or functions to determine what the text should be.

For more information on configuring the page list and dialog settings, refer to the *XFDL Specification*.

Pages vs. Sheets

Forms often contain multiple pages. These pages are just like the pages of a paper form - you complete one page at a time, and ″flip″ between the pages (usually with a next or previous page button) while completing the form.

However, when a form is printed, sometimes a single page of the form will be too large to fit on one piece of paper. Since there is no limit to the space you can take up on the computer screen, some form pages may in fact cover many pieces of paper when printed. To make the distinction between a ″form page″ and the ″number of pieces of paper″ more clear, we call the pieces of paper ″sheets.″ So, if the first page of a form prints on three pieces of paper, we say that the page covers three sheets.

This distinction is important for numbering, since the Header/Footer functions allow you to number both pages and sheets when printing your form.

Calls

The Header/Footer functions are a series of calls that return text-based information. These function calls are listed and described below.

Function	Description
printTime()	Prints the current time from your computer’s system clock. The time is formatted using a 12-hour clock as HH:MM AM/PM. For example, 3:00 PM.
printDate()	Prints the current date from your computer’s system clock. Dates are formatted as MMM DD YYYY. For example, Aug 24 1999.
printFormPage()	Prints the page number of the page currently being printed. Numbering begins from the first page of the form, and includes all pages regardless of whether they are being printed. For example, if the third page of the form was being printed, the number 3 would be printed on the third page. This would be true even if page 1 was not printed. PAGE 1 PAGE 2 PAGE 3 X 2 2 3 SHEET 1 SHEET 2 Notice that pages with multiple sheets will have the same number on each sheet.

Function	Description
printActualFormPage()	Prints the page number of the page currently being printed. Numbering begins from the first page being printed, and does not include pages not being printed. For example, if the third page of the form was being printed, but pages 1-2 were not, the number 1 would be printed on the third page. PAGE 1 PAGE 2 PAGE 3 X X X 1 SHEET 1 SHEET 2 Pages with multiple sheets will have the same number on each sheet.
printTotalFormPages()	Prints the total number of pages in the form. Numbering begins from the first page of the form, and includes all pages regardless of whether they are being printed. For example, if a form that contained 3 pages was being printed, then the total number of pages would be 3. This would be true even if page 1 was not being printed. PAGE 1 PAGE 2 PAGE 3 X 3 3 3 SHEET 1 SHEET 2 Note that the total page count is not affected by the number of sheets any page may print on. For example, if a three page form prints on four sheets, the page count is still three.
printActualTotal FormPages()	Prints the total number of pages being printed. Numbering begins with the first page being printed, and does not include pages not being printed. For example, if a form that contained 3 pages was being printed, but page 1 was not being printed, then the total number of pages would be 2. PAGE 1 PAGE 2 PAGE 3 X 2 2 2 SHEET 1 SHEET 2 Note that the total page count is not affected by the number of sheets any page may print on. For example, if two pages of the form printed on three sheets, the page count would still be two.

Function	Description
printSheet()	Prints the sheet number of the form currently being printed. Each piece of paper used in printing is one ″sheet,″ and numbering begins with the first piece of paper used to print the form. For example, if a form printed page one on one sheet of paper, page two on two sheets of paper, and page three on one sheet of paper, the sheets would be numbered 1 through 4. PAGE 1 PAGE 2 PAGE 3 1 2 3 4 SHEET 1 SHEET 2 If a page is not printed, the sheet count does not include that page. For example, if page one in the example above were not printed, the other sheets would be numbered 1, 2, and 3 respectively.
printPageSheet()	Prints the sheet number of the page currently being printed. Each piece of paper used in printing is one ″sheet.″ In this case, numbering begins with the first sheet used to print the current page. For example, if a form printed page two on two pieces of paper, then the numbers of those pieces of paper would be 1 and 2 respectively. PAGE 1 PAGE 2 PAGE 3 1 1 2 1 SHEET 1 SHEET 2
printTotalPage Sheets()	Prints the total number of sheets necessary to print the current page. Each piece of paper used in printing is one ″sheet.″ In this case, numbering begins with the first sheet used to print the current page. For example, if a form printed page two on two pieces of paper, then the number 2 would be printed on both sheets. PAGE 1 PAGE 2 PAGE 3 1 2 2 1 SHEET 1 SHEET 2

Parameters

None of these functions take parameters.

Returns

Each function returns a number or a string, depending on the specific function. See the ″Call″ section above for more information.

Example

The following example uses the *printdate* and *printtime* functions to add date and time information to the header of the form. The *printformpage* and *printsheet* functions are also used to number the pages and sheets in the footer.

```
<printsettings>
   <pages>page list</pages>
   <dialog>dialog settings</dialog>
   <header>
      <left>DRAFT</left>
      <center compute="'Printed on ' +. viewer.printDate() &#xA;
         +. 'at' +. viewer.printTime()"></center>
      <right>DRAFT</right>
   </header>
   <footer>
      <center compute="'Page Number ' +. viewer.printFormPage()"
         ></center>
      <right compute="'Sheet Number ' +. viewer.printSheet()"></right>
   </footer>
</printsettings>
```

On the printed form, the header and footer would look like this:

Header

DRAFT   Printed on Aug 24 1999 at 3:00 PM   DRAFT

Footer

Page Number 1   Sheet Number 1

measureHeight

This function calculates how tall an item would have to be to display all of its text. This calculation is based on the current width of the item. For example, when using a monospace font, if a field with a width of 60 characters contained 150 characters of text, the field would have to be 3 lines tall to display all of the text.

This function should be used as part of a *size* or *itemlocation* option, and allows items to be dynamically sized based on the amount of text in those items. The **measureHeight** function must be used in conjunction with the *toggle* function (see the *XFDL Specification* for more information on the *toggle* function). When the user moves out of a field that is sized by the **measureHeight** function, the field's height will be updated automatically.

Measuring Height in Pixels

Pixel values for height should be used to set the third element of the ″extent″ array in the *itemlocation* option. For example:

```
<itemlocation>
   <x>10</x>
   <y>10</y>
   <width>300</width>
```

```
      <height compute="(toggle(value) == '1') and &#xA;
         (viewer.measureHeight('pixels') > '22') ? &#xA;
         viewer.measureHeight('pixels') : '22'"></height>
   </itemlocation>
```

Using the itemlocation option to set the height of the item allows an exact fit, regardless of the font or font size of the text displayed.

Measuring Height in Characters

Character values for height should be used to set the second element of the size option. For example:

```
<size>
   <width>60</width>
   <height compute="(toggle(value) == '1') ? &#xA;
      viewer.measureHeight('chars') : '1'"></height>
</size>
```

Using the size option to set the height of an item does not always allow for an exact fit. In some cases the item may be slightly larger than the text displayed, depending on the font and font size used.

Call

measureHeight(*units*, [*item*])

Parameters

Expression	Setting	Description
units	**string**	Either **chars** or **pixels**, depending on which unit you want to use for measurement.
item	**reference string**	*Optional*. A reference to the item you want to measure. If no item is specified, the item containing the **measureHeight** function will be measured.
scrollvert	**string**	*Optional*. Determines whether a scollbar should be taken into consideration when measuring the height of a field. Used only if you have a dynamically-added vertical scrollbar that only appears when the field has the focus. If this parameter is used, its setting must be **always**.

Returns

Returns a number representing the height in lines or pixels.

Usage Details

1. You must use the event model to trigger the **measureHeight** function. This means you must use the *toggle* function, *keypress*, *mouseover*, or some other event. If you want **measureHeight** to run when the form opens, *toggle* the function off of the value of the *global.global.activated* option. This option will switch to *on* when the form is opened.
2. The **measureHeight** function should only be used with *field*, *label*, or *box*.

Example

The following example uses the **measureHeight** function to set the height of a field in characters:

```
<field sid = "description_FIELD">
   <scrollhoriz>wordwrap</scrollhoriz>
   <scrollvert>always</scrollvert>
   <size>
      <width>60</width>
      <height compute="(toggle(value) == '1') ? &#xA;
         viewer.measureHeight('chars') : '1'"></height>
   </size>
   <value></value>
</field>
```

The following example uses the measureHeight function to set the height of a field in pixels:

```
<field sid = "description_FIELD">
   <scrollhoriz>wordwrap</scrollhoriz>
   <scrollvert>always</scrollvert>
   <value></value>
   <itemlocation>
      <x>10</x>
      <y>10</y>
      <width>300</width>
      <height compute="(toggle(value) == '1') and &#xA;
         (viewer.measureHeight('pixels') > '22') ? &#xA;
         viewer.measureHeight('pixels') : '22'"></height>
   </itemlocation>
</field>
```

Note: In this example, the value ’22’ is the height (in pixels) that the field will default to when it is empty. This value must be determined based on the font size used and how many blank lines of text you want to show when the field is empty.

messageBox

This function displays a message box that prompts the user. There are two types of message boxes that can be displayed:

1. An **OK** box, that prompts the user to acknowledge a message by pressing OK.
2. A **QUESTION** box, that prompts the user to answer Yes or No to a question.

In each case, the message box will display a specified title and message. The message box will return a value based on the user’s response.

Call

messageBox(*message*, [*caption*, *messagetype*])

Parameters

Expression	Setting	Description
message	**string**	Contains the message to display in the main portion of the message box.
caption	**string**	Contains the caption to display in the title bar of the message box.

Expression	Setting	Description
messagetype	**message box type**	Specifies whether the message box is an **OK** or a **QUESTION** box. If the type is **OK**, then the box will contain an OK button. If the type is **QUESTION**, then the box will contain a Yes button and a No button. The default type is OK.

Returns

1 if the user selects an **OK** button.

1 if the user selects a **Yes** button.

0 if the user selects a **No** button.

Example

This example uses an action to open a message box when the form opens. The message box asks the user if they want to close the form.

```
<action sid = "cancel_form">
   <custom:message_display xfdl:compute="viewer.messageBox( &#xA;
      'Do you want to close this form?', 'Just Checking', &#xA;
      'QUESTION')"></custom:message_display>
   <active compute="cancel_form.message_display == '1' ? &#xA;
      'on' : 'off'"></active>
   <type>cancel</type>
   <delay>
      <repeat>once</repeat>
      <interval>0</interval>
   </delay>
</action>
```

If the user clicks ″No″, the *messageBox* function returns a value of ″0″. The compute on the active option of the **cancel_form** action then evaluates to **false**, and the active option of **cancel_form** is set to **off**, so the form opens as normal. If the user clicks ″Yes″, the value returned by the function is **1**, and the action’s active option is set to **on**, closing the form.

param

The *param* function allows you to call one of several *name* attributes of the HTML *param* element. It returns the value of the specified *name* attribute’s associated *value* attribute. This function is only valid if the Viewer is embedded in an HTML page.

Call

param(*name*)

Parameters

Expression	Setting	Description
name	**string**	The value of the *name* attribute in the HTML *param* element. They are: • **XFDLID** • **TTL** • **detach_id** • **refresh_URL** • **retain_viewer** • **portlet_URL** • **instance_1... instance_n** For more information regarding these properties, see the "Usage Details" section below.

Returns

The value of the HTML *param* element's *value* attribute.

Usage Details

The HTML *object* element is used to embed XFDL forms inside HTML pages. The HTML *param* element consists of *name* and *value* attributes that have no meaning in HTML. However the *properties* of the *name* and *value* attributes determine the Viewer's behavior when embedded in an HTML page. The Viewer function *param* returns the value of the *param* element's *value* attribute for use in XFDL computes. To do this, it must call the relevant attribute property by name to retrieve its value. These properties are:

- **XFDLID —** Returns the ID of the tag that contains the form information.
- **TTL —** Returns the length of time the detached form will live before being destroyed automatically. Value given in seconds. For example, **60**.
- **detach_id —** Returns the unique ID of the form instance. Used in successive objects to allow the form to be reattached and updated.
- **refresh_URL —** Returns the URL called to reload the *XFDL form* if the *detach_id* has timed out.
- **retain_viewer —** Returns either **off** or **on**, depending on whether the Viewer remains available after completing *replace* or *done* actions. If *retain_viewer* is **off**, the Viewer closes after completing either action. If it is **on**, the Viewer remains available for further use, such as to retain form data after a submission.
- **portlet_URL —** Returns the URL of the portlet.
- **instance_1... instance_n —** Returns information regarding the XML data inside the HTML document that will replace or be appended to a specific XML instance inside the XFDL form. This includes:
  - The ID of the new instance.
  - The ID of the form instance.

    Two additional values may also be returned:
  - Either *replace* or *append*, depending upon whether the new instance data replaces or adds to the original instance data. Note that *replace* is the default value.

– The reference within the instance that indicates where the new data should be placed. Note that any namespaces listed in this value resolve relative to the document root.

Note: You must use the event model to trigger the *param* function. This means you must use the toggle function, keypress, mouseover, or some other event. If you want *param* to run when the form opens, toggle the function **off** of the value of the *global.global.activated* option. This option will switch to **on** when the form is opened.

Example

In the following example, when a user selects BUTTON1, *param* returns the **XFDLID** value to FIELD3:

```
<field sid="FIELD3">
   <itemlocation>
      <x>17</x>
      <y>25</y>
      <width>48</width>
      <height>397</height>
   </itemlocation>
   <value compute="toggle(BUTTON1.activated, 'off', 'on') == &#xA;
      '1' ? viewer.param('XFDLID') : value"></value>
</field>
```

setCursor

The *setCursor* function has two uses:

1. To place the cursor at a specific location in a field.
2. To highlight a specific section of text in a field.

This function is useful when you want the user to start typing after some information that is already in a field, or when you want the user to replace a specific section of text.

Call

setCursor(*startValue*, [*endValue*])

Parameters

Expression	Setting	Description
startValue	**integer**	The start position of the cursor within a field.
endValue	**integer**	The end position of the highlighted text within a field.

Returns

1 if function is successful

0 if errors occur

Usage Details

1. If both parameters have the same value, the cursor is placed at the location indicated (since both parameters indicate the same location).

2. If the *endvalue* is less than *startvalue*, the second parameter is ignored and the cursor is placed at the location indicated by the first parameter.
3. If the *endvalue* is greater than the length of the field, all of the text and white space (such as spaces) in the field, from the location indicated by the first parameter to the end of the field, is highlighted.

Example

In this example, when the user tabs into the field, the word ″shall″ will be highlighted. Note that **setCursor** must be used in conjunction with the *toggle* function (see the *XFDL Specification* for more information about the *toggle* function).

```
<field sid = "FIELD1">
   <label>Set Cursor Field</label>
   <custom:set_cursor xfdl:compute="toggle(focused, 'off', 'on') &#xA;
      == '1' ? viewer.setCursor('6', '10') : ''"></custom:set_cursor>
   <value>What shall we do with the drunken sailor?</value>
</field>
```

setDefaultFilename

The Viewer maintains a default filename for all open forms, unless they are temporary files. In general, a form will be considered temporary if it is passed to the Viewer by the web browser. For example, a form passed to the Viewer in response to a web transaction would be a temporary file. Temporary files have no default filenames.

Calling **setDefaultFilename** changes the default filename to a specified value.

This can be useful if you are using the same form many times, and you want to uniquely identify each copy of the form based on who completed it and when it was filled out. For example, you might create a new filename based on the user name and the date.

Call

setDefaultFilename(*Filename*)

Parameters

Expression	Setting	Description
Filename	**string**	The new default filename for the form. This string should not include path information.

Returns

Nothing.

Usage Details

1. You must use the event model to trigger the **setDefaultFilename** function. This means you must use the *toggle* function, *keypress*, *mouseover*, or some other event. If you want **setDefaultFilename** to run when the form opens, *toggle* the function off of the value of the *global.global.activated* option. This option will switch to *on* when the form is opened.

Example

In this example, **setDefaultFilename** is used when the save button is clicked. Note that this example also uses the *toggle* function (see the *XFDL Specification* for more information about the *toggle* function).

```
<button sid = "save_BUTTON">
   <value>Save</value>
   <custom:filenameSet_Option xfdl:compute="toggle &#xA;
      (save_BUTTON.activated) == '1' ? viewer.setDefaultFilename( &#xA;
      'myform' +. date()) : ''"></custom:filenameSet_Option>
   <type>saveform</type>
</button>
```

When the button is clicked, the default filename is set to be ″myform<*date*>″. For example, if the form was saved on September 13, 1999, the filename would be set to ″**myform19990913**″.

setHelpMode

The Viewer has a help mode that is entered when the user clicks the appropriate icon on the Viewer's toolbar. While the help mode is active, help messages that have been added to the form are displayed for the user as tool tips.

Calling *setHelpMode* will set the Viewer's help mode to either *on* or *off*.

Note: Help mode can only be set to on if the page currently being displayed contains help items. If there is no help available in the current page, then help mode cannot be initialized.

Call

setHelpMode(*helpModeSetting*)

Parameters

Expression	Setting	Description
helpModeSetting	**string**	The status the help mode should be set to. Valid settings are **on** or **off**.

Returns

Nothing.

Example

The following example creates a button that will turn the help mode on when clicked. Note that this example also uses the *toggle* function (see the *XFDL Specification* for more information about the *toggle* function).

```
<button sid = "help_mode_BUTTON">
   <value>Help Mode</value>
   <custom:mode_OPTION xfdl:compute="toggle &#xA;
      (help_mode_BUTTON.activated)== '1' ? &#xA;
      viewer.setHelpMode("on") : ''"></custom:mode_OPTION>
</button>
```

showCalendar

Calling this function displays a calendar widget on the form. The calendar pops up from the item containing the *showCalendar* function, in the same way a list pops up from a popup item. The user can then select a date from the widget, which is returned by the function.

Note: You cannot call *showCalendar* from an action, cell, or spacer item.

Call

showCalendar([*date*], [*formatNode*])

Parameters

Expression	Setting	Description
date	**string**	*Optional*. A date. This sets the default date for the widget, which is the date the widget shows when it first opens. If no date is provided, the widget defaults to the current date. Note that this date is interpreted based on the user preferences set for the Viewer. To ensure that the date is not misinterpreted, use a long format that leaves no room for error, such as: March 24, 2004.
formatNode	**string**	*Optional*. A reference to a *format* option in the form. This option is used to interpret the date in the *date* parameter. This is important for dates that are ambiguous. For example, in 2004 02 02 it isn't obvious which number is the month and which number is the day. If this option is not set, the function uses the date setting in the Viewer preferences to interpret the date. However, if the date that you are provide is not in long format, it is strongly recommended that you provide a *formatNode*.

Returns

The date selected by the user. The date is formatted according to the user preferences set for the Viewer.

Usage Details

You must use the event model to trigger the **showCalendar** function. This means you must use the *toggle* function, *keypress*, *mouseover*, or some other event. If you want **showCalendar** to run when the form opens, *toggle* the function off of the value of the *global.global.activated* option. This option will switch to *on* when the form is opened.

Example

The following example shows a field and a button. The field's value option contains a *showCalendar* function that is triggered when the user clicks the button. In this case, the calendar appears connected to the field, and opens the calendar to the current date. The function then returns the date chosen by the user, which populates the value option.

```
<button sid="calendarButton">
   <value>Set Date</value>
</button>
```

```
<field sid="dateField">
   <value compute="toggle(calendarButton.activated) == '1' ? &#xA;
      viewer.showCalendar() : ''"></value>
</field>
```

The next example again shows a field and a button. In this case, the button contains a custom option with a compute that opens the calendar widget when the user clicks the button. In this case, the calendar appears connected to the button. Notice that the compute also uses a *set* function to set the return value of the *showCalendar* function into the field's value option.

```
<button sid="calendarButton">
   <value>Set Date</value>
   <custom:calendar xfdl:compute="
      toggle(calendarButton.activated) == '1' ? &#xA;
      (set('dateField.value', viewer.showCalendar())) : &#xA;
      ''"></custom:calendar>
</button>
<field sid="dateField">
   <value></value>
</field>
```

By using the *set* function (rather than creating a compute in the field's *value* option that copies the custom option) you ensure that the user can also type a date directly into the field if that is desired.

Appendix. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright IBM Corp. 2003, 2006

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Office 4360
One Rogers Street
Cambridge, MA 02142
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX
IBM
Workplace
Workplace Forms

Other company, product, or service names may be trademarks or service marks of others.

IBM®

Program Number:

Printed in USA

S325-2609-00