IBM® Workplace Forms™

Version 2.6.1

XFDL Specification

S325-2600-00

Note

Before using this information and the product it supports, read the information in “Notices,” on page 433.

First Edition (September 2006)

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

Contents

Introduction

This document describes a class of XML documents called Extensible Forms Description Language (XFDL) Forms and partially describes the behavior of computer programs that process them. An XFDL processor is a software program that reads, processes, and writes XFDL forms. Processing may include such tasks as GUI rendering, data extraction, or modification.

Origin and Goals

From 1993 to 1998, PureEdge (since acquired by IBM®) developed the Universal Forms Description Language (UFDL). XFDL is the result of developing an XML syntax for the UFDL, thereby permitting the expression of powerful, complex forms in a syntax that promotes application interoperability and adherence to worldwide Internet standards. The current design goals of XFDL are to create a high-level computer language that:

1. Represents forms as single objects without dependencies on externally defined entities, thus allowing them to act as contractual documents.
2. Is represented by human-readable plain text.
3. Is a publicly accessible open standard.
4. Provides a syntax for inline mathematical and conditional expressions.
5. Permits the enclosure of an arbitrary size and number of base-64 encoded binary files.
6. Offers precision layout needed to represent and print dense business/government forms.
7. Facilitates server-side processing via client-side input validation and formatting.
8. Permits extensibility including custom items, options, and external code functions.
9. Offers comprehensive signature support, including:
   - Capture of the whole context of a business transaction
   - Multiple signers
   - Different signers of (possibly overlapping) portions of a form
   - Freezing computations on signed portions of a form

Maintaining the data, logic, and presentation layers in a single, legally binding document is a paradigm shift in electronic commerce, which traditionally separates each layer. The decision to use a more ″document-centric″ model was not made lightly - it was found to be necessary in order to provide legal non-repudiation.

The original version of XFDL was published as a W3C Note in 1998. A number of features of XFDL have since been incorporated into a W3C Recommendation called XForms in 2003. XForms defines constructs for a standard XML-based data model, input validation, calculations, constraints, and other properties, server submission characteristics, event-based action sequences, and a user interface vocabulary that includes the ability to hierarchically group and to iterate user interface controls. XForms standardizes the core business processing model of a web application, but it is designed to be incorporated into host languages that provide extensions as necessary to satisfy diverse additional requirements of web applications. XForms leaves to the host language the task of providing the presentation definition of the user interface such as fonts and colors and other augmentations. The XFDL language now incorporates XForms and augments its functionality with many additional features such as precise layout and digital signatures.

References

[1] Boyer, J. Lexical and Syntactic Specification for the Universal Forms Description Language (UFDL) Version 4.0. PureEdge Solutions. 6 SEP 1997.

[2] Bray, T., Paoli, J. & Sperberg-McQueen, C.M. (Eds.) Extensible Markup Language (XML) 1.0. W3C Recommendation. http://www.w3.org/TR/1998/REC-xml-19980210.html. 10 FEB 1998.

[3] Clark, J. & DeRose, S. (Eds.) XML Path Language (XPath) 1.0. W3C Recommendation. http://www.w3.org/TR/1999/REC-xpath-19991116.html. 16 NOV 1999.

[4] Dubinko, M., Klotz, L., Merrick, R., & Raman, T.V. (Eds.) XForms 1.0. W3C Recommentation. http://www.w3.org/TR/2003/REC-xforms-20031014/. 14 OCT 2003.

[5] Gordon, M. (Ed.) UFDL v4.0.1 Specification. PureEdge Solutions. 1993-1998.

Terminology

Terms are defined in Section 1.2 of the XML specification (see Reference [2] above).

Notation

XFDL forms are XML documents; the form definition is encoded using XML elements and attributes. In addition, XFDL imposes many constraints on the contents of the elements and the values of the attributes. In this specification, the nesting and sequence relationships between the elements and attributes are given, where possible, in DTD notation, while the constraints on certain attribute values are given in the BNF notation found in the XML specification. The DTD-syntax description of the elements and attributes is "almost complete" in that it illustrates XFDL constructs but not additional markup variations allowed by XML-related standards (e.g. namespace declarations and interspersed elements or attributes in other namespaces). Furthermore, the content models of some XFDL elements depends on an attribute value, which is also not expressible using DTD notation.

Overlap With Other Specifications

To serve its purpose, XFDL requires comprehensive presentation control and data typing machinery. This document describes a set of elements and attributes that meet these requirements. It may be the case that the presentation controls can be replaced by a W3C-specified set of form controls; however, existing specifications allow too much user interface flexibility to be suitable for security and non-repudiation purposes.

Non-Repudiation and the Document-Centric Model

A digital signature attached to a file accurately identifies the individual who used it, based on the digital certificate provider's security and the security of the user's hardware. However, to provide full non-repudiation and auditability, a business transaction not only needs to be signed by someone whose identity is verifiable, it also needs to be representative of the context in which it was signed.

With paper-based forms and documents, this is easily accomplished. Everything that appears on the signed document is considered part of the transaction. Electronic forms and documents, however, present a more complex problem in that the exact appearance and functionality of the document must be signed as well as the user's input, or the transaction is meaningless. Legal standards for font size and color must also be observed both when the document is signed and when it is subsequently examined.

Digital signature technology alone can provide the first part of the solution, but not the second. According to the Performance Guidelines for the Legal Acceptance of Records Produced by Information Technology Systems, as published by the Association for Information and Image Managements' (U.S.), the only way in which an electronic document can be considered to provide non-repudiation and auditability is if it contains the following elements, clearly recognizable, in one file:
- Individual letters, numbers and symbols
- Combinations of letters, numbers and symbols forming words or sentences
- Graphics, such as signatures, logos, pictures, and so on.
- Sounds
- Other features of records such as color, shape, texture, and so on, that relate to the content of the information

XFDL can be used to create forms that meet the above criteria by presenting a business transaction as a single entity, which is updated as the user fills it in. Item values are stored in XForms instance data, which appears in the same file that contains the user interface and presentation layer markup.

When a user digitally signs a form, the XFDL markup for the presentation layer as well as the underlying XForms instance data is signed. Subsequently, when the form is opened in an XFDL viewing or processing application, the current XFDL markup and XForms instance data are compared to those that were used to create the digital signature. If any discrepancies exist, the signature is flagged as invalid, and the form no longer provides non-repudiation or auditability.

Secondary documents can also be placed into an XFDL form as attachments, thus enabling the user to sign both the attachments and the form itself.

This method of representing and collecting information in forms and digitally signing and encrypting them ensures that the identity of the signer can be confirmed and that the signer can be proven to have signed the full content and context of the form.

Form Names and Extensions

To ensure cross-platform compatibility, form names should avoid the use of characters that are illegal under popular operating systems or in URLs. Form names should also include the .xfdl extension. The extension .xfd can also be used, but it should be avoided if possible.

MIME Types

XFDL supports the following MIME types:
- application/vnd.xfdl

The Structure of XFDL Forms

Top-Level Structure

An XFDL form is an XML 1.0 document whose root element tag is XFDL. This element must be in the XFDL namespace, a URI that includes the major and minor version of XFDL. For example,

```
[1]      <XFDL xmlns="http://www.ibm.com/xmlns/prod/XFDL/7.0"> ... </XFDL>
```

The XFDL element may contain many namespace attributes. By convention, the XFDL namespace is declared to be the default and it is also assigned to the prefix ’xfdl’. Other prefixes that are likely to appear include the ’xforms’ prefix bound to the XForms 1.0 namespace, the ’xsd’ prefix and possibly the ’xsi’ prefix from XML schema, the ’ev’ prefix for XML events, and namespace prefixes for the data vocabulary being processed by the XFDL form.

The XFDL element must contain a <globalpage> element as the first child element, followed by one or more <page> elements.

```
[2]      <!ELEMENT XFDL (globalpage, page+)>
```

The <globalpage> element must contain a single <global> element, which can contain zero or more option elements. These are referred to as form global options; they typically contain information applicable to the whole form or default settings for options appearing in the element content of pages. The <globalpage> and <global> elements must contain an attribute called *sid* which must be set to the value global. Although the attribute has a fixed value, it is still required because XFDL processors must be able to clearly identify global objects by sid even in the presence of interspersed custom elements in non-XFDL namespaces.

```
[3]      <!ELEMENT XFDL (globalpage, page+)>
[4]      <!ELEMENT global (%options;*)>
[5]      <!ATTLIST globalpage sid CDATA #REQUIRED #FIXED "global">

         <!-- This rule is only intended to communicate the  restriction with
         DTD-like notation; DTDs don't allow required and fixed (XML Schema does)
         -->
```

A <page> element contains a <global> element followed by zero or more ’item’ elements. The options in the page’s global element typically contain information applicable to the whole page or default settings for options appearing within element content of items. The page global options take precedence over form global options. A page is also required to have a ’sid’ attribute, which provides an identifier that is unique among all <page> elements (sid is short for scope identifier). The ’sid’ attribute value must not be the word ’global’ and is otherwise a letter followed by any combination of zero or more letters, digits and underscores.

```
[6]      <!ELEMENT page (global, %items;*)>
[7]      <!ATTLIST page sid CDATA #REQUIRED>
```

[8]	`sid ::= (Letter | Special) (Letter | Special | Digit | '_')* - ('global')`
[9]	`Letter ::= [A-Z] | [a-z]`
[10]	`Digit ::= [0-9]`
[11]	`Special ::= [0x00C0-0x00FF] - (0x00D7 | 0x00F7)`

The intention of using multiple pages in a form is to show the user one page at a time. Each page should contain items that describe GUI widgets including items that allow users to switch to different pages without necessarily contacting a server program. XFDL allows the page switching items to be defined in the form so the form developer can add computations that control the flow of pages based on context.

Overview of XForms Models

A form global option of particular import is called the ’xformsmodels’ option, which contains one or more <xforms:model> elements. It is recommended that a form contain only one XForms model, but multiple models are allowed (though they have no ability to interact).

An XForms model has a number of possible components, but the principal components that it defines are as follows:

- One or more instances of XML data over which the form operates. Instances usually appear directly within the form, but they may also be externally referenced
- Optional XML schema definitions for the data instances. If XML schema are used, they are often externally referenced, but they may be placed directly within an XForms model.
- Model Item Property (MIP) Definitions:
  - **calculate** — Defines an XPath formula with a string result that gives the content (value) of a node of instance data.
  - **constraint** — Defines an XPath formula with a Boolean result that helps determine the validity of a node of instance data.
  - **readonly** — Defines an XPath formula with a Boolean result that helps determine whether a node is modifiable.
  - **relevant —** Defines an XPath formula with a Boolean result that helps determine whether a node is relevant to processing. By default, the GUI widgets bound to non-relevant nodes are invisible and inactive (unless overridden by the appropriate XFDL options), and they are not submitted.
  - **type —** Defines a static string (not a dynamic XPath formula) that gives a basic schema data type for a node of instance data (these can be assigned without using an XML schema definition).
- Parameters for submission of data to a server (e.g. an http or https URL, a method of get or post, and an indication of whether the result should replace a data instance or replace the entire form). The XForms submission process removes non-relevant instance data nodes.

The XFDL items that define GUI widgets include within their content XForms controls that allow the GUI widgets (as well as invisible ’custom’ items) to connect to instance data in the XForms models. Modifications of the XFDL items cause data to be pushed through the user interface bindings of the XForms controls and into the instance data. The formulae definitions in the XForms model are then executed automatically, resulting in changes to calculated values as well as updates to validity constraints and other model item properties. All of the changes to values, validity and properties that were made by an XForms model are then percolated back out to the user interface layer by modifications to the GUI widgets and other XFDL items.

The full validity of a node of instance data is assessed by combining under Boolean-And the conformance of its content to any XML schema declarations for the node, to the schema type given by the type MIP, and to the formula given by its constraint MIP (if any of these are defined).

An instance node with a non-relevant or readonly ancestor is non-relevant or readonly regardless of any state directly declared for the node. While non-relevance and readonly status are inherited, the converse is not true. If every ancestor of a node is relevant or not readonly, then the node's relevance and readonly status are determined by the direct settings for the node (if any, or defaults otherwise).

Items

An item is a single object in a page of a form. Some items represent GUI widgets, such as buttons, check boxes, popup lists, and text fields. Other items are used to carry information such as attached word processing documents or digital signatures.

Each item must have a sid attribute, which provides a scope identifier that uniquely identifies the item from among all child items of its parent element.

An item can contain zero or more option elements. The options define the characteristics of the item, and many take default values if not defined. XForms user interface controls appear as options of XFDL items, and the XFDL item is said to be the *skin* of the XForms form control that it contains.

```
12      <!ELEMENT %items; (%options;*)>

13      <!ATTLIST %items; sid CDATA #REQUIRED>
```

XFDL allows elements in custom namespaces to appear at the item level (as long as they contain an xfdl:sid attribute). To define the items available in the XFDL namespace, the parameter entity reference to "%item;" could be defined partially as:

```
14      <!ENTITY % items "(action | box | button | check | checkgroup |
         combobox | data | field | label | line | list | pane | popup |
         radiogroup | signature | slider | spacer | table | toolbar)">
```

The details of each type of item listed in the rule above are discussed in "Details on Items" on page 35, but are summarized here for your convenience.

action A non-visible item that can perform similar tasks to a button (print, cancel, submit, and so on) either after a certain period of time or with a regular frequency.

| Skin for: <xforms:submit>, <xforms:trigger>

box An item that provides a graphic effect used to visually group a set of the GUI widgets on the page. A box is drawn under all widgets on a page.

This item is useful in some circumstances, but it is usually better to use a pane item (see below) to both visually and logically group related user interface elements.

button Performs one of a variety of tasks when pressed by the user, such as saving, printing, canceling, submitting, digitally signing the form, viewing documents enclosed in the form, and so on. A button can have a text or image face.

Skin for: <xforms:submit>, <xforms:trigger>, <xforms:upload>

check Defines a single check box.

Skin for: <xforms:input>

checkgroup
Defines a group of checkboxes that operate together to provide a multiselection capability.

Skin for: <xforms:select>, <xforms:select1>

combobox
An edit field combined with a popup list; its value can be either selected or typed.

Skin for: <xforms:select1> (select or type input), <xforms:input> (date selector)

data Used to carry binary information using base-64 encoding and compression, such as enclosed files or digital images, using base-64 encoding. This item appears when advanced XFDL enclosure mechanisms are used. When a basic <xforms:upload> is used, the data appears in an <xforms:instance> data node.

field Used to capture single- or multiple-line textual input from the user; it includes input validation and formatting features as well as enriched text capabilities.

Skin for: <xforms:input> (single-line text), <xforms:secret> (single-line, write-only), <xforms:textarea> (for multiline plain text or enriched text)

label Shows either an image or a single or multiple line text value.

Skin for: <xforms:output>

line A simple graphic effect used as a separator.

list Shows a list box populated with choices from which the user may select one.

Skin for: <xforms:select>, <xforms:select1>

pane Provides an hierarchic grouping capability for other items that are defined within the content of the pane. Also, may provide the ability to switch between multiple groupings.

Skin for: <xforms:group>, <xforms:switch>

popup Shows either the text of the currently selected choice or a label if there is no selection; the popup provides a small button that causes the list of selectable choices to appear, from which the user may select one.

Skin for: <xforms:select1>

radiogroup
: Defines a group of radio buttons. Initially none may be selected, but a maximum of one radio button can be selected within the group.

signature
: Receives the signature that ultimately results when a user presses a signature button.

  Skin for: <xforms:select1>

slider
: Creates a sliding control, similar to a volume control, that lets the user set a value within a specific range.

  Skin for: <xforms:range>

spacer
: An invisible GUI widget that facilitates spacing in the relational positioning scheme.

table
: Provides a template of XFDL items that are to be duplicated according to the amount of data available to be displayed. This item provides the ability to dynamically adjust the form rendition based on the amount of data and the amount of changes to that data.

  Skin for: <xforms:repeat>

toolbar
: Items associated with a toolbar item appear in a separate window pane above the pane for the form page; it is the typical location for page switching and other buttons as its contents are not printed if the form is rendered on paper.

Note: The parameter entity %items is not intended as a formal definition of the content model of a page (after the global element). It is only intended to present the list of items. The items could be explicitly namespace qualified with a prefix bound to the XFDL URI (given in Rule 1). Moreover, XFDL permits form authors to intersperse custom elements among the items as long as those elements have a ’sid’ attribute in the XFDL namespace. Custom elements can be used, for example, to carry complex instructions and logic for server-side components. While simple static application-specific information could be represented with XML processing instructions, many server side applications (e.g. workflow and database requests) require complex instructions that can include the use of the XFDL compute system to collect information from around the form. For more information, see <custom_option>.

Options and Array Elements

Options can appear as form globals, page globals, or as the contents of items. An option defines a named property of an item, page, or form. The parameter entity reference to ″%option;″ could partially be defined as follows:

```
[15]    <!ENTITY % options "(xformsmodels | xforms:input | xforms:secret |
        xforms:textarea | xforms:range | xforms:select1 | xforms:select |
        xforms:trigger | xforms:submit | xforms:upload | xforms:output |
        xforms:repeat | xforms:group | xforms:switch | acclabel |
        activated | active | bgcolor | border | colorinfo | coordinates |
        data | datagroup | delay | dirtyflag | excludedmetadata | filename |
        first | fontcolor | fontinfo | format | formid | fullname | image |
        imagemode | itemlocation | justify | label | labelbgcolor |
        labelborder | labelfontcolor | labelfontinfo | last | layoutinfo |
        linespacing | mimedata | mimetype | next | pageid | previous |
        printbgcolor | printfontcolor | printlabelbgcolor |
        printlabelfontcolor | printsettings | printvisible | readonly |
        requirements | rtf | saveformat | scrollhoriz | scrollvert |
        signature | signatureimage | signdatagroups | signdetails | signer |
        signformat | signgroups | signinstance | signitemrefs | signitems |
        signnamespaces | signoptionrefs | signoptions | signpagerefs | size |
        texttype | thickness | transmitdatagroups | transmitformat |
        transmitgroups | transmititemrefs | transmititems | transmitnamespaces |
        transmitoptionrefs | transmitoptions | transmitpagerefs | triggeritem |
         type | url | value | visible | webservices | writeonly)">
```

Again, the definition is partial because XFDL supports namespace qualification of options as well as the interspersion of custom options in non-XFDL namespaces. Typically, application-defined options occur in application-defined items, but they are also sometimes used in XFDL-defined items to store intermediate results of complex computations, thereby allowing the form developer to arbitrarily break down a problem into manageable pieces. For more information, see “<custom option>” on page 203.

Also, note that only a subset of these options is valid for any XFDL item. For example, an xforms:repeat is only valid in a table item. The XFDL options are fully discussed in “Details on Options and Array Elements” on page 69 and are summarized in the following sections.

XForms-related Options

<xforms:input>
: Binds to a node of instance data for the purpose of collecting/presenting a single line of text or piece of information. A <field> presents the text, and automatically translates various data types like dates and currencies to schema compliant values. A <check> item appears checked or unchecked based on an xsd:boolean interpretation of the bound instance node. The <combobox> skin is specific to date selection. A custom item skin can be used to help move any data from the XFDL layer to the instance.

<xforms:secret>
: Binds to a node of instance data for the purpose of collecting a single-line password. A <field> contains this option and presents itself as write-only.

<xforms:textarea>
: Binds to a node of instance data for the purpose of collecting/presenting multiline plain text or enriched text in a <field> item.

<xforms:output>
: Presents text or an image in a <label> item. If the control binds to a node of instance data, and the mediatype attribute contains an image-related type (e.g. image/*), then an image is presented. Otherwise, text is presented.

<xforms:select1>
: Provides the ability to select one from a set of choices. The presentation of the set of choices is governed by the item type that skins the control, which can be <popup>, <combobox>, <list>, <checkgroup>, or <radiogroup>

<xforms:select>
: Provides the ability to select choices from a set of choices presented by a <checkgroup> or <list>.

<xforms:submit>
: Provides the ability to activate an <xforms:submission> appearing in an XForms model, as the result of activating either an XFDL <button> or <action> item.

<xforms:trigger>
: Provides the ability to activate a sequence of XForms actions, as the result of activating either an XFDL <button> or <action> item.

<xforms:upload>
: Provides the ability for an XFDL button to attach content from the local computer system to a form by placing an encoded version of it into an indicated instance node.

<xforms:group>
: Allows a set of user interface controls to be packaged together, visually and logically, with the <pane> item skin.

<xforms:switch>
: Appears in a <pane> skin item and offers grouping capabilities similar to an <xforms:group> except that multiple grouping cases can be specified and switched to during the run of the form.

<xforms:repeat>
: Appears in the <table> item and provides the ability to iterate its content of XFDL items once per node in a set of nodes selected from the XForms instance data.

<xforms:range>
: Sets the range of values a user can select with a <slider> item.

Options in the XFDL Namespace

acclabel
: Provides a special description of input items that is read by screen reading software.

active
: Specifies whether an item is active or inactive. In XFDL items containing XForms controls, the default for this option is set by the relevant model item property.

bgcolor, fontcolor, labelbgcolor, and labelfontcolor
: Specify the colors for an item or its label using either predefined names or RGB triplets in decimal or hexadecimal notation.

border and labelborder
: Control whether an item or its label has a border.

colorinfo
: Records the colors used to draw the form when the user signs the form. This is only necessary when the operating system colors are used instead of the colors defined in the form (which is a feature for users with vision impairments).

coordinates
: Receives the location of a mouse click on an image, if the image is in a button.

data and datagroup
: Used to create an association between data items and the buttons that provide file enclosure functionality.

delay Used in an action item to specify the timing for the event and whether it should be repeated.

excludedmetadata
: Used to store special information that is automatically excluded from signatures.

filename and mimetype
: Give additional information about an enclosed document.

fontinfo and labelfontinfo
: Defines the typeface, point size, and special effects (bold, italics, and underline) for the font used to display the item's value or label.

format Contains sub-elements that parameterize input validation for the item's value.

formid Defines a unique identifier for the form, such as a serial number.

fullname, layoutinfo, signature, signatureimage, signdatagroups, signdetails, signer, signformat, signgroups, signinstance, signitemrefs, signitems, signnamespaces, signoptionrefs, signoptions, and signpagerefs
: Work together to provide a full-featured digital signature as defined in "Origin and Goals" on page 1 (goal 9).

image Identifies the data item containing the image for the button or label.

imagemode
: Specifies the display behavior of the image within the data item; the image may be clipped, resized, or scaled to fit the item.

itemlocation, size and thickness
: Help to define the location and size of the item.

justify Controls whether text in the item should be left, center, or right justified.

label Associates a simple text label with the item; labels can also be created independently with a label item.

linespacing
: Adjusts the spacing between lines of text in an item.

mimedata
: Used to store large binary data blocks encoded in bas-64 gzip compressed or base-64 format.

next and previous
: Link the item into the tab order of the page.

pageid Defines a unique identifier for a page, such as a serial number.

printbgcolor, printlabelbgcolor, printfontcolor, and printlabelfontcolor
Provide the ability to set printing colors for each indicated option different from the display colors on the screen.

printvisible
Determines whether an item should be visible when the form is printed. Has no effect on the visibility of the item on the screen.

printsettings
Parameterizes the paper rendition of a form.

readonly
Sets the item to be readonly. In XFDL items containing XForms controls, the default for this option is set by the readonly model item property.

rtf Contains the rich text value of rich text fields.

requirements
Specifies the requirements for the Web Services to be used by the form.

saveformat and transmitformat
Control how the form is written (XFDL, HTML) when it is saved or submitted.

scrollhoriz and scrollvert
Control whether a text field item has horizontal and vertical scroll bars or whether it wordwraps, allows vertical sliding, and so on.

texttype
Sets whether a field contains plain text or rich text.

transmitdatagoups, transmitformat, transmitgroups, transmititiemrefs, transmititems, transmitnamespaces, transmitoptionrefs, transmitoptions, and transmitpagerefs
Work together to allow you to transmit form submissions.

triggeritem
Set in the form globals to identify which action, button, or cell activated a form transmission or cancellation.

type Specifies whether the action, button, or cell item will perform a network operation, print, save, digitally sign, and so on.

url Provides the address for a page switch, or for a network link or submission.

value Holds the primary text associated with the item. In XFDL items that contain XForms controls, this option (and all options, such as those that are computed) are treated as transient, which means that any updates to the content are not serialized when the form is written because the updates are reflected in instance data.

visible Determines whether the item should be shown to the user or made invisible.

webservices
Defines the nameof the Web Services used by the form.

writeonly
Sets the item to be writeonly. This option is only for use with field items that do not skin XForms controls.

Implicit Options

There are some options that are defined within XFDL for the purpose of allowing them to be referenced without being defined by the form author. These options are dynamically added to the document object model (DOM) of the XFDL form while it is being processed, and they are removed when it is serialized. These options tend to be informational in nature or representative of events that can occur while the form is being processed.

activated, focused, and mouseover
: Indicates whether the form, page or item has been activated or focused or contains the mouse pointer.

dirtyflag
: In the form global item, this option indicates whether the end-user of the form viewing program has changed the form.

focuseditem
: At the page global level, records the scope identifier of the item that currently has the focus.

itemprevious, itemnext, itemfirst, itemlast
: Used to help create a doubly linked list of items in each page. The itemprevious and itemnext options occur in each item, and itemfirst and itemlast appear at the page global level.

keypress
: Records a keypress by the user that was not used as input to an XFDL item. The keypress is propagated upwards to the page and form global items.

pageprevious, pagenext, pagefirst, pagelast
: Used to help create a doubly linked list of pages in the form. The pageprevious and pagenext options occur in each page, and pagefirst and pagelast appear at the form global level.

printing
: In the form global item, this option indicates whether the form is currently printing.

version
: Appears in the form global item and defines the version of XFDL used to write the form. It is obtained from the XFDL namespace declaration.

Content Models for XFDL Options

The content of an option can take one of two formats: simple character data or a subtree of XML elements. In the latter case, the element children are typically referenced as a zero-based array. The XFDL-specific options that have simple content versus array content are defined by the parameter entity references %simpleOption; and %arrayOption; below:

[16]
```
<!ENTITY % simpleOptions "(acclabel | activated | active | bgcolor |
border | data | dirtyflag | filename | focused | focuseditem |
fontcolor | formid | fullname | image | imagemode | justify |
keypress | label | labelbgcolor | labelborder | labelfontcolor |
linespacing | mimedata | mimetype | mouseover | next | pageid |
previous | printbgcolor | printfontcolor | printing | printlabelbgcolor |
printlabelfontcolor | printvisible | readonly | rtf | saveformat |
scrollhoriz | scrollvert | signature | signatureimage | signer |
signformat | texttype | thickness | transmitformat | triggeritem |
type | url | value | version | visible | writeonly)">
```

[17]
```
<!ELEMENT %simpleOptions; (#PCDATA)>
```

[18]
```
<!ENTITY % arrayOptions "(colorinfo | coordinates | datagroup | delay
excludedmetadata |  fontinfo | format | itemlocation | labelfontinfo |
layoutinfo | printsettings | requirements | signdatagroups |
signdetails | signitemrefs | signitems | signnamespaces |
signoptionrefs | signoptions | signpagerefs | size |
transmitdatagroups | transmitgroups | transmititemrefs | transmititems |
transmitnamespaces | transmitoptionrefs | transmitoptions |
transmitpagerefs | webservices)">
```

While the simple options are clearly shown above to contain character data, the further details of their content models are not shown. For example, all color options support specification of an RGB triplet using either #RRGGBB in hexadecimal or *rrr,ggg,bbb* in decimal as well as any color name given in "Color Table" on page 399.

The array options defined above also each contain subelements that could contain either simple character data or further element depth. The following are the content models of the various array options, where all undefined elements resolve to simple character content (PCDATA):

[19]
```
<!ELEMENT colorinfo (window,windowtext,borderlight,bordershadow,
buttonface,buttontext)>
```
[20]
```
<!ELEMEMT coordinates (x,y)>
```
[21]
```
<!ELEMENT datagroup (datagroupref+)>
```
[22]
```
<!ELEMENT delay (type,interval)>
```
[23]
```
<!ELEMENT excludedmetadata (servernotarizations)>
```
[24]
```
<!ELEMENT servernotarizations (notarization+)>
```
[25]
```
<!ELEMENT fontinfo (fontname, size, effect+)>
```
[26]
```
<!ELEMENT format (datatype (date, day_of_month, day_of_week,
date_time, currency, float, integer, month, string, time, void,
year)*, presentation (calendar, casetype, currencylocale,
decimalseparator,fractiondigits, groupingseparator, keepformatindata,
negativeindicator, pad, padcharacter, pattern, patternrefs, round,
separator, showcurrency, significantdigits, style)*, constraints
(casesensitive, checks, decimalseparators, groupingseparators, length,
mandatory, message, patterns, range, separators, template, yearwindow)*)>
```
[27]
```
<!ELEMENT itemlocation (x | y | width | height | offsetx | offsety |
above | after | before | below | within | alignb2b | alignb2c | alignb2t |
alignc2b | alignc2l | alignc2r | alignc2t | alignhorizbetween |
alignhorizc2c | alignl2c | alignl2l | alignl2r | alignr2c | alignr2l |
alignr2r | alignt2b | alignt2c | alignt2t | alignvertbetween |
alignvertc2c | alignb2b | expandb2c | expandb2t | expandl2c |
expandl2l | expandl2r | expandr2c | expandr2l | expandr2r | expandt2b |
expandt2c | expandt2t)+>
```
[28]
```
<!ELEMENT labelfontinfo (fontname, size, effect+)>
```
[29]
```
<!ELEMENT layoutinfo (pagehashes (pagehash (pageref, hash))+)>
```
[30]
```
<!ELEMENT printsettings (pages, dialog, border, pagelayout,
radiosaschecks, radioswithoutvalues, scroll barsonfields,
singlelinefieldsaslines)>
```
[31]
```
<!ELEMENT pages (filter, pageref+)>
```
[32]
```
<!ELEMENT dialog (active, copies, orientation, printpages)>
```
[33]
```
<!ELEMENT printpages (active, choice)>
```
[34]
```
<!ELEMENT requirements (requirement*, detected)>
```

[35]	`<!ELEMENT (signdatagroups | transmitdatagroups) (filter, datagroupref+)>`
[36]	`<!ELEMENT signdetails ((dialogcolumns (property+))?, (filteridentity (filter (tag, value))+)?)>`
[37]	`<!ELEMENT (signitemrefs | transmititemrefs) >`
[38]	`<!ELEMENT (signitems | transmititems) >`
[39]	`<!ELEMENT (signnamespaces | transmitnamespaces) >`
[40]	`<!ELEMENT (signoptionrefs | transmitoptionrefs) >`
[41]	`<!ELEMENT (signoptions | transmitoptions) >`
[42]	`<!ELEMENT (signpagerefs | transmitpagerefs) >`
[43]	`<!ELEMENT size (width, height)>`
[44]	`<!ELEMENT webservices (wsdl*)>`
[45]	`<!ELEMENT xformsmodels (xforms:model+)>`

For any character data option or array element in XFDL, an *encoding* attribute can be specified for the content. As shown below, the content model of simple options is character data. The values 'xml' (for plain text), 'base64' and 'base64-gzip' can be used in the encoding attribute. The default is 'xml' except for the mimedata option, which as 'base64-gzip' as the default. A content encoding of base64 is useful if the content being encoded is already in a compressed format, such as PNG or JPEG. If the encoding is not 'xml' then an XFDL-compliant processor applies the requisite decoders before interpreting the content in the application context, and it applies the appropriate encoders during serialization. The XFDL encoding attribute can also be used on custom options and array elements, but it would have to be namespace qualified since custom options are in a non-XFDL namespace.

Any character data option or array element can be qualified with the *transient* attribute. The valid values are on and off, with a default of off for options and array elements in XFDL items not containing XForms user interface elements and on in XFDL items that do contain XForms user interface controls. When an option or array element is transient, its content can be changed during run time, but the original content obtained when the form was read will be restored upon serialization. In an XFDL item whose <value> option is controlled by an XForms user interface binding, the instance data holds the current value bound to the control, so there is no need for the presentation layer node to be changed. Transience allows options and array elements to be changed during run-time of the form without breaking a digital signature over the option.

The content of any character data option or array element can also be qualified by a *compute* attribute, which defines an expression used to obtain and update the content based on content elsewhere in the form. Typically, computations over data are performed in an XForms model using the <xforms:bind> element and its calculate attribute or other model item properties such as constraint, readonly, relevant, and required. However, calculations of presentation layer properties are performed with XFDL computes, and where they are dependent on instance data, they reference the transient <value> option of a user interface item bound to that data. The details of XFDL computes appear in "The XFDL Compute System" on page 419, the basic idea can be gleaned from the following example, which shows label that turns red when the value is negative:

```
<label sid="IncomeTax">
   <xforms:output ref="incomeTax/Total"/>
   <fontcolor compute="value >= '0' ? 'black' : 'red'"/>
</label>
```

At the start-up, the fontcolor option has empty content. When the XForms user interface binding creates the <value> option and places the purchase order total value into it, then the compute in the fontcolor is automatically evaluated, with a result of either red or black. When the form is serialized, e.g saved to disk, submitted or signed, the fontcolor content rendered into the serialization is empty because all options in an XForms controlled XFDL item are transient. Thus, XFDL computes can run even if the label item is signed because the countenance of the label can only be changed if the data changes, which can only occur if the data is not signed.

Locales

XFDL forms are designed to be locale and language aware. This means that each form is designed for a specific language and set of locales. Locale support is identified through the xml:lang attribute. This attribute is primarily added to the XFDL tag in the form, and identifies which locale the form was designed for. For example, the following form was designed for the English U.S. locale:

```
<XFDL xmlns="http://www.ibm.com/xmlns/prod/XFDL/7.0" xml:lang="en-US">
```

If the xml:lang attribute is not included on the <XFDL> tag, it defaults to the en-US locale.

For more information:

- Regarding language codes, see:
  `http://www.loc.gov/standards/iso639-2/englangn.html#ef`
- Regarding country codes, see:
  `http://ftp.ics.uci.edu/pub/websoft/wwwstat/country-codes.txt`
- About the locales supported, refer to the *Locales Specification for XFDL*.

Characters in Character Data Content

The ampersand (&) and left angle bracket (<) are not permitted in character data content of XML elements, since these characters mark the beginning of entity references and XML tags, respectively. There may be occasions in which a developer needs to include these in XFDL character content (for example, e-mail URLs). In such cases, the developer can include the ampersand and left angle bracket in a few different ways, such as using the XML entities *&amp;* and *&lt;* or simply wrapping the character content in a CDATA section. The character sequence]]>, which normally cannot appear in character data content, appears in a CDATA section as the delimiter token that marks the end of a CDATA section. When the sequence]]> appears as character data, then the right angle bracket (>), if necessary, should be expressed with an entity reference such as *&gt;*.

These rules apply to all XFDL elements, but they are often used when constructing e-mail URLs, which use the ampersand to indicate additional parameters. For example:

```
<url><![CDATA[mailto:everyone@world.earth?body=Hello]]></url>
```

Base-64 and Compressed Encoding of Binary Data

In XFDL, option and suboption elements are allowed to store base-64 encoded and compressed base-64 encoded binary data such as signatures, images, enclosed word processing or spreadsheet documents, and so on. XFDL allows an *encoding* attribute to control whether an element contains data in a format other than plain

XML character data. Both compressed and uncompressed base-64 encoding use no characters that are illegal in character data. Typically, based-64 and compressed base-64 encodings are used with the <mimedata> option in XFDL.

Since binary data tends to be long, XFDL processors are expected to ″pretty print″ the lines of base-64 (whether or not the binary data is first compressed). Each level of element depth beyond the root <XFDL> tag corresponds to three spaces. The first line of base-64 is expected to be on the line after the element start tag, and the lines of base-64 are expected to be indented three spaces more than the element start tag. For example, the <mimedata> option's start tag is indented 9 spaces (three for the page level, three for the item level and three for the option level), so each line of base-64 content in a <mimedata> option begins with 12 spaces. This should be followed by 76 characters of base-64 (except for the last line), then a linefeed (not a return-newline sequence). Thus, the element end tag is on the line after the last line of base-64 content. The end tag is indented so that it aligns under the start tag (e.g. indent 9 spaces for the <mimedata> option).

The above method describes the creation of new base-64 encoded content. However, the normal XML preservation of whitespace in element content should be used when reading XFDL forms so that prior whitespace indenting techniques used with older XFDL forms and by custom options that use the XFDL encoding attribute will continue to function properly (esp. with regard to digital signature validation).

Scope Identifiers (sid)

An XFDL scope identifier, or sid, uniquely identifies an element within the scope of its logical parent. Each <page> element must have a *sid* attribute that uniquely identifies the page within the surrounding XFDL form element. An item element must have a *sid* attribute that uniquely identifies the item within the surrounding page element.

When an item appears in an <xforms:group>, <xforms:repeat> or a case of an <xforms:switch>, then the scope identifier uniquely identifies the item within the group, switch case or repeat template. When a repeat template is instantiated for each node in the repeat nodeset, the items generated are unique within each template instance (each logical row of a table).

In XFDL, each option element is defined to be uniquely identified within the scope of the surrounding item element by its XML tag, which is why options (and array elements) do not require a *sid* attribute.

Commenting XFDL

Comments are text added to the form that is ignored by XFDL processors. This allows form developers to document the form from within the XFDL source code. This helps subsequent form and application developers to immediately understand the purpose of a particular block of markup, such as a complex compute or function call. Comments are always wrapped in a special sequence of characters that indicate the beginning and end of a comment section.

XFDL respects the standard XML comment style, which opens with **<! --** and closes with **-->**. For example:

```
<! -- This is a code comment. -->
```

Because XFDL is an XML vocabulary, comment blocks can be of any length, from one line to multiple lines.

Document Reproducibility

XFDL processors are expected to preserve the XML prolog and epilog, the comments within the XFDL element, and all attributes appearing in start tags but not specifically defined by XFDL. The attributes must be associated with their respective start tags, and the comments must be associated with the respective pages, items, options, or array elements to which they apply. Additionally, all foreign-namespaced elements and attributes must be preserved. The XFDL processor must be able to reproduce these language components for signatures and for saving or transmitting the form.

Small XFDL Form Examples

The first example in Figure 2 is designed to show a whole XFDL form. After the XML prolog, the root XFDL element declares the XFDL namespace URI to be the default namespace, and it binds the prefix *xfdl* to the XFDL namespace and the prefix *xforms* to the XForms namespace. Implicit in the XFDL namespace URI is the XFDL language version (7.0) to which the form complies. There is a form global variable stating that all pages should have a light gray background color. However, the page global background color is set to cornsilk. Since page globals override form globals, the page will have a cornsilk background (see "Color Table" on page 399 for a list of valid color names in XFDL).

The form global also contains an XForms model, which creates a dataset representing the three sides of a triangle (a, b, and c). The model also includes a <bind> element that set sets the value of c based on the values of a and b using the pythagorean theorem.

The page global item contain a *label* option that declares the caption bar text for the window used to display the page. Note that *label* is a keyword that is used both as an item type and an option scope identifier. Widgets such as fields and comboboxes can have text labels associated with them, but image and text labels can also be placed anywhere on the form, so a separate label item is required in the language.

After the global options, the page contains three fields that are bound to the a, b, and c elements in the data model. The first two fields collect side lengths for a right angle triangle. The third fields displays the length of the hypotenuse, which is automatically calculated by the <bind> in the data model based on the length of the other two sides. The *readonly* option is added to prevent the user from accidentally overwriting the value for field C.

```
<?xml version="1.0" encoding="UTF-8"?>
<XFDL xmlns="http://www.ibm.com/xmlns/prod/XFDL/7.0"
   xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
   xmlns:xforms="http://www.w3.org/2002/xforms">
   <globalpage sid="global">
      <global sid="global">
         <xformsmodels>
            <model xmlns="http://www.w3.org/2002/xforms">
               <instance>
                  <data xmlns="">
                     <a>3</a>
                     <b>4</b>
                     <c></c>
                  </data>
               </instance>
               <bind nodeset="c" calculate="power(../a * ../a +
                  ../b * ../b, 0.5)"/>
            </model>
         </xformsmodels>
         <bgcolor>lightgrey</bgcolor>
      </global>
   </globalpage>
   <page sid="PAGE1">
      <global sid="global">
         <label>Pythagorean Theorem Form</label>
         <bgcolor>cornsilk</bgcolor>
         <fontinfo>
            <fontname>Times</fontname>
            <size>24</size>
            <effect>plain</effect>
         </fontinfo>
      </global>
      <field sid="side1">
         <xforms:input ref="a">
            <xforms:label>Enter length of side 1:</xforms:label>
         </xforms:input>
         <labelfontinfo>
            <fontname>Times</fontname>
            <size>24</size>
            <effect>bold</effect>
         </labelfontinfo>
      </field>
      <field sid="side2">
         <xforms:input ref="b">
            <xforms:label>Enter length of side 2:</xforms:label>
         </xforms:input>
         <labelfontinfo>
            <fontname>Times</fontname>
            <size>24</size>
            <effect>bold</effect>
            </labelfontinfo>
      </field>
      <field sid="Hypotenuse">
         <xforms:input ref="c">
            <xforms:label>The hypotenuse length is:</xforms:label>
         </xforms:input>
         <labelfontinfo>
            <fontname>Times</fontname>
            <size>24</size>
            <effect>bold</effect>
            </labelfontinfo>
         <readonly>on</readonly>
      </field>
   </page>
</XFDL>
```

Figure 1. A Simple XFDL Form

The second example in Figure 3 (below) omits the XML prolog and the declarations for the root XFDL element and page. The example only shows two items. It is designed to demonstrate deeper element depth and more computes than the form shown in Figure 2.

The first item is a field that purports to ask the user what portion of a bill, such as a credit card bill, will be paid. The *format* option contains a number of array elements. The first element sets the data type to 'currency', which indicates the type of user input that is permitted.

The <presentation> elements determine how user input is displayed. In this case, there is only one setting, which stipulates that the dollar symbol should be appended to the user's input.

The <constraints> elements further restrict user input. The first setting turns the 'mandatory' status on, which means that input is required in that field (i.e., emptiness is not a permitted response). The second constraint, named 'range', contains an array of two elements that define the lower and upper bounds of the user's input. For a credit card bill, the range of payment is typically bounded above by what the cardholder owes and bounded below by some small percentage of the current balance. Thus, the *format* option shows the possibility of unlimited array element depth as well as the inclusion of computes deep within the element hierarchy. XFDL offers what is known as a fine-grain compute system.

The second item element in Figure 3 is a label that demonstrates a longer compute expression, including several array element references. Note that at the end of the compute, the 700 is concatenated to the end of the string rather than added to the 35. Because addition is left associative, the entire portion of the string prior to the 700 has already been constructed. Therefore, due to run-time type identification, the last + operator performs string concatenation, not a mathematical addition.

```
<field sid="PayNow">
   <label>
      What portion of this bill do you want to pay now?
   </label>
   <value></value>
   <format>
      <datatype>currency</datatype>
      <presentation>
         <showcurrency>on</showcurrency>
      </presentation>
      <constraints>
         <mandatory>on</mandatory>
         <range>
            <min compute="Balance.value * '0.05'">35</min>
            <max compute="Balance.value">700</max>
         </range>
      </constraints>
   </format>
</field>
<label sid="DemonstrateSuboptionReferencing">
   <value compute="PayNow.format[datatype] + ' ' + &#xA;
      PayNow.format[presentation][showcurrency] + ' ' &#xA;
      + PayNow.format[constraints][range][min] + &#xA;
      PayNow.format[range][max]"
      >currency on 35700</value>
</label>
```

Figure 2. Example of Suboption Array Elements

signatures in XFDL

As discussed previously (in "Non-Repudiation and the Document-Centric Model" on page 2), digital signature technology provides part of the solution for creating non-repudiation in a document-centric model. As such, XFDL supports the application of digital signatures to any XFDL document. However, to properly implement support for digital signatures, both common use scenarios and overall security must be considered.

For example, signatures are often used to sign only a portion of a document. Furthermore, a secondary signature is often used to sign the rest of the document while also endorsign the first part of the document. The classic example of this is the "For Office Use Only" section in any form. The implementation of digital signatures in XFDL must support scenarios like this, allowing both for filtering of what is signed and for overlapping signatures.

Furthermore, while digital signatures clearly identify the user, the application of digital signatures must also add a measure of security to the document itself. That is, once a document is signed, it should be impossible to change any of the information that was signed. Thus, a number of algorithms and rules must be enforced by the XFDL processor in use.

Applying signature Filters

XFDL supports a filtering system for signatures. In effect, this allows any combination of form elements to be either included or excluded from a signature, which in turn allows forms to be divided into logical sections for the purposes of signing.

For example, a document may include a "For Office Use Only" section that should not be signed by the original user. By applying the correct filters, this section can be excluded from the signature, allowing office workers to complete those portions of the document even after the document has been signed.

XFDL includes a series of signature filters. Each filter applies to a different cross-section of XFDL elements. For example, the *signitems* and *signitemrefs* filters control which items are signed or ignored, while the *signoptions* and *signoptionrefs* filters control which options are signed or ignored. Each level of filter also has an assigned order of precedence. For example, filters at the option level override filters at the item level.

By using these filters in combination, XFDL provides complete control over which elements are omitted from a signature (or alternately to indicate which elements should be included in a signature, though 'inclusive logic' filters should be used sparingly and with great care).

The complete list of available filters is: *signitems*, *signoptions*, *signpagerefs*, *signdatagroups*, *signgroups*, *signitemrefs*, *signnamespaces*, and *signoptionrefs*. These filters are implemented as XFDL options, and are detailed later in this document. The order of precedence for these filters is outlined in "Order of Precedence of Filters" on page 395.

Namespaces in signature Filters

Some signature filters refer to the elements to be omitted (or included) by their element tag names. These filters are *signitems*, *signoptions* and *signoptionrefs*. These options are lists whose members are compared to element tags. The comparison is namespace aware. For example, if *<itemref>xfdl:field</itemref>* is a member of the *signitems* filter, then the member will match any item-level element with the local name *field* and a namespace URI equal to the one bound to the prefix *xfdl*. Note that if a namespace prefix used in an element tag is not mapped to the same namespace URI as it is in a signature filter member, then the signature filter member will not match the element. For example, if the xfdl prefix is mapped to the XFDL namespace URI in the above *signitems* member, then the following elements will match:

```
<xfdl:field xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0">
<field xmlns="http://www.ibm.com/xmlns/prod/XFDL/7.0">
<custom:field xmlns:foobar="http://www.ibm.com/xmlns/prod/XFDL/7.0">
```

but it will not match:

```
<xfdl:field xmlns:xfdl="http://custom.HR">
<foobar:field xmlns:foobar="http://custom.HR">
<field xmlns="">
```

The last <field> element above has an empty namespace URI. To match such an element, the *null* prefix must be used, e.g. *<itemref>null:field</itemref>*.

The namespace prefixes used in evaluating each member of a signature filter are obtained from the namespace context of the element containing the signature filter member. Each member of a signature filter may define different namespace prefixes. However, the default namespace is the XFDL namespace. For example, the member *<itemref>field</itemref>* in *signitems* matches any item-level <field> element with a default namespace URI equal to the XFDL namespace URI as well as namespace qualified elements such as <xfdl:field> if the given prefix is bound to the XFDL namespace URI.

signature filter members that are namespace qualified with a prefix bound to the XFDL namespace URI are logically equivalent to signature filter members that are associated with the XFDL namespace URI by the default namespace. For example, a *signitems* member of *<itemref>field</itemref>* with the default namespace of XFDL is logically equivalent to the following, where the prefix *xfdl* is bound to the XFDL namespace URI:

```
<xfdl:itemref>field</xfdl:itemref>
<xfdl:itemref xmlns="http://custom.HR">field</xfdl:itemref>
<xfdl:itemref xmlns="">field</xfdl:itemref>
```

Suboptions of a signature filter that are not in the XFDL namespace are ignored. For example, the *signitems* suboption *<itemref xmlns="">field</itemref>* has no effect on signature filtration.

Applying Multiple signatures

Documents often require multiple signatures. Furthermore, it is common practice for some signatures to endorse other signatures. These secondary signatures can be said to overlap the original signatures, since they sign both the document and the original signature.

For example, an insurance claim requires the claimant to sign the document. Later, the insurance adjuster may also have to sign the document, both to endorse the information provided by the claimant and to endorse information they have added to the claim.

XFDL allows any number of signatures in a single document. The signatures will sign separate portions of the form, or will overlap with other signatures, as specified by the filters used.

For example, the first signature may use a set of filters that includes all elements in the top half of a page. The second signature may use a filter that includes the first signature and the top half of the page. Finally, a third signature might use a filter that includes the entire page and both the first and second signatures.

Securing signed Elements

Paper documents rely on ink to secure the document. That is, once a document is signed, it is difficult to change the document because it is difficult to erase ink from paper. The very nature of paper and ink enforce the security of the document, since attempts to change the document generally leave detectable traces.

In contrast, digital documents do not share this type of inherent security. In fact, most digital documents allow easy and undetectable modification. For example, word processing documents are easily opened, changed, and saved again without leaving any evidence of what, if anything, was changed or who made the changes.

For this reason, the XFDL processor must provide the necessary security. Once an element in a document is signed, it is implicit that future readers should be unable to change that element. Thus, once an XFDL document is signed, the XFDL processor must ensure that those elements included in the signature filters cannot be changed.

This feature is not explicitly supported by XFDL, in that there are no flags in the language that indicate whether a particular element in a document is signed. Instead, the software processing the XFDL must interpret existing signatures and enforce the rules correctly based on the filters in those signatures.

Preventing Layout Changes

Once a document is signed, it is also implicit that the layout of that document should be secure. For example, if it were possible to move a paragraph, or even a line, the meaning of the document could be changed.

To reflect this, any software processing XFDL must maintain the position of signed visual elements. This means that both the position and the size of the visual elements must be secured. If a visual element can change size, then it could be enlarged to obscure another visible element and thereby change the meaning of the document. Clearly, this must be prevented.

Thus, when a document is signed, the width, height, and position of all visible signed elements must be recorded. XFDL provides the *layoutinfo* option as a place to store this information within a given signature element. Furthermore, the *layoutinfo* option itself should be signed as part of the signature, ensuring that it cannot be changed.

The layout can later be tested by re-calculating the position of all signed elements and comparing this to the information stored in the layoutinfo option for that signature. If the information does not match, then the document has been modified and cannot be trusted.

The software processing the XFDL should perform this layout test at the following times:

- Immediately after a signature is created, it should test the entire document. This ensures that the process of generating the signature did not change the information.
- Whenever a page of the document is viewed, it should test the signed contents of that page.
- Whenever an item is computationally added, deleted, or moved, it should test the appropriate page.
- Whenever the details of a signature are viewed, it should test all portions of the document signed by that signature.

Preventing Exploitable Overlaps of signed Elements

Unlike paper documents, digital documents also offer the potential for visual elements to overlap. For example, it is possible to create a block of text in a document, and to then obscure or hide that text with a second, overlapping block of text. In this scenario, even if the first block of text was secured with a signature, it would be possible to move the second block of text after the document was signed. This would change the meaning of the document by revealing information that was previously hidden.

Since the guiding principle of signatures is that ″you sign what you see,″ a scenario in which visual items are hidden or significantly overlapped cannot be allowed. If the signer cannot see elements of the form, then the signature cannot be considered valid.

When a document is signed, the XFDL processor must ensure that none of the signed visual elements are overlapping with unsigned visual elements. If an overlap is detected, the software must either warn the user or prevent the signature from being created. Furthermore, if an existing signature is found to include elements that are overlapping with unsigned elements, the document has been altered and the software must warn the user.

However, this test must allow for certain tolerances. Most of the visual elements in an XFDL document are surrounded by a small border of unused space which can be allowed to overlap without obscuring the item itself. For example, two labels might overlap slightly without the text in either label being obscured. In fact, this sort of overlap is often necessary when reproducing tightly spaced paper forms. Thus, an overlap of two pixels should be allowed for each item.

This test may also ignore signed elements that overlap each other, since the layout tests discussed earlier prevent signed elements from being moved. Furthermore, this test must also make exceptions for box items. Boxes are often used to visually create sections in a document, and will overlap other visual elements as a result. This overlap is allowed in the following cases:

- A signed box can overlap with any unsigned item, with the exception of other boxes.

- A signed box can overlap with an unsigned box if the unsigned box appears on top of the signed box (that is, if the unsigned box comes after the signed box in the XFDL serialization, such that it is drawn after the signed box). This allows the desired behavior of signing a large box but allowing unsigned items (including boxes) to appear on top of part of the signed box, and it disallows the unsigned box from later being moved in the XFDL serialization such that it disappears. Note that an unsigned box can be added after signing and allowed to overlap a signed box that was previously unobscured, but the other overlap rules prevent this from happening if the unsigned box overlaps other signed items in the signed box. To protect empty spaces in signed boxes from being obscured by unsigned boxes, the form author should place empty signed transparent labels in the spaces.

The XFDL processor should perform this overlap test at the following times:

- Immediately after a signature is created, it should test the entire document. This ensures that the process of generating the signature did not create overlaps.
- Whenever a page of the document is viewed, it should test that page.
- Whenever an item is computationally added, deleted, or moved, it should test the appropriate page.
- Whenever the details of a signature are viewed, it should test all portions of the document signed by that signature.

Global Settings

Form Globals

Form globals specify particular settings for the form and determine its physical characteristics. For example, the *bgcolor* option determines the background color of all pages in the form. Form globals appear within the *global* item in the *globalpage*, which appears at the top of a form. Options defined within a page or item can override global settings for that particular page or item.

Available Options

activated, bgcolor, border, dirtyflag, focused, fontcolor, fontinfo, formid, keypress, label, previous, printbgcolor, printfontcolor, printing, printsettings, requirements, saveformat, transmitformat, triggeritem, version, webservices, xformsmodels

Note: For descriptions, see "Details on Options and Array Elements" on page 69.

Example

This example defines settings and characteristics for the form:

```
<?xml version="1.0"?>
<XFDL xmlns="http://www.ibm.com/xmlns/prod/XFDL/7.0"
   xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
   xmlns:xforms="http://www.w3.org/2002/xforms">
   <globalpage sid="global">
      <global sid="global">
         <saveformat>application/vnd.xfdl</saveformat>
         <label>Time Sheet</label>
         <bgcolor>ivory</bgcolor>
         <fontinfo>
            <fontname>Helvetica</fontname>
            <size>10</size>
            <effect>plain</effect>
         </fontinfo>
      </global>
   </globalpage>
   ...
```

These global settings specify that:

- The form is written in XFDL version 7.0.
- All saves activated from the form should save the form as an XFDL form, unless otherwise specified in an item that initiates a save.
- The title Time Sheet should appear in the title bar of all pages, unless specified otherwise in a page global.
- All pages, toolbars and boxes should have an ivory background, unless they contain an option specifying otherwise.
- All pages and items should use a plain, Helvetica, 10-point font, unless they contain an option specifying otherwise. (Note: Labels that are parts of other items, like fields, are excluded from the fontinfo option. They are set using the labelfontinfo option, which is not available at this level.)

Usage Details

1. Define form globals within the *global* item of the *globalpage*.
2. The *global* item and the *globalpage* must always have a sid of *global*.
3. The *globalpage* follows the XFDL tag.
4. You can give the form a title that appears in the title bar by setting a global *label* option.
5. When referencing form globals from within the form, the following syntax applies:

   ```
   global.global.option[n|name]
   ```

   For example:

   ```
   global.global.bgcolor
   ```

Page Globals

Page globals specify settings (like *next* and *saveformat*) and characteristics (like *bgcolor*) for the page within which they appear. Page globals appear within a *global* item at the top of each page definition, and apply to the whole page. They can be overridden by option settings within items.

Available Options

activated, bgcolor, border, focused, focuseditem, fontcolor, fontinfo, itemfirst, itemlast, keypress, label, mouseover, next, previous, pagefirst, pageid, pagelast, pagenext, pageprevious, printbgcolor, printfontcolor, printsettings, saveformat, transmitformat

Note: For descriptions, see “Details on Options and Array Elements” on page 69.

Example

The following example shows page globals on two pages within a single form:

```
<page sid="PAGE_1">
   <global sid="global">
      <printsettings>
         <border>on</border>
      </printsettings>
      <bgcolor>gray84</bgcolor>
      <label>Administration Form 1</label>
      <next>FIELD_date</next>
   </global>
   ...

<page sid="PAGE_2">
   <global sid="global">
      <printsettings>
         <border>on</border>
      </printsettings>
      <bgcolor>#C0C0FF</bgcolor>
      <label>Administration Form 2</label>
      <next>FIELD_adminname</next>
   </global>
   ...
```

″PAGE_1″ has a medium-gray background, and directs the focus to the item called ″FIELD_date″ as soon as it opens. It assumes the rest of its settings from the form global. (If no form global exists, the page will assume the XFDL defaults.)

On ″PAGE_2″, the focus is directed to the item called ″FIELD_adminname″ as soon as the page opens. ″PAGE_2″ assumes the rest of its settings from its page global and XFDL defaults.

Usage Details

1. Page globals are defined in the *global* item at the top of a page, after the page declaration.
2. The *global* item must have a sid of *global*.
3. Page globals apply only to the page they are on.
4. Page globals are optional.
5. To specify a title to appear in the page's title bar, use the label option as a page global.
6. When referencing page global options within the form, the following syntax applies:

   ```
   pageid.global.option[n|name]
   ```

   For example:

   ```
   PAGE_1.global.bgcolor
   ```

Details on Items

Items are the basic elements of a page. The syntax of an item definition is as follows:

```
<itemType sid="itemTag">
   option definition₁
      ...
   option definitionₙ
</itemType>
```

Note:

- The itemType states the type of item to create. It must be one of the item types defined in this specification, or must be a custom item that follows the rules for custom items outlined in this specification.
- The *sid* attribute is mandatory.
- The value of each item sid must be unique in the page.

The *sid* attribute uniquely identifies an item. (See "Scope Identifiers (sid)" on page 18.) Every item sid in a page must be unique. The *ItemType* element identifies the type of item to create. (For example, <*field*...> defines the item as a field.) This section contains information about XFDL-defined item types and the options available for each.

Note: Defining an option more than once in an item's definition is not permitted. See "Details on Options and Array Elements" on page 69 for descriptions of each option type.

action

Specifies form-initiated actions that execute automatically. The actions can be any of the following types: link, replace, submit, done, display, print, refresh, pagedone, save, select, enclose, extract, remove, signature, or cancel. See section "type" on page 194 for a description of each of these actions.

Action items can be defined to occur only once or repeat at specified time intervals, and after the page opens but before the page appears. See the section on the delay option for information on timing options. *Action* items can trigger either background actions or actions involving user interaction. A form can contain only hidden items such as *action* items and operate in the background. Such forms are called daemon forms.

Available Options

activated, active, data, datagroup, delay, itemnext, itemprevious, printsettings, saveformat, transmitdatagroups, transmitformat, transmitgroups, transmititemrefs, transmititems, transmitnamespaces, transmitoptionrefs, transmitoption, transmitpagerefs, type, url, xforms:submit, xforms:trigger

Example

The following action will send a status message to the server. The transaction happens automatically every 10 minutes (600 seconds).

```
<action sid="sendStatus_action">
   <delay>
      <type>repeat</type>
      <interval>600</interval>
   </delay>
   <type>submit</type>
   <url>mailto:manager@company.com</url>
</action>
```

Usage Details

1. Repeating automatic actions is one method of creating a sparse-stated connection. It allows the form to indicate periodically to a server application that it is still running. Use the delay option to specify repetition.
2. Actions, by the form definition rules, reside on a page; therefore, actions occur only when the page is open, and repeating actions stop when the page closes. Actions defined to occur before the page displays, occur each time the page opens.

box

Specifies a rectangular box on the form. Other items may be positioned on top of boxes (using itemlocation). The purpose of *box* items is simply to add visual variety to the form.

Available Options

bgcolor, border, fontinfo, itemlocation, itemnext, itemprevious, printbgcolor, printvisible, size, visible

Example

The following example shows a typical box description. The box is 25 characters wide and 4 characters high. The background color is blue.

```
<box sid="blue_box">
   <bgcolor>blue</bgcolor>
   <size>
      <width>25</width>
      <height>4</height>
   </size>
</box>
```

Usage Details

1. To make the box more visible, assign a background color that differs from the page background color (the default).
2. When setting the *size* option of a box, the height and width of the box will be based on the average character size for the font in use (set with the *fontinfo* option). The default font, if none is specified in the page global settings, is Helvetica 8 plain.
3. If you are creating an XForms form, you should use the *pane* item rather than the *box* item. For more information about the *pane* item, refer to “pane” on page 59.

button

Specifies a click button that performs an action when clicked with the mouse or activated with the space bar when it receives the input focus. Buttons can request data from a web server, submit or cancel the form, sign the form, save it to disk, or enclose external files.

Available Options

acclabel, activated, active, bgcolor, borderwidth, coordinates, data, datagroup, focused, fontcolor, fontinfo, format, help, image, imagemode, itemlocation, itemnext, itemprevious, justify, keypress, linespacing, mouseover, next, printbgcolor, printfontcolor, printvisible, signature, signatureimage, signdatagroups, signdetails, signer, signformat, signgroups, signinstances, signitemrefs, signitems, signnamespaces, signoptionrefs, signoptions, signnamespaces, signpagerefs, size, transmitformat, transmititemrefs, transmititems, transmitnamespaces, transmitoptionrefs, transmitoptions, transmitpagerefs, type, url, value, visible, xforms:submit, xforms:trigger, xforms:upload

Examples

Submit button

Buttons that trigger form processing requests must have a *type* option setting of **submit** or **done**. The definition for such a button might look like this:

```
<button sid="submit_button">
   <value>Process Form</value>
   <fontinfo>
      <fontname>Helvetica</fontname>
      <size>18</size>
      <effect>bold</effect>
      <effect>italic</effect>
   </fontinfo>
   <type>done</type>
   <url>http://www.ibm.server.com/cgi-bin/formProcessor</url>
</button>
```

Enclosure button

This button encloses an external file in the form. The action to enclose a file is enclose. The *datagroup* option identifies the list of datagroups, or folders, in which the user can store the enclosed file. An enclose button might take the following form:

```
<button sid="enclose_button">
   <value>Enclose File</value>
   <fontinfo>
      <fontname>Helvetica</fontname>
      <size>18</size>
      <effect>bold</effect>
      <effect>italic</effect>
   </fontinfo>
   <type>enclose</type>
   <datagroup>
      <datagroupref>Images_Asia</datagroupref>
      <datagroupref>Images_Eur</datagroupref>
      <datagroupref>Images_SAmer</datagroupref>
   </datagroup>
</button>
```

This button will allow users to enclose files into one of three datagroups (folders): Images_Asia, Images_Eur, Images_SAmer.

Usage Details

1. The text displayed by the button is defined by: the *value* option if given and not empty; otherwise, the *xforms:label* if the button contains an XForms option.
2. When setting the *size* option of a button, the height and width of the button is based on the average character size for the font in use (set with the *fontinfo* option).
3. If you set the width for a button, but not the height, then the button will automatically grow to accommodate the text within the given width. In other words, the text will wrap to fit within the width specified, and the height will increase to accommodate the text.
4. If a button's image *option* points to a data item that does not exist or contains no data, then the button will display its *value* option instead.
5. If a button's *image* option points to a data item that dynamically changes its *mimedata* (but not its item sid), then the button will update the image it displays. For information on how to update an image by enclosing a new one, see the *data* option description.
6. The *format* option is available in buttons in order to force users to sign forms before submitting them.
7. If a button of type *enclose*, *extract*, *display*, or *remove* contains both a *datagroup* and a *data* option, the *data* option takes precedence.
8. There are two steps to making a signature button mandatory:
   - Assign the following elements to the *format* option: **string** and **mandatory**.
   - Set the button's *value* equal to the button's *signer* option setting.
   - Setting the *format* to **mandatory** specifies that the button must have a *value* setting that is not empty before the user submits the form. Equating the *value* to the setting of the *signer* option ensures that the only way a button's *value* is set is if somebody uses it to sign the form. (The *signer* option stores the identity of the person who signed the form using the button.)

Usage Details: Signature Buttons

1. A signature button is the means by which the user can sign a form. To make a button a signature button, set its *type* to **signature**.
2. A signature button can be set up to sign the whole form or just part of it by setting up filters on the signature, using the *signdatagroups*, *signgroups*, *signitemrefs*, *signitems*, *signnamespaces*, *signoptionrefs*, and *signoptions* options.

   Note: It is recommended that the triggeritem and coordinates options should be filtered out. These options change when a submission is triggered or when a user clicks an image button, respectively. Filtering out parts of the form that a subsequent user will change, including subsequent signatures and signature buttons and custom options that might change , should also be taken into consideration.
3. Signature buttons allow users to do the following:
   - sign the form or portion of the form the button specifies.
   - Delete their signatures (a signature can be deleted only by the user whose signature it is, and if the signature is currently valid and not signed by some other signature).
   - View the signature and view the XFDL text of what the signature applies to.

4. All option references, calculations, and other formulas in any signed portion of a form are frozen once they have been signed. Their setting will be valued at the setting they contained at the moment when the signature was created. If the user deletes the signature, however, then the formulas will become unfrozen, and will change dynamically as normal.
5. It is **strongly recommended** that you consider the Usage Details in the *signitemrefs*, *signoptionrefs*, *signnamespaces*, and *signdatagroups* filters, and avoid using the other advanced filters unless a thorough security review has been performed.

   The usual options for other buttons (i.e. *size*, *image*, *value*) can also be used with signature buttons.

cell

Populates *combobox*, *list* and *popup* items. A cell can belong to multiple comboboxes, lists and popups. See the *combobox*, *list* and *popup* item sections for information on associating cells with these items.

Cells fall into two categories according to their behavior:
- **Action cells —** These cells perform the same set of actions normally associated with buttons. This includes such things as canceling, saving, and submitting the form.
- **Select cells —** These cells provide users with a mutually exclusive set of values from which to choose. When chosen, these cells appear selected. In a list this means the cell is highlighted in some way. In a popup, the cell's label becomes the popup's label.

Available Options

activated, active, data, datagroup, group, label, itemnext, itemprevious, printsettings, saveformat, transmitdatagroups, transmitformat, transmitgroups, transmititemrefs, transmititems, transmitnamespaces, transmitoptionrefs, transmitoptions, transmitpagerefs, type, url, value

Example

The following example shows a list with three cells. To learn how to get the value of the user's selection, see Usage Details below.

```
<popup sid="CountryPopup">
   <label>Country</label>
   <group>country</group>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
</popup>
<cell sid="albCell">
   <value>Albania</value>
   <group>country</group>
   <type>select</type>
</cell>
<cell sid="algCell">
   <value>Algeria</value>
   <group>country</group>
   <type>select</type>
```

```
</cell>
<cell sid="banCell">
   <value>Bangladesh</value>
   <group>country</group>
   <type>select</type>
</cell>
```

Usage Details

1. Use the *type* option to establish a cell's behavior. Select cells have a type of *select* (the default type).
2. Cells can have both *value* and *label* options. These options affect the form differently depending on whether the cell is linked to a *combobox*, a *popup*, or a *list* item. In general, the *label* of the cell will be displayed as a choice, while the *value* of the cell will be displayed if that cell is selected. For more information, refer to the appropriate item type.
3. Cells take their color and font information from the *combobox*, *list* and *popup* items with which they are associated. In this way, a cell's appearance can vary according to the list the user is viewing.
4. To get the value of a cell that a user has selected from a list, it is necessary to dereference it in the following manner:

   ```
   page_tag.list_tag.value->value
   ```

   For example:

   ```
   page1.countryPopup.value->value
   ```

   When a user selects a cell from a list, the item sid of the cell is stored as the value of the list. Hence the dereference syntax.
5. Starting with version 4.5, items of type *cell* are the only items that can be filtered using the *signgroups* and *transmitgroups* options.

check

Provides a simple check box to record a selected or not selected answer from a user. A selected check box appears filled while a deselected box appears empty.

The exact appearance of the check box is platform-dependent; but the shape is rectangular. The check box appears as a normal check box for the users of each platform.

Available Options

acclabel, active, bgcolor, focused, fontcolor, fontinfo, help, itemlocation, itemnext, itemprevious, keypress, label, labelbgcolor, labelborder, labelfontcolor, labelfontinfo, mouseover, next, previous, printbgcolor, printfontcolor, printlabelbgcolor, printlabelfontcolor, printvisible, readonly, saveformat, size, suppresslabel, value, visible, xforms:input

Example

This *value* option setting in this check box is **on**, so the check box will appear selected when it displays. The item's label is Activate Health Plan, and the label will display in a Times 14 Bold font colored blue.

```
<check sid="healthPlan_check">
   <value>on</value>
   <label>Active Health Plan</label>
   <labelfontinfo>
      <fontname>Times</fontname>
```

```
         <size>14</size>
         <effect>bold</effect>
      </labelfontinfo>
      <labelfontcolor>blue</labelfontcolor>
   </check>
```

Usage Details

1. The *value* option setting indicates the user's answer. If the user selects or checks the check box, the *value* option contains **on**, otherwise it contains **off**. The default value is **off**.
2. Check boxes do not belong to groups like radio buttons - each check box may be turned on or off independently of the others.
3. The *label* option defines the label for the check box. The label appears above the check box and aligned with the box's left edge. There is no default label.
4. When setting the *size* option of a check box, the height and width of the bounding box will be based on the average character size for the font in use (set with the *fontinfo* option).
5. The *fontcolor* option determines the color of the check box fill pattern (defaults to **red**).
6. For *check* items that contain an *xforms:input* option, the **on** or **off** value is translated into **true** or **false**, respectively, when stored in the data model.

combobox

Comboboxes act like a hybrid of a field and a popup. Unopened, a combobox with a label occupies the same space as two labels, and a combobox without a label occupies the same space as a single label. After a user chooses a cell, the combobox closes (that is, returns to its unopened state).

If none of the cells are appropriate, the user can type other information into the combobox. When information is typed in, it is stored in the *value* option of the combobox. When a cell is selected, the *value* option stores the *value* of that cell, so unlike other list items (popup and list), dereferencing is not necessary.

A combobox's label appears above the *combobox* item.

Available Options

acclabel, activated, active, bgcolor, borderwidth, focused, fontcolor, fontinfo, format, group, help, itemlocation, itemnext, itemprevious, keypress, label, labelbgcolor, labelborder, labelfontcolor, labelfontinfo, mouseover, next, previous, printbgcolor, printlabelbgcolor, printfontcolor, printlabelfontcolor, printvisible, readonly, size, suppresslabel, value, visible, xforms:input, xforms:select1

Example

This is an example of a combobox containing a set of selections allowing users to choose a color.

```
   <combobox sid="CATEGORY_POPUP">
      <group>combo_Group</group>
      <label>Choose a Color:</label>
   </combobox>
```

The default label is "Choose a Color:". This will display above the combobox. Until the user types in something or makes a selection, the field area of the combobox will be blank.

These are the cells that make up the combobox. They are select cells and they belong to the same group as the combobox: combo_Group.

```
<cell sid="RED_CELL">
   <group>combo_Group</group>
   <type>select</type>
   <value>Red</value>
</cell>
<cell sid="WHITE_CELL">
   <group>combo_Group</group>
   <type>select</type>
   <value>White</value>
</cell>
<cell sid="BLUE_CELL">
   <group>combo_Group</group>
   <type>select</type>
   <value>Blue</value>
</cell>
```

Usage Details

1. Place cells in a combobox by creating a group for the combobox and assigning cells to the group. Create a group using the *group* option in the combobox definition. Assign cells to the group using the *group* option in the cell definition.
2. When first viewed, a combobox will display its *value*. If no value is set, the combobox will be empty.
3. The *label* option sets the text displayed above the item, as with a field.
4. When a combobox is opened, its list displays:
   - The *label* of each cell.
   - The *value* of each cell that does not have a *label*.
5. When a cell is selected from the combobox's list, the following occurs:
   - If the cell is of *type* **select**, the combobox's *value* is set to the *value* of the selected cell and the combobox displays that *value*.
   - If the cell is of any other *type*, the appropriate action for the type is taken and the combobox's *value* is set to empty.
6. When a value is typed into a combobox (rather than selected from the list), the combobox's *value* is set to the typed value.
7. The *label* option for each cell is used to create a long form for each choice. For example, a cell might have a *label* of "United States of America" and a *value* of "USA". The long form is displayed in the combobox's list, but once that cell is selected, the short form is displayed by the popup.
8. *Combobox*, *popup*, and *list* items with the same group reference display the same group of cells.
9. Unlike popups and lists, comboboxes do not need to be dereferenced in order to obtain the *value*.
10. When setting the *size* option of a combobox, the height and width of the popup will be based on the average character size for the font in use (set with the *fontinfo* option).
11. If you set the *readonly* option to **on**, the combobox will refuse all input, although it will function normally otherwise and formulas will still be able to change the *value*.

12. When a *format* is applied to a combobox, the formatting will be applied to the *value* of each cell linked to the combobox. Those cells that fail the check will be flagged or filtered. Those cells that pass the check will have their *value* replaced with a formatted value. See the *format* option for more information.

    If any two combobox, list, or popup items use the same set of cells, they must apply the same formatting.
13. To make a combobox that displays a calendar widget, create a combobox with no cells and give it a date function.

data

Stores an information object such as an image, a sound, or an enclosed file in an XFDL form. Whenever any of these objects are are added to a form, the data that describes the object is stored in a *data* item. A *data* item can only store the data from a single object. Data in *data* items must be encoded in **base64** format.

Data items are created automatically when files are enclosed in a form. Enclose files using items with a *type* option setting of **enclose**.

Available Options

datagroup, filename, itemnext, itemprevious, mimedata, mimetype

Example

This is an example of a *data* item.

```
<data sid="Supporting_Documents_1">
   <mimetype>text/plain</mimetype>
   <filename>HelloWorld.txt</filename>
   <mimedata encoding="base64">
       SGVsbG8sIHdvcmxkIQ==
   </mimedata>
   <datagroup>
      <datagroupref>Supporting_Documents</datagroupref>
   </datagroup>
</data>
```

Usage Details

1. Stores the data in the *mimedata* option, and the data's MIME type in the *mimetype* option.
2. If a *data* item contains a *datagroup* option, it can be associated with one or more other data items. *Data* items with a *datagroup* option are not replaced if a button or cell of type **enclose** point to new *data* items. The new *data* items simply become members of the same *datagroup*. Additionally, buttons and cells with the same *datagroup*option can access the contents of the *data* item.
3. If a button or cell of type **enclose** contains a *data* option that points to a *data* item (as opposed to using the *datagroup* option), then special rules apply to the *data* item's behavior. If a user encloses a new *data* item using that button, the new information overwrites the old. For example, if the *data* item originally contained a jpeg image of a dog, and then a user enclosed a png image of a house, then the *data* item's *mimedata*, *mimetype*, and *filename* options update themselves to contain the information about the house image.
4. Starting with XFDL version 4.5, *data* items are the only items that can be filtered using the *signdatagroups* and *transmitdatagroups* options.

field

The *field* item creates a text area where users can display and enter one or more lines of data. The field's characteristics determine the number of lines, the width of each line, and whether the field is scrollable.

Field data can be protected from modification, made to display in the system password format (typically, hidden from view), and forced to conform to data type and formatting specifications.

Available Options

acclabel, active, bgcolor, borderwidth, focused, fontcolor, fontinfo, format, help, itemlocation, itemnext, itemprevious, justify, keypress, label, labelbgcolor, labelborder, labelfontcolor, labelfontinfo, mouseover, next, previous, printbgcolor, printlabelbgcolor, printfontcolor, printlabelfontcolor, printvisible, readonly, rtf, scrollvert, scrollhoriz, size, suppresslabel, texttype, value, visible, writeonly, xforms:input, xforms:secret, xforms:textarea

Example

This is an example of a single line *field* item that allows 20 characters of input. An initial value of 23000 has been defined for the field. When the form appears, the field will contain this value.

```
<field sid="income_field">
   <label>Annual income</label>
   <value>23000</value>
   <size>
      <width>20</width>
      <height>1</height>
   </size>
   <fontinfo>
      <fontname>Courier</fontname>
      <size>12</size>
      <effect>plain</effect>
   </fontinfo>
   <labelfontinfo>
      <fontname>Helvetica</fontname>
      <size>12</size>
      <effect>plain</effect>
   </labelfontinfo>
   <labelfontcolor>blue</labelfontcolor>
</field>
```

Usage Details

1. When setting the *size* option of a field, the width of the field will be based on the average character size for the font in use (set with the *fontinfo* option) and the height will exclude the external font leading.
2. Use the *readonly* option to create a readonly field.
3. Use the *writeonly* option to create a write only field. This is useful for passwords.
4. The *format* option specifies the data type of the field's data. It also contains flags that allow the application of specified edit checks and formatting to the data.
5. The *label* option defines the field's label. The label is placed above the field and aligned with the field's left edge.

6. The *scrollvert* and *scrollhoriz* options govern a field's scrolling characteristics. They must be set to always to permit scrolling. With scrolling enabled, scroll bars display along the bottom (horizontal scrolling) and right (vertical scrolling) edges of the field.
7. The *texttype* option determines whether a field contains plain text or rich text. Note that you cannot dynamically change a rich text field into a plain text field.
8. When using XForms, the following rules apply:
   - To create a single line field, use *xforms:input*.
   - To create a multi-line field, use *xforms:textarea*.
   - To create a rich text field, use *xforms:textarea*. In this case, it is the *rtf* option that is bound to the data model (as opposed to the *value* option). Rich text fields do not support the use of *xforms:input* or *xforms:secret*.
   - To create a write only field, use *xforms:secret*.

help

Defines a help message that can be used to support various external items in the form. Separate *help* items can be created for every item supported, or one *help* item can be used to support several items.

Available Options

active, itemnext, itemprevious, value

Example

This is an example of a *button* item that links to a *help* item using the *help* option:

```
<button sid="fullPicture_button">
   <value>View Full-Sized Picture</value>
   <help>button_help</help>
   <fontinfo>
      <fontname>Times</fontname>
      <size>14</size>
      <effect>plain</effect>
   </fontinfo>
   <type>link</type>
   <url>http://www.ibm.server.com/application/fullPic.frm</url>
</button>
```

The following example shows the *help* item referred to in the *button* item definition. The contents of the *value* option are presented as the help message when the user asks for help with the button.

```
<help sid="button_help">
   <value>
      Pressign this button will bring a full-sized image in a form
      down to your viewer.
   </value>
</help>
```

Usage Details

1. The *help* item's *value* option contains the help message text.
2. The link between the *help* item and the supported item is created by the *help* option in the supported item's definition. The *help* option contains the *help* item's item reference.

label

Defines a static text message or an image to display on the form. If both an image and a text message are defined for the label, the image takes precedence in viewers able to display images.

Available Options

active, bgcolor, borderwidth, fontcolor, fontinfo, format, help, image, imagemode, itemlocation, itemnext, itemprevious, justify, linespacing, printbgcolor, printfontcolor, printvisible, size, suppresslabel, value, visible, xforms:output

Example

This is an example of a multiple-line text label:

```
<label sid="RHYME LABEL">
   <value>
      Little miss Muffet Sat on her tuffet,
      Eating her curds and whey.
      When along came a spider, who sat down beside her,
      and frightened miss Muffet away!
   </value>
   <fontinfo>
      <fontname>Times</fontname>
      <size>16</size>
      <effect>italic</effect>
   </fontinfo>
   <justify>right</justify>
</label>
```

Usage Details

1. The text displayed by a label is defined by: the *xforms:output* option, if given; otherwise the *value* option.
2. To define an image for a label, use the *image* option.
3. To create a multiple line text message, add line breaks to the message text. Use the escape sequence '\n' to indicate a line break.
4. When setting the *size* option of a label, the height and width of the label will be based on the average character size for the font in use (set with the *fontinfo* option).
5. If you set the width for a label, but not the height, then the label will automatically grow to accommodate the text within the given width. In other words, the text will wrap to fit within the width specified, and the height will increase to accommodate the text.
6. If a label's image *option* points to a data item that does not exist or contains no data, then the label will display its *value* option instead.
7. If a label contains both a *value* option and an *xforms:output* option, then the *xforms:output* overrides the *value*.
8. If a label contains an *xforms:output* with an *xforms:label*, the value of the *xforms:label* is concatenated with the value of the *xforms:output* (always label first). This allows you to create a decoration for the text that is displayed. For example, you might want your label to read "Total: 200", where the "Total:" is provided by an *xforms:label* and the "200" is provided by the *xforms:output*. Note that the *xforms:label* can only contain text and is not affected by the *format* option.

9. If a label is decorated by an *xforms:label*, and the *justify* option it set to right justify, then the value of the label item is right justified but the additional text from the *xforms:label* remains left justified. This allows you to create space between the *xforms:label* and the value, as shown:

```
Total:    200
```

10. If a label contains a *suppresslabel* option, it prevents the *xforms:label* from being displayed.
11. If a label's *image* option points to a *data* item that dynamically changes its *mimedata* (but not its item sid), then the label will update the image it displays. For information on how to update an image by enclosing a new one, see the *data* option description.
12. Set the image display behavior with the *imagemode* option.
13. The label's background color defaults to being transparent - and thus the label will allow whatever item it is over to show through. For example, it is possible to place a label over another label holding an image. The image will show through the top label.
14. Label contents (if text) can be formatted using *format* flags (see section "format" on page 90).
15. The *suppresslabel* option suppresses the label, so that it is not displayed.

line

Draws a simple vertical or horizontal line on the form. Lines are useful for visually separating parts of a page.

Available Options

fontcolor, fontinfo, itemlocation, itemnext, itemprevious, printfontcolor, printvisible, size, thickness, visible

Example

This is an example of a horizontal line with a thickness of five pixels:

```
<line sid="BLUE_LINE">
   <size>
      <width>40</width>
      <height>0</height>
   </size>
   <thickness>5</thickness>
</line>
```

Usage Details

1. Specify the dimensions of a line using the *size* and *thickness* options. The *size* option determines whether the line is vertical or horizontal. If the horizontal dimension is set to zero, then the line is vertical. If the vertical dimension is set to zero, then the line is horizontal. Size is calculated in characters.
2. The *thickness* option determines how thick the line will be. Thickness is calculated in pixels.
3. The *fontinfo* option information is used when calculating the line's size. The *size* option's unit of measurement is characters; therefore, choice of font can affect the size. See the *size* option for more information.
4. The *fontcolor* option defines the color of the line.

list

Creates a list that allows the user to select one or more choices. Additionally, the choices can be set to triggers actions, such as saving or submitting the form.

The entries in the list are *cell* items. Selections are cells with a *type* option setting of **select**. Actions are cells with any other *type* option setting.

Available Options

acclabel, active, bgcolor, borderwidth, focused, fontcolor, fontinfo, format, group, help, itemlocation, itemnext, itemprevious, keypress, label, labelbgcolor, labelborder, labelfontcolor, labelfontinfo, mouseover, next, previous, printbgcolor, printlabelbgcolor, printfontcolor, printlabelfontcolor, printvisible, readonly, size, suppresslabel, value, visible, xforms:select, xforms:select1

Example

This is an example of a list containing three actions: submit form, save form, and cancel form.

```
<list sid="MAINMENU_LIST">
   <group>list_Group</group>
   <label>Options Menu</label>
   <labelfontcolor>blue</labelfontcolor>
   <size>
      <width>3</width>
      <height>20</height>
   </size>
</list>
```

These are the cells that make up the list. They are action cells and they belong to the same group as the list: **list_Group**.

```
<cell sid="SUBMIT_CELL">
   <group>list_Group</group>
   <type>submit</type>
   <url>http://www.ibm.server.com/cgi-bin/processForm</url>
   <value>Submit Form</value>
</cell>
<cell sid="SAVE_CELL">
   <group>list_Group</group>
   <type>saveas</type>
   <value>Save Form</value>
</cell>
<cell sid="CANCEL_CELL">
   <group>list_Group</group>
   <type>cancel</type>
   <value>Cancel this Form</value>
</cell>
```

Usage Details

1. Users can only select one choice from a *list* item, unless the item includes the *xforms:select* option, in which case they can select any number of choices.
2. Create non-XForms lists by using the *group* option to link the *list* to a number of *cell* items.
3. When first viewed, a list displays it's *label* above the item (as with a field) and as many choices as its size allows.
4. The choices in the list display:
   - The *label* of each cell.

- The *value* of each cell that does not have a *label*.

5. When a cell is selected from the list, the following occurs:
   - If the cell is of *type* **select**, the list's *value* is set to the name of the selected cell and that cell is highlighted.
   - If the cell is of any other *type*, the appropriate action for the type is taken, the list's *value* is set to empty, and the selected cell is highlighted.
6. The *label* option for each cell is used to create a long form and and abbreviated form for each choice. For example, a cell might have a *label* of "United States of America" and a *value* of "USA". The long form is displayed in the list, but the short form available as the cell's *value*.
7. To get the value of a cell that a user has selected from a list, it is necessary to dereference it in the following manner:
   ```
   page_tag.list_tag.value->value
   ```
   For example:
   ```
   compute="page1.countryPopup.value->value"
   ```
8. *List*, *combobox* and *popup* items with the same group reference display the same group of cells.
9. The *value* option will contain one of the following:
   - The item reference of the most recently chosen cell, if the cell was of type **select**.
   - Nothing, if the cell most recently chosen was of any type other than **select**.
10. When setting the *size* option of a list, the height and width of the list will be based on the average character size for the font in use (set with the *fontinfo* option).
11. A vertical scroll bar will appear beside the list if the number of cells is greater than the height (defined with the *size* option) of the list.
12. When a *format* is applied to a list, the formatting will be applied to the value of each cell linked to the list. Those cells that fail the check will be flagged or filtered. Those cells that pass the check will have their *value* replaced with a formatted *value*. See the *format* option for more information.
13. If any two *combobox*, *list*, or *popup* items use the same set of cells, they must apply the same formatting.

popup

Creates a popup menu from which users can make selections (as in a list of names) and trigger actions (such as enclosing files and submitting the form). A popup can contain both selections and actions.

The entries in the popup are *cell* items. Selections are cells with a *type* option setting of **select**. Actions are cells with any other *type* option setting.

Popups act like a hybrid of a label, a button, and a list. Unopened, a popup occupies only the space required for its label. Open, the popup displays a list of selections and actions. After a user chooses a selection or an action, the popup closes (that is, returns to its unopened state). A popup's label displays inside the *popup* item.

Available Options

acclabel, activated, active, bgcolor, borderwidth, focused, fontcolor, fontinfo, format, group, help, itemlocation, itemnext, itemprevious, justify, keypress, label, mouseover, next, previous, printbgcolor, printfontcolor, printvisible, readonly, size, value, visible, xforms:select1

Example

This is an example of a *popup* list containing a set of selections allowing users to choose a category.

Here is the *popup* definition. The default label is ″Choose a Category:″. This will display until a user makes a selection. Afterwards, the cell's value will display as the label.

```
<popup sid="CATEGORY_POPUP">
   <group>popup_Group</group>
   <label>Choose a Category:</label>
</popup>
```

These are the cells that make up the popup. They are select cells and they belong to the same group as the popup: **popup_Group**.

```
<cell sid="HISTORY_CELL">
   <group>popup_Group</group>
   <type>select</type>
   <value>World History</value>
</cell>
<cell sid="SCIENCE_CELL">
   <group>popup_Group</group>
   <type>select</type>
   <value>Physical Sciences</value>
</cell>
<cell sid="MUSIC_CELL">
   <group>popup_Group</group>
   <type>select</type>
   <value>Music</value>
</cell>
```

Usage Details

1. Place cells in a popup by creating a group for the popup and assigning cells to the group. Create a group using the *group* option in the popup definition. Assign cells to the group using the *group* option in the cell definition.
2. When first viewed, a popup will display its *label* if no *value* is set. If there is no *value* or *label*, the popup will be empty.
3. When a popup is opened, its list displays:
   - The *label* of each cell.
   - The *value* of each cell that does not have a *label*.
4. When a cell is selected from the popup's list, the following occurs:
   - If the cell is of *type* **select**, the popup's *value* is set to the name of the selected cell and the popup displays the cell's *value*.
   - If the cell is of any other *type*, the appropriate action for that type is taken, the popup's *value* is set to empty, and the popup displays its *label* option
5. The *label* option for each cell is used to create a long form for each choice. For example, a cell might have a *label* of ″United States of America″ and a *value* of ″USA″. The long form is displayed in the popup's list, but once that choice is selected, the short form is displayed by the popup.

6. To get the *value* of a cell that a user has selected from a list, it is necessary to dereference it in the following manner:

   ```
   page_tag.list_tag.value->value
   ```

   For example:

   ```
   compute="page1.countryPopup.value->value"
   ```

   When a user selects a cell from a list, the item sid of the cell is stored as the *value* of the list. Hence the dereference syntax.
7. *Popup*, *combobox* and *list* items with the same group reference display the same group of cells.
8. When setting the *size* option of a popup, the height and width of the popup will be based on the average character size for the font in use (set with the *fontinfo* option).
9. When a *format* is applied to a popup, the formatting will be applied to the value of each cell linked to the popup. Those cells that fail the check will be flagged or filtered. Those cells that pass the check will have their *value* replaced with a formatted *value*. See the *format* option for more information.
10. If any two comboboxes, lists, or popups use the same set of cells, they must apply the same formatting.

radio

Intended for use with one or more other *radio* items. A group of radio buttons presents users with a set of mutually exclusive choices. Each radio button represents one choice the user can make.

There is always one selected radio button in the group. As well, since radio buttons present a mutually exclusive set of choices, only one radio button in a group can be selected. When a user chooses a radio button, that radio button becomes selected.

A selected radio button appears filled in some way. All other radio buttons in the group appear empty.

Available Options

acclabel, active, bgcolor, focused, fontcolor, fontinfo, group, help, itemlocation, itemenxt, itemprevious, keypress, label, labelbgcolor, labelborder, labelfontcolor, labelfontinfo, mouseover, next, previous, printbgcolor, printlabelbgcolor, printfontcolor, printlabelfontcolor, printvisible, readonly, size, suppresslabel, value, visible

Example

This example shows a group of three radio buttons. The first radio button is the initial choice: the *value* option setting is on. The buttons all belong to the group **search_Group**.

```
<radio sid="NAME_RADIO">
   <value>on</value>
   <group>search_Group</group>
   <label>Search by Name</label>
</radio>
<radio sid="NUMBER RADIO">
   <group>search_Group</group>
   <label>Search by Number</label>
</radio>
```

```
<radio sid="OCCUPATION RADIO">
   <group>search_Group</group>
   <label>Search by Occupation</label>
</radio>
```

As shown here, only the chosen radio button needs to have a *value* option setting. The remaining radio buttons will receive the (default) value setting of **off**.

Usage Details

1. Group radio buttons by assigning them to the same group. Do this by including the *group* option in each radio button's definition, and using the same group reference in each case.
2. The *value* option contains the status indicator. It can be either **on** or **off**. The value **on** indicates a status of chosen. The value **off** indicates a status of not chosen. The default status is not chosen.
3. When the form opens, if no radio button has the status chosen, then the last radio button defined for the group becomes chosen. If multiple radio buttons are chosen, then only the last 'chosen' radio button retains that status.
4. The *label* option defines a label to appear above the radio button and aligned with its left edge.
5. When setting the *size* option of a radio button, the height and width of the bounding box will be based on the average character size for the font in use (set with the *fontinfo* option).
6. The *fontcolor* option determines the color of the radio button fill pattern (defaults to red).
7. For *radio* items that contain an *xforms:input* option, the **on** or **off** value is translated into **true** or **false** respectively when stored in the data model.

signature

Contains a signature and the data necessary to verify the authenticity of a signed form. It is created by a form viewer or other program when a user signs a form (usually using a signature button). The signature item contains an encrypted hash value that makes it impossible to modify the form without changing the hash value that the modified form would generate. To verify, one can generate the hash value and then see if it matches the one in the signature.

Available Options

colorinfo, excludedmetadata, fullname, itemnext, itemprevious, layoutinfo, mimedata, signature, signdatagroups, signdetails, signer, signformat, signgroups, signinstances, signitemrefs, signitems, signnamespaces, signoptionrefs, signoptions, signpagerefs

Example

This example shows a signature item below the signature button that created it.

```
<button sid="empSigButton">
   <type>signature</type>
   <value compute="signer"></value>
   <signer>Jane D Smith, jsmith@insurance.com</signer>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <signformat>application/vnd.xfdl;
      csp="Microsoft Base Cryptographic Provider v1.0";
      csptype=rsa_full;hashalg=sha1
   </signformat>
   <signoptions>
      <filter>omit</filter>
      <optiontype>triggeritem</optiontype>
      <optiontype>coordinates</optiontype>
   </signoptions>
   <signitemrefs>
      <filter>omit</filter>
      <itemref>PAGE1.mgrSigButton</itemref>
      <itemref>PAGE1.admSigButton</itemref>
      <itemref>PAGE1.empsignature</itemref>
      <itemref>PAGE1.mgrsignature</itemref>
      <itemref>PAGE1.admsignature</itemref>
   </signitemrefs>
   <signature>empsignature</signature>
</button>
...
<signature sid="empsignature">
   <signformat>application/vnd.xfdl;
      csp="Microsoft Base Cryptographic Provider v1.0";
      csptype=rsa_full;hashalg=sha1
   </signformat>
   <signer>Jane D Smith, jsmith@insurance.com</signer>
   <fullname>
      "Verisign, Inc.", Verisign Trust Network,
      "www.verisign.com/repository/RPA Incorp. by
      Ref.,LIAB.LTD(c)98", Persona Not Validated,
      Digital ID Class 1 - Microsoft, Jane D
      Smith, jsmith@insurance.com
   </fullname>
   <signature>PAGE1.empsignature</signature>
   <signitemrefs>
      <filter>omit</filter>
      <itemref>PAGE1.mgrSigButton</itemref>
      <itemref>PAGE1.admSigButton</itemref>
      <itemref>PAGE1.empsignature</itemref>
      <itemref>PAGE1.mgrsignature</itemref>
      <itemref>PAGE1.admsignature</itemref>
   </signitemrefs>
   <!-- The items listed above MUST have itemlocation options with
      absolute and extent as the last settings in order for the filter
      below to be sufficient in terms of security -->
   <signoptionrefs>
      <filter>keep</filter>
      <optionref>PAGE1.mgrSigButton.itemlocation</optionref>
      <optionref>PAGE1.admSigButton.itemlocation</optionref>
      <optionref>PAGE1.empsignature.itemlocation</optionref>
      <optionref>PAGE1.mgrsignature.itemlocation</optionref>
      <optionref>PAGE1.admsignature.itemlocation</optionref>
   </signoptionrefs>
   <signoptions>
      <filter>omit</filter>
      <optiontype>triggeritem</optiontype>
      <optiontype>coordinates</optiontype>
   </signoptions>
   <mimedata encoding="base64">
      MIIFMgYJKoZIhvcNAQcCoIIFIzCCBR8CAQExDzANBgkgAQUFADALB\ngk
      qhkiG9w0BBwGgggQZMCA36gAwSRiADjdhfHJl6hMrc5DySSP+X5j\nANf
      BGSOI\n9w0BAQQwDwYDVQQHEwhJbn<Rlcm5ldDEXMBUGA1UEChM\nOVmV
      yaVNpZ24sIEluYy4xNDAKn1ZlcmlTaWduIENsYXNzIDEgQ0Eg\nLSJbmR
      dWFsIFN1YnNjcmliyZXIwHhcNOTgwMTI3MwMDAwOTgwM\M1OTU5WjCCAR
      ExETA
   </mimedata>
</signature>
```

Usage Details

1. When a user signs a form using a signature button, the viewer creates the *signature* item as specified in the button's *signature* option. The viewer also associates the signature with the signature button, using the signature's *signature* option.
2. When a user signs a form, the *signer*, *signformat*, *signgroups*, *signitemrefs*, *signitems*, *signnamespaces*, *signoptionrefs*, and *signoptions* options are copied from the button description to the signature description.
3. While *signformat* is not mandatory for *button* items, it is mandatory for signature items.
4. A copy of the XFDL description of the form or portion of the form that is signed is included in the signature's *mimedata* option. This data is encrypted using the hash algorithm specified in the button's *signformat* option.
5. signatures always filter out the *mimedata* option for their own signature item, regardless of the signature filter settings. This is done because the *mimedata* is not populated with the signature information until after the signature has been applied. (In other words, the signature can't include itself because it hasn't been generated yet.)
6. When a program checks a signed form, it compares the data in the *mimedata* option with that of the portion of the form that is apparently signed. If the descriptions match, then the signature remains valid. If the signatures do not match, the signature breaks, and the user is prompted.
7. An attempt to create a signature will fail if:
   - The item named by the signature button's *signature* option already exists.
   - The signature button is already signed by any signature in the form.
   - The signer's private key is unavailable for signing.
8. Filters can be used to indicate which items and options to keep and to omit. The explicit and implicit settings of an existing filter take precedence over an implication that might be drawn from a non-existing filter. Set up these filters in the signature button description.
9. To use certain types of digital signatures (CryptoAPI or Netscape, for example), it is necessary for the user to obtain a digital signature certificate. Other types of digital signatures require the user to have a pen/pad device installed on the user's computer.

spacer

Creates space between items on a form. It can be any size specified. It is invisible.

Available Options

fontinfo, itemlocation, itemnext, itemprevious, label, linespacing, size

Example

This example shows a *spacer* item that uses the *size* option to define the amount of space it will occupy.

```
<spacer sid="THREE_SPACER">
   <size>
      <width>1</width>
      <height>3</height>
   </size>
</spacer>
```

This example shows the *spacer* item that uses a label to define the amount of space it will occupy. This sizing technique is useful when creating a spacer that is exactly the same size as a real label on the form.

```
<spacer sid="WELCOME_SPACER">
   <label>Welcome to Information Line</label>
</spacer>
```

Usage Details

1. A spacer can be sized either by giving it length and width dimensions (using *size*), by expanding the default size using the *itemlocation* option or by giving it a *label*. If a *label* is used, the spacer equals the size of the text typed into the label. The label does not appear; it is simply used to determine the spacer's size.
2. When setting the *size* option of a spacer, the height and width of the spacer will be based on the average character size for the font in use (set with the *fontinfo* option).
3. If you set the width for a spacer, but not the height, then the spacer will automatically grow to accommodate the text within the given width. In other words, the text will wrap to fit within the width specified, and the height will increase to accommodate the text.

toolbar

Allows the definition of a toolbar for a page. A toolbar is a separate and fixed area at the top of the page. It functions much like a toolbar in a word processing application. Items placed in the toolbar are always visible at the top of the form, no matter what portion of the page they are viewing.

The toolbar is visible no matter what portion of the page body is visible. However, if the toolbar is larger than half the form window, it is necessary to scroll to see everything it contains.

Available Options

bgcolor, itemnext, itemprevious, mouseover

Example

This example shows a toolbar that contains a label.

Here is the toolbar definition:

```
<toolbar sid="TOOL_BAR">
   <bgcolor>cornsilk</bgcolor>
</toolbar>
```

Here is the label that will appear in the toolbar.

```
<label sid="COMPANY_NAME">
   <value>My Company</value>
   <itemlocation>
      <within>TOOL_BAR</within>
   </itemlocation>
</label>
```

Usage Details

1. The background color of the toolbar becomes the default background color for items in the toolbar.
2. Add items to the toolbar using the **within** modifier of the *itemlocation* option. Code the *itemlocation* option in each included item's definition.
3. Each page can contain only one toolbar.
4. If an XForms item is placed in the toolbar, then any items controlled by it are also placed in the toolbar, regardless of individual declarations. For example, if a table were placed in the toolbar, then all items in that table would also appear in the toolbar.

<custom item>

Allows form designers to add application specific information to the form definition. This is useful when submitting forms to applications requiring non-XFDL information. An example of non-XFDL information might be an SQL query statement.

Available Options

All XFDL options, the *xforms:input* or *xforms:textarea* options, and any custom options can be used with custom items.

Example

This is an example of a custom item definition. It includes both an XFDL and a custom option.

```
<custom:event xfdl:sid="STATUS_EVENT"
   xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom">
   <xfdl:active>off</xfdl:active>
   <custom:ID>UF45567/home/users/preferences01</custom:ID>
</custom:event>
```

Usage Details

1. Custom items must be in a non-XFDL namespace.
2. Custom items can also change the default namespace from the XFDL namespace URI in order to eliminate excess prefixes within the desired namespace.
3. Custom items may omit the XFDL scope identifier, but a *sid* attribute in the XFDL namespace must be provided in order to reference the custom item's content with the XFDL compute system. Since default namespaces are not applied to attributes, the *sid* attribute must still be qualified by a namespace prefix associated with the XFDL namespace URI.
4. Both XFDL options and custom options within the custom item can have computed values by using the XFDL compute attribute, which must be qualified with a namespace prefix associated with the XFDL namespace URI.

Details on XForms Items

XForms items are only required when you are creating an XFDL form that contains an XForms data model. These items are used to contain XForms constructs that do not normally exist in XFDL, such as tables and checkgroups that automatically repeat elements based on the structure of the data model.

XForms items follow the same syntax rules as XFDL items, and are described in detail in the following sections.

checkgroup

Creates a group of check boxes. This is useful if you want to create a list of options and allow the user to select some of them. You can configure a *checkgroup* to allow only one selection, or to allow any number of selections.

Each check box appears as an empty box that is filled with a marker, such as a check mark or an X, when selected.

Note that *checkgroup* items are only valid for XForms forms. If you are not using XForms, use the *check* item instead.

Available Options

acclabel, active, bgcolor, border, focused, format, help, itemlocation, itemnext, itemprevious, label, labelbgcolor, labelborder, labelfontcolor, labelfontinfo, mouseover, next, previous, printbgcolor, printlabelbgcolor, printlabelfontcolor, printvisible, readonly, suppresslabel, value, visible, xforms:select, xforms:select1

Example

The following code creates a *checkgroup* with three choices: US dollars, Canadian dollars, and Euros. The choices themselves are defined within the *xforms:select* option.

```
<checkgroup sid="currency">
   <xforms:select ref="currency" appearance="full">
      <xforms:label>Select the currencies you accept:</xforms:label>
      <xforms:item>
         <xforms:label>US Dollars</xforms:label>
         <xforms:value>USD</xforms:value>
      </xforms:item>
      <xforms:item>
         <xforms:label>CDN Dollars</xforms:label>
         <xforms:value>CDN</xforms:value>
      </xforms:item>
      <xforms:item>
         <xforms:label>Euro</xforms:label>
         <xforms:value>Euro</xforms:value>
      </xforms:item>
   </xforms:select>
</checkgroup>
```

Alternatively, you could create the choices in your data model as follows:

```
<xforms:instance xmlns="">
   <data>
      <currency/>
      <choice show="US Dollars">USD</choice>
      <choice show="CDN Dollars">CDN</choice>
      <choice show="Euros">Euro</choice>
   </data>
</xforms:instance>
```

In this case, you would use the *xforms:select* option to link to those choices, as illustrated by the following checkgroup:

```
<checkgroup sid="currency">
   <xforms:select ref="currency" appearance="full">
      <xforms:label>Select the currencies you accept:</xforms:label>
      <xforms:itemset nodeset="instance('currency')/choice">
         <xforms:label ref="@show"></xforms:label>
         <xforms:value ref="."></xforms:value>
      </xforms:itemset>
   </xforms:select>
</checkgroup>
```

Usage Details

1. To allow the user to select any number of choices in the *checkgroup*, use an *xforms:select* option. To limit the user to selecting one choice from the *checkgroup*, use an *xforms:select1* option.
2. The check boxes in a *checkgroup* item are arranged vertically by default (that is, each check appears immediately below the previous choice).
3. To arrange a *checkgroup* item in another manner, use the <xforms:extension> element to add an *itemlocation* to each check in the group. For example, you might set the checks to appear one after another horizontally with the following *itemlocation*:

   ```
   <xforms:extension>
      <itemlocation>
         <after compute="itemprevious"/>
      </itemlocation>
   </xforms:extension>
   ```

   For more information about the <xforms:extension> element, refer to "xforms:select" on page 227 or "xforms:select1" on page 232.
4. Each check box in a *checkgroup* can have its own label. These labels are displayed immediately to the right of each check box (rather than above each check box, as with the *check* item).
5. The single node binding in the *xforms:select* or *xforms:select1* option creates a link between the *value* option for the *checkgroup* and the bound element in the data model, so that they share data. When the user makes a selection, the *xforms:value* of that selection is stored in both locations.
6. If the user makes multiple selections, those choices are stored as a space delimited list. Because this list is space delimited, the choices themselves cannot contain spaces.
7. The contents of the *value* option for the *checkgroup* are never serialized.
8. The *mouseover* option is active for each check box in the group, not for the group as a whole. This means that you can use the <xforms:extension> element to make changes to individual check boxes in the group. For example, the following *checkgroup* changes the background color of each radio button when the mouse is over it:

   ```
   <xforms:itemset nodeset="instance('currency')/choice">
      <xforms:label ref="@show"></xforms:label>
      <xforms:value ref="."></xforms:value>
   ```

```
      <xforms:extension>
         <bgcolor compute="mouseover == 'on' ? 'blue' : 'green'"/>
      </xforms:extension>
   </xforms:select>
```

9. The *itemlocation* of radio buttons in a *radiogroup* is interpreted relative to the top-left of the group, not the top-left of the form. The top-left of the group is set 3 pixels in from the left edge of the group, and 3 pixels down from the top edge of the group. This allows room for a border to be added to the group.
10. If using relative positioning, you can give each *check* an *itemlocation* of after the *itemprevious*, as shown:

    ```
    <xforms:extension>
       <itemlocation>
          <after compute="itemprevious"/>
       </itemlocation>
    </xforms:extension>
    ```

    This will place each *check* after its predecessor, except for the first *check*, which is placed at the top left corner of the group. For the first *check*, the after command is ignored because the *checkgroup* position is not set until after the contained *check* items are positioned, and *itemlocation* keywords are ignored when they refer to items that have not been positioned.
11. The *itemnext* is set as follows:
    - The *checkgroup* itself refers to the first check in the group.
    - The last check in the group refers to the item that follows the *checkgroup* in the build order.
    - The item that precedes the *checkgroup* in the build order refers to the *checkgroup* itself.
12. The size of a *checkgroup* is determined by the size of the radio items and labels in that group. As such, the extent setting in the *itemlocation* option has no affect on the group's sizing.
13. You can use the *constraint* setting in the data model to limit the number of selections the user can make. To do this, set the <xforms:value> of each selection to a standard length. For example, you might have five choices and give them values of 1-5 (each being one character long). When the user makes some selections, the values of the selections are concatenated into a space delimited list. You can predict the length of this list based on the length of your standard values. For example, if the user selects two items, then the list will be three characters long (1 2), while selecting three items will create a list that is 5 characters long (1 3 5). This allows you to set a constraint on the length of that value (which is stored by the data element that is bound to the <xforms:select>) that will actually limit the number of selections the user can make.
14. You can then set a constraint on the data element that is bound to the <xforms:select> so that it cannot exceed a certain length. In this case, limiting it to a length of less than or equal to five will only allow three selections to be made.

pane

A pane is used to contain one of the following:

- A group of items that can be positioned or made visible as a unit, and that can be given a common border or background.
- A switch, which allows you to group items into sets, and then display one set of items at a time to the user.

The pane itself can also have a physical appearance, such as a border or a different background color, that visually groups the items for the user.

Available Options

active, bgcolor, border, first, focused, itemlocation, itemnext, itemprevious, last, label, labelbgcolor, labelfontcolor, labelfontinfo, next, previous, printbgcolor, printlabelbgcolor, printlabelfontcolor, printvisible, suppresslabel, visible, xforms:group, xforms:switch

Example

The following example shows a data model that we will link to a group of items contained within a pane:

```
<xformsmodels>
   <xforms:model>
      <xforms:instance xlmns="" id="customerInfo">
         <customerData>
            <name></name>
            <address>
               <street></street>
               <city></city>
               <country></country>
            </address>
         </customerData>
      </xforms:instance>
   </xforms:model>
</xformsmodels>
```

The following pane contains a group of items in an *xforms:group* that link to the data model:

```
<pane sid="Address">
   <xforms:group ref="address">
      <field sid="street">
         <xforms:input ref="street">
            <xforms:label>Street:</xforms:label>
         </xforms:input>
      </field>
      <field sid="City">
         <xforms:input ref="city">
            <xforms:label>City:</xforms:label>
         </xforms:input>
      </field>
      <field sid="Country">
         <xforms:input ref="country">
            <xforms:label>Country:</xforms:label>
         </xforms:input>
      </field>
   </xforms:group>
</pane>
```

The following example shows a *pane* that contains an *xforms:switch*. This pane creates a "wizard" style form, in which the user completes one section then clicks a button to move to the next section. Each section is enclosed by an *xforms:case* within the *xforms:switch*, and each section except the last contains a button that changes the switch to the next case:

```
<pane sid="wizard">
   <xforms:switch>
      <xforms:case id="applicantName" selected="true">
         <field sid="firstName">
            <xforms:input ref="applicant/firstname">
```

```
               <xforms:label>Enter your first name:</xforms:label>
            </xforms:input>
         </field>
         <field sid="lastName">
            <xforms:input ref="applicant/lastname">
               <xforms:label>Enter your last name:</xforms:label>
            </xforms:input>
         </field>
         <button sid="nextCase1">
            <xforms:trigger>
               <xforms:label>Next</xforms:label>
               <xforms:toggle case="spouseName"
               ev:event="DOMActivate"/>
            </xforms:trigger>
         </button>
      </xforms:case>
      <xforms:case id="spouseName" selected="false">
         <field sid="spouseFirstName">
            <xforms:input ref="spouse/firstname">
               <xforms:label
                  >Enter the first name of your spouse:</xforms:label>
            </xforms:input>
         </field>
         <field sid="spouseLastName">
            <xforms:input ref="spouse/lastname">
               <xforms:label
                  >Enter the last name of your spouse:</xforms:label>
            </xforms:input>
         </field>
         <button sid="nextCase2">
            <xforms:trigger>
               <xforms:label>Next</xforms:label>
               <xforms:toggle case="address"
               ev:event="DOMActivate"/>
         </xforms:trigger>
      </button>
      </xforms:case>
      <xforms:case id="address" selected="false">
         <field sid="street">
            <xforms:input ref="address/street">
               <xforms:label
                  >Enter your street address:</xforms:label>
            </xforms:input>
         </field>
         <field sid="city">
            <xforms:input ref="address/city">
               <xforms:label
                  >Enter the city you live in:</xforms:label>
            </xforms:input>
         </field>
      </xforms:case>
   </xforms:switch>
</pane>
```

Usage Details

1. A *pane* item will always expand to contain the items you place within it, regardless of the specific size you set for the pane.
2. If a pane is contained within an *xforms:repeat*, then the pane and all items within the pane are duplicated for each row of the table.
3. The *next* and *previous* options in the *pane* determine which items precede and follow the *pane* in the tab order, but do not affect the tab order within the *pane*. Instead, the *next* and *previous* options within the items in the *pane* control the tab order within the *pane*.

4. The *first* and *last* options determine where the focus is initially place when the user tabs into a *pane* item.
5. If a pane is not visible, then none of the controls in the pane will be visible, regardless of their state of visibility.
6. If a pane does not have a *visible* option (or if it is empty), then the signed node binding of the *xforms:group* helps determine visibility.
7. The size of a *pane* is determined by the size of the items in the pane. As such, the extent setting in the *itemlocation* option has no affect on the pane's sizing.
8. The *itemprevious* is set as follows:
   - The *pane* refers to the item that precedes the *pane* in the build order.
   - The first item in the first row of the *pane* refers to the *pane* itself.
   - The item that follows the *pane* in the build order refers to the last item in the last row of the *pane*.
9. The *itemnext* is set as follows:
   - The *pane* itself refers to the first item in the *pane*.
   - The last item in the *pane* refers to the item that follows the *pane* in the build order.
   - The item that precedes the *pane* in the build order refers to the *pane* itself.
10. The *label* option defaults to the *xforms:label* of the *xforms:group* within the pane.

radiogroup

Creates a group of radio buttons. This is useful if you want to create a list of options from which the user may select only one choice.

Each radio button appears as an empty circle that is filled with a marker, such as a dot, when selected.

Note that *radiogroup* items are only valid for XForms forms. If you are not using XForms, use the *radio* item instead.

Available Options

acclabel, active, bgcolor, border, focused, format, help, itemlocation, itemnext, itemprevious, label, labelbgcolor, labelborder, labelfontcolor, labelfontinfo, mouseover, next, previous, printbgcolor, printlabelbgcolor, printlabelfontcolor, printvisible, readonly, suppresslabel, value, visible, xforms:select1

Example

The following code creates a *radiogroup* with three choices: US Dollars, CDN Dollars, and Euro. The choices themselves are defined within the *xforms:select1* option.

```
<radiogroup sid="currency">
   <xforms:select1 ref="selectedCurrency" appearance="full">
      <xforms:label
         >Select the currencies you accept:</xforms:label>
      <xforms:item>
         <xforms:label>US Dollars</xforms:label>
         <xforms:value>USD</xforms:value>
      </xforms:item>
      <xforms:item>
         <xforms:label>CDN Dollars</xforms:label>
         <xforms:value>CDN</xforms:value>
      </xforms:item>
      <xforms:item>
         <xforms:label>Euro</xforms:label>
         <xforms:value>Euro</xforms:value>
      </xforms:item>
   </xforms:select1>
</radiogroup>
```

Alternatively, you could create the choices in your data model as follows:

```
<xforms:instance id="currency" xmlns="">
   <data>
      <choice show="US Dollars">USD</choice>
      <choice show="CDN Dollars">CDN</choice>
      <choice show="Euros">Euro</choice>
   </data>
</xforms:instance>
```

In this case, you would use the *xforms:select* option to link to those choices, as illustrated by the following checkgroup:

```
<checkgroup sid="currencyType">
   <xforms:select1 ref="selectedCurrency" appearance="full">
      <xforms:label>Select your preferred currency for payment:</
         xforms:label>
      <xforms:itemset nodeset="instance('currency')/choice">
         <xforms:label ref="@show"></xforms:label>
         <xforms:value ref="."></xforms:value>
      </xforms:itemset>
   </xforms:select1>
</checkgroup>
```

Usage Details

1. To allow the user to select any number of choices use a *checkgroup* instead of a *radiogroup*.
2. Each radio button in a checkgroup can have it's own label. These labels are displayed immediately to the right of the radio button (rather than above the radio button, as with the *radio* item).
3. The single node binding in the *xforms:select* or *xforms:select1* option creates a link between the *value* option for the *radiogroup* and the bound element in the data model, so that they share data. When the user makes a selection, the *xforms:value* of that selection is stored in both locations.
4. The contents of the *value* option for the *radiogroup* are never serialized.
5. The *mouseover* option is active for each radio button in the group, not for the group as a whole. This means that you can use the <xforms:extension> element to make changes to individual radio buttons in the group. For example, the following *radiogroup* changes the background color of each radio button when the mouse is over it:

   ```
   <xforms:itemset nodeset="instance('currency')/choice">
      <xforms:label ref="@show"></xforms:label>
      <xforms:value ref="."></xforms:value>
      <xforms:extension>
         <bgcolor compute="mouseover == 'on' ? 'blue' : 'green'"/>
      </xforms:extension>
   </xforms:itemset>
   ```
6. The *itemlocation* of radio buttons in a *radiogroup* is interpreted relative to the top-left of the group, not the top-left of the form. The top-left of the group is set 3 pixels in from the left edge of the group, and 3 pixels down from the top edge of the group. This allows room for a border to be added to the group.

7. If using relative positioning, you can give each *radio* an *itemlocation* of after the *itemprevious*, as shown:

   ```
   <xforms:extension>
      <itemlocation>
         <after compute="itemprevious"/>
      </itemlocation>
   </xforms:extension>
   ```

   This will place each *radio* after its predecessor, except for the first *radio*, which is placed at the top left corner of the group. For the first *radio*, the after command is ignored because the *radiogroup* position is not set until after the contained *radio* items are positioned, and *itemlocation* keywords are ignored when they refer to items that have not been positioned.
8. The *itemnext* is set as follows:
   - The *radiogroup* itself refers to the first radio in the group.
   - The last check in the group refers to the item that follows the *checkgroup* in the build order.
   - The item that precedes the *radiogroup* in the build order refers to the *radiogroup* itself.
9. The size of a *radiogroup* is determined by the size of the radio items and labels in that group. As such, the extent setting in the *itemlocation* option has no affect on the group's sizing.

slider

Creates a sliding control, similar to a volume control, that lets the user set a value within a specific range. The slider is always horizontal.

Note that this item is only available in an XForms form. There is no equivalent for an XFDL form.

Available Options

acclabel, active, bgcolor, border, focused, fontcolor, fontinfo, format, help, itemlocation, itemnext, itemprevious, label, labelbgcolor, labelborder, labelfontcolor, labelfontinfo, next, previous, printbgcolor, printfontcolor, printlabelbgcolor, printlabelfontcolor, printvisible, readonly, size, suppresslabel, value, visible, xforms:range

Example

The following example shows and slider that allows the user to select any number between 1 and 10:

```
<slider sid="rating">
   <xforms:range ref="rating" start="1" end="10" step="1">
      <xforms:label>Rate this form on a scale of 1 to 10</xforms:label>
   </xforms:range>
</slider>
```

Usage Details

1. The numbers that indicate the value are always shown at the bottom the slider.
2. The *fontcolor* and *fontinfo* option affect the numbers that show the value of the slider.

3. The single node binding in the *xforms:range* option creates a link between the *value* option for the *slider* and the bound element in the data model, so that they share data. When the user makes a selection, that value is stored in both locations.
4. The contents of the *value* option for the *slider* are never serialized.
5. The *supresslabel* option supresses the slider's label, but does not affect the numbers that indicate the value.

table

Creates a traditional table of repeated items organized into rows. This is accomplished by creating a template row that includes all of the items that should appear in each row. This row is then linked to the XForms data model.

Each time a new row is added to the table, the template items are duplicated to create the new row. This occurs when the elements in the data model that are linked to those items are duplicated. This allows the table to expand to any size while ensuring that data for each row in the table is still maintained in the data model.

The template row is created within an *xforms:repeat* option. This option is used to group the items that create the template row, and also links the row to particular portion of the data model. The template items can be configured with any location relative to one another. This means that they need not appear in horizontal succession.

Rows are added or removed from a table using the *xforms:insert* and *xforms:delete* action respectively. For more information about actions, see "Details on XForms Actions" on page 245.

Available Options

active, bgcolor, border, first, focused, itemlocation, itemnext, itemprevious, last, next, previous, printbgcolor, printvisible, visible, xforms:repeat

Example

Before creating a table, you should create the data model that the table will reflect. The data model should group the elements you want in the table by rows. For example, suppose you wanted to create a purchase order form. Your table might collect three pieces of information: what the user wants to purchase, how many units they want to purchase, and what the cost of that item is. You might create the following data model to store that information:

```
<po>
   <order>
      <row>
         <product></product>
         <quantity></quantity>
         <lineTotal></lineTotal>
      </row>
   </order>
</po>
```

As you can see, the three data elements are contained by a <row> element. This will allow us to duplicate any number of rows.

For the actual table, we will use the following items to reflect the data model: a popup that lets the user select which product to purchase, a field that lets them enter a quantity for the product, and a label that displays the cost of the products. Furthermore, these items will be contained within an *xforms:repeat* option, we will link that option to the <row> element in the data model, as shown:

```
<table sid="itemsTable">
   <xforms:repeat nodeset="order/row">
      <popup sid="Product">
         <xforms:select1 appearance="minimal" ref="product">
            <xforms:label>Choose product</xforms:label>
            <xforms:item>
               <xforms:label>Widget</xforms:label>
               <xforms:value>widget</xforms:value>
            </xforms:item>
            <xforms:item>
               <xforms:label>Gadget</xforms:label>
               <xforms:value>gadget</xforms:value>
            </xforms:item>
         </xforms:select1>
      </popup>
      <field sid="Qty">
         <xforms:input ref="quantity">
            <xforms:label></xforms:label>
         </xforms:input>
      </field>
      <label sid="LineTotal">
         <xforms:output ref="lineTotal"></xforms:output>
         <value></value>
      </label>
   </xforms:repeat>
</table>
```

When the user adds rows to the table, the description of the table does not change. Instead, the data model expands to include the new data. For instance, if the user had entered two rows of data, your data model might look like this:

```
<po>
   <order>
      <row>
         <product>widget</product>
         <quantity>2</quantity>
         <lineTotal>2.00</lineTotal>
      </row>
      <row>
         <product>gadget</product>
         <quantity>5</quantity>
         <lineTotal>15.00</lineTotal>
      </row>
   </order>
</po>
```

Based on this data model, the Viewer will create two rows of the popup, field, and label items, with no changes to the markup for the *table* item. This occurs because the nodeset in the *xforms:repeat* binds to every <row> element in the <order>, duplicating the template items for each.

Usage Details

1. To add rows to a table or remove rows from a table, you must use the *xforms:insert* and *xforms:delete* actions respectively. For more information about these actions, refer to “Details on XForms Actions” on page 245.

2. When using an *xforms:insert* action or an *xforms:delete* action, you should add an *xforms:bind* that makes the data elements for the last row non-relevant. For example:

   ```
   <xforms:bind nodeset="order/row[postion()=last()]"
      relevant="false()"/>
   ```

   This allows you to start with an empty table but also helps you to preserve a row of data that is used as a prototype when you insert a new row.

   You should also amend the delete action so that it does not delete this prototypical row. For example:

   ```
   <xforms:delete nodeset="order/row[last()>1]" at="index('table')/>
   ```

   If the prototypical row of data is deleted, the *xforms:repeat* becomes non-functional, as there is no data tempate for the *xforms:insert* to use. This is a limitation of XForms 1.0.
3. The *itemlocation* of items in table rows is interpreted relative to the top-left of the table row, not the top-left of the form. The top-left of each row is determined as follows:
   - Each row is set 3 pixels in from the left edge of the table.
   - The first row is set 3 pixels below the top of the table.
   - Each successive row is set 1 pixel below the previous row.

     By default, only 1 pixel of space is inserted between rows. As such, we recommend that you position the items in your rows to add the desired amount of additional space to the top of each row. For example, you might set each item in a row to have a <y> value of 2, thereby adding 2 pixels of space to the top of the row.
4. The *itemlocation* of items in table rows may reference items in the same row of a table, or items outside of the table. For instance, you might want to align an item in a row with a column heading. However, the *itemlocation* cannot reference containers (such as panes or tables) that contain that item.
5. The *next* and *previous* options in the *table* determine which items precede and follow the *table* in the tab order, but do not affect the tab order within the *table*.
6. When tabbing into a table, a forward tab places you in the first row of the table and reverese tab places you on the last row of the table. The *first* and *last* options determine which item in that row receives the focus.
7. Tab order within a table is determined by the *next* and *previous* options within the items in each row, combined with the natural order of the rows. When you are on the *last* item in a row and tab forward, you move to the *first* item in the next row. When you are on the *first* item in a row and tab backward, you move to the *last* item in the previous row.
8. When using computes with a table, the following restrictions apply:
   - Computes written within a row may not reference elements in a different row.
   - Computes written outside a row may not reference elements within a row.

     Note that this means computes within a row of a table *may* reference elements outside of the table.
9. If the nodeset binding of the *xforms:repeat* is empty or contains non-relevant nodes, then the *xforms:repeat* provides a default of **false** to the table's *visible* and *active* options.
10. If the table is not visible, then none of the controls in the table will be visible, regardless of their states of visibility.

11. When working with tables, users will also need controls (such as buttons) that can add and delete rows. To do this, you must use the *xforms:insert* and *xforms:delete* actions. For more information about these actions, refer to "Details on XForms Actions" on page 245.
12. The size of a *table* is determined by the size of the items in the table. As such, the extent setting in the *itemlocation* option has no affect on the table's sizing.
13. The *itemprevious* is set as follows:
    - The *table* refers to the item that precedes the *table* in the build order.
    - The first item in the first row of the *table* refers to the *table* itself.
    - The item that follows the *table* in the build order refers to the last item in the last row of the *table*.
14. The *itemnext* is set as follows:
    - The *table* itself refers to the first item in the first row of the *table*.
    - The last item in the last row of the *table* refers to the item that follows the *table* in the build order.
    - The item that precedes the *table* in the build order refers to the *table* itself.

Details on Options and Array Elements

In the XFDL language, items contains options, which define the characteristics of an item. Options themselves may contain any number of array elements, which further define the characteristics.

This chapter outlines the common features of options, then details each of the XFDL options separately. For information about XForms options, refer to “Details on XForms Options” on page 205.

Syntax

For simple character data content:

```
<optionTag>character data content</optionTag>
```

For computed options:

```
<optionTag compute="expression">character data content</optionTag>
```

For array options:

```
<optionTag>
   <!-- suboption elements -->
</optionTag>
```

An option defines a characteristic given to a form, a page, or an item. For example, the *bgcolor* option set at the form or page global level defines the background color of the pages of the form itself whereas a *bgcolor* option set at the item level defines the background color for the containing item. Some form and page global options define defaults for item-level options. For example, if an item has no fontinfo option, then the fontinfo in the page globals are used, and if the page globals contain no *fontinfo* option, then the *fontinfo* in the form globals (and the form global *fontinfo* has implied defaults if it is not specified).

The definition of an option consists of content between start and end tags. The element tag defines the type of option. This type must be one of the option types defined in this specification, or a user-defined option that follows the rules in the ″custom option″ description in this specification.

Option Content

The content of an option can take one of three formats: simple character data, a compute, or an array of subordinate XML elements. Computes are identified by a compute attribute, while arrays are identified by the presence of subordinate elements.

Simple Character Data

The default is simple character data, in which case the option must contain text with no child elements. For example:

```
<value>This is the value</value>
```

Compute

If the character content must be computed, then the computational expression appears in the start tag of the option in an attribute named *compute*. If the XFDL computation system has been applied to the form, then the the option also contains simple character data for the current computed value of the expression. For example:

```
<value compute="price1Field.value + price2Field.value * '0.07'">205.68</value>
```

It is typical to have a form run its computes on a client machine, then have server modules simply read the current values, ignoring the content of the compute attributes. In essence, an application can ignore the compute attributes unless it must change element values that are referenced by computed options. See section "The XFDL Compute System" on page 419 for details on how the compute expression is represented.

Array

The third case for an option's content is an array of subordinate elements. The option must contain one or more array elements. For example:

```
<itemlocation>
   <below>nameField</below>
   <after>addressField</after>
</itemlocation>
```

Each array element may also contain an array. This recursive definition permits arbitrary depth for XFDL arrays.

Array Element Names

A number of the XFDL-defined options use array elements. XFDL assigns names to each of these array elements so that they are easier to reference.

Order of Precedence of Options

An option set at a lower level in the form hierarchy overrides a similar option set at a higher level. It overrides it for only the level it is in and any that come below it in the hierarchy. For example, the *fontinfo* option in the following example would override a global *fontinfo* setting for the page it is in, and also for any items in that page.

```
<page sid="Page1">
   <global="global">
      <fontinfo>
         <fontname>Helvetica</fontname>
         <size>12</size>
         <effect>plain</effect>
      </fontinfo>
   </global>
```

Defining Form Global and Page Global Options

Form global options are optional and must be defined in a <global> element in a <globalpage> element after the XFDL start tag and before the first <page> in a form. Page global options are optional and must be defined in a <global> element after the <page> start tag and before the first item in a page. To determine whether an option is a valid form global or page global option, see section "Form Globals" on page 31.

Data Type Designators Used in Option Descriptions

XFDL defines a set of data types that describe type of content allowed in an option. Each option description in this specification uses one or more of the following data type designators:

char A single ASCII character.

string A series of ASCII characters.

color A color name, an RGB triplet, or a hexadecimal RGB value that represents the color.

The syntax of an RGB triplet is:

```
<bgcolor>Red, Green, Blue</bgcolor>
```

Where red, green, and blue are values from 0 to 255.

The syntax for a hexidecimal RGB value is:

```
<bgcolor>#RRGGBB</bgcolor>
```

Where RR, GG, and BB are the hexidecimal values for the red, green, and blue settings.

coordinate
Whole number in the range 0 to 1,000 representing one coordinate of a position.

integer
Positive or negative whole number in the range -32,768 to 32,767.

unsigned byte
Whole number in the range 0 to 255.

unsigned
Whole number in the range 0 to 65,535.

numeric boolean
A value of 0 or 1, in which 0 is false and 1 is true.

acclabel

Defines a message that is available to active screen readers. When the focus shifts to the item containing the acclabel, the message is read aloud by the screen reader. The message should contain additional information about the item to assist users with vision impairments.

Syntax

```
<acclabel>message</acclabel>
```

message	*string*	a message that is read aloud to users

Available In

button, check, checkgroup, combobox, field, list, popup, radio, radiogroup, slider

Example

The following example shows a field that contains an accessibility message.

```
<field sid="firstName">
   <acclabel>Type your first name.</acclabel>
</field>
```

When the focus moves to the field, the acclabel message will be passed to the screen reader, which will read ″Type your first name″ aloud.

Usage Details

1. Default: none
2. Screen readers normally provide information in addition to the *acclabel* option. This information is defined by the screen reader in use, and cannot be controlled through XFDL.
3. If the item contains a *label* option, the screen reader will read the *label* option as well as the *acclabel* option.
4. If the item has a *value* option, the screen reader will read the item's value. If the item has a blank value, the screen reader will say that the item is empty.
5. If the item has associated cells, the screen reader reads the *value* option of the cells.
6. If the item contains a *help* option, the screen reader will read the *value* of the associated help item. The screen reader will also read this help message if the item contains a *format* option and the item's content is invalid when the user tries to tab to a new item
7. If the item contains a *format* option with a message flag, the screen reader will read the contents of the message flag if the item's content is invalid when the user tries to tab to a new item.
8. If the item is signed, the screen reader reads the signed message as well as the *acclabel* option. The signed message is defined by the Viewer and is not modifiable through XFDL.
9. If you are using a *label* item to provide information for another item, the *acclabel* option of the second item should include the contents of the *label* item. Since *label* items never receive the focus, screen readers will never read their contents. The *acclabel* option can provide the same information to users, with further explanation if necessary.
10. The following table provides the actions which invoke the screen reader and the order in which it reads the messages provided by the form and the Viewer:

Action	**Viewer Help On**	**Item is signed**	**Messages Read by Screen Reader**
Item gains focus	No	No	label option + acclabel + value
Item gains focus	Yes	No	label option + acclabel + value + help
Item gains focus	n/a	Yes	label option + acclabel + value + help + signed
Tab out of item failed	n/a	n/a	format + help
Spacebar or arrow keys activate list of choices	n/a	No	cell value

activated

Specifies whether an item, page, or form is currently activated by the user or not. This option is usually set by an external program such as a parser, but under certain circumstances can be set by computes in the form. The *activated* option for an item can be set by a compute if the item is an action, button or cell utilized on the current page. The *activated* option must be set to either on or off and the item must be capable of being activated. Cells that are not grouped with a *popup*, *combobox* or *list* on the current page cannot be activated. Buttons that are not visible cannot be activated.

This option is not saved or transmitted as part of the form. Instead, it is automatically created each time the form is read into memory, and is maintained only during display or processing.

Syntax

```
<activated>status</activated>
```

status	*on*	item, page, or form is currently activated by user
	off	item, page, or form is not currently activated by user
	maybe	button only: item might be activated, as user has pressed it, but has not yet released it

Available In

action, button, cell, combobox, popup, page global, form global

Example

The following example shows a button that changes color when it is activated:

```
<button sid="saveButton">
   <type>saveas</type>
   <value>Save</value>
   <activated>off</activated>
   <bgcolor compute="toggle(activated, 'off', 'on') == '1'
      ? 'white' : 'gray'"></bgcolor>
</button>
```

The button will appear white when the user activates it, and gray otherwise.

The following example shows how the activated option can be set for an item based on user input. This use of the activated option works with the keypress option to establish a default button on the form.

```
<button sid="DefaultButton">
   <type>cancel</type>
   <activated>off</activated>
   <custom:myoption compute="toggle(global.keypress, '', &#xA;
      'ESC') == '1' ? set('activated','on') : &#xA;
      ''"></custom:myoption>
</button>
```

When the user presses the ESC key (which is not processed by any item on the form), the activated option for the button will be set to on and the button will fire.

Usage Details

1. Default: off
2. *Activated* is set to on when an item is activated, and remains on until any transaction initiated by the item is properly under way. For example, in a print button, activated will be turned on when the user initiates the print action, and will remain on until network results indicate the print action is taking place.
3. The *activated* option is not included in form descriptions that are saved or transmitted.
4. Specific details on activated behavior for each item:
   - **action —** actions set *activated* to on when they fire, and off when the transaction they initiate is under way.
   - **button —** buttons set *activated* to maybe when the user holds the mouse pointer or space bar down on the button. They set it to on if the user releases the pointer or space bar while over the button, and they set activated to off when the transaction the button initiates is under way.
   - **cell —** cells behave in the same manner as buttons. In the split second during which a user selects a select type of cell, it sets *activated* to on. It turns *activated* off as soon as the action of being selected is finished. Cells that initiate network transactions set *activated* to on from the beginning of the request to the time when the request produces results. Note that there is no maybe status for a cell.
   - **combobox and popup —** comboboxes and popup lists set *activated* to on when their lists are popped open, and off when the lists are not open. Note that the "field" portion of a combobox does not register an *activated* setting.
   - **page —** a page sets *activated* to on while it is open, and off when it is not. A page can be open and activated even when the form is minimized (not actively on screen).
   - **form —** a form sets *activated* to on while it is open, and off when it is not. A form can be open and activated even when it is minimized (not actively on screen) .

active

Specifies whether an item is active or inactive. Inactive items do not respond to user input and, if possible, appear dimmed. For example, an inactive check box will be dimmed and the user will not be able to select or deselect the box.

Syntax

```
<active>status</active>
```

status	*on*	item is active
	off	item is inactive

Available In

action, button, cell, check, checkgroup, field, help, label, list, popup, radio, radiogroup, slider

Example

This sample specifies the item is active.

```
<active>on</active>
```

Usage Details

1. Default:
   - XFDL: **on**.
   - XForms: defaults to the *relevant* property for the data element to which the containing item is bound.
2. Setting active to **off** is similar to setting the *readonly* option to **on**.

bgcolor

Defines the background color of a page or an item.

Syntax

```
<bgcolor>color</bgcolor>
```

color	*special*	The color may be expressed in any of the following formats: • Comma-separated RGB values. For example: `192, 192, 192` • Hexadecimal-based RGB values. For example: `#336699` • Color name. For example: `blue`

Available In

box, button, check, checkgroup, combobox, field, label, list, popup, radio, radiogroup, slider, toolbar, page global, form global

Examples

These samples all set the background color to forest green.

```
<bgcolor>forest green</bgcolor>
<bgcolor>34,139,34</bgcolor>
<bgcolor>#228B22</bgcolor>
```

Usage Details

1. Default: varies depending on the object
2. The transparent color has no RGB equivalent.
   - Form: **white**
   - Page: the form bgcolor setting or default (white)
   - Item (depends on the item type):

- button items: **gray** (or grey)
- check, combobox, field, list, popup, radio, and slider items: **white**
- checkgroup and radiogroup items: **transparent**, but the individual check and radio items that form the group default to white.
- label, table, and pane items: **transparent** (if no color is specified, the label background color will be the same as the page background color).
- All other items: the background color of the page.

border

Defines whether an item is displayed with a border. Borders are drawn as a three dimensional effect.

Syntax

```
<border>status</border>
```

status	*on*	item has a border
	off	item does not have a border

Available In

box, button, check, checkgroup, combobox, field, label, list, pane, popup, radio, radiogroup, slider, table

Example

This sample sets the item to display a border:

```
<border>on</border>
```

Usage Details

1. Default:
   - For *label* items, the default is always **off**.
   - For all other items, the default is **on**.

colorinfo

Records the colors used to draw the form when a user signs it. This option is only created if the user is allowing the operating system colors to override the color settings in the form. This is most common for users with vision disabilities who may set the operating system colors to provide better contrast between elements on the screen. When the operating system colors override those set by the form itself, it is useful to create a record of those colors so that the appearance of the document, when signed, can be recreated.

Syntax

```
<colorinfo>
   <color_name_1>color</color_name_1>
   ...
   <color_name_n>color</color_name_n>
</colorinfo>
```

color_name	*text*	the name of the operating system color. Possible color names include: • **window** — The color of the window displaying the form. • **windowtext** — The color of the text used in the form. • **borderlight** — The color of all three dimensional borders drawn on the form. • **buttonshadow** — The color used to draw the shadow on a button. • **buttonface** — The color used for to draw the face of a button. • **buttontext** — The color used to draw the text on a button.

color	*special*	The color may be expressed in any of the following formats:
		• Comma-separated RGB values. For example:
		`192,192,192`
		• Hexadecimal-based RGB values. For example:
		`#336699`
		• Color name. For example:
		`blue`

Available In

signature

Example

When a user signs a form that is respecting the operating system color, a colorinfo block similar to the following is added to the signature item:

```
<colorinfo>
   <window>255,255,255</window>
   <windowtext>0,0,0</windowtext>
   <borderlight>255,255,255</borderlight>
   <bordershadow>157,157,157</bordershadow>
   <buttonface>224,224,224</buttonface>
   <buttontext>0,0,0</buttontext>
</colorinfo>
```

Usage Details

1. Default: **none**

coordinates

Records the position of the mouse pointer on an image. The image must exist in a button item. The recording occurs when a user selects (i.e. clicks) the button using the mouse pointer.

The position is an intersection on an unseen grid overlaying the image. The points along each axis of the grid range from zero (0) through 1000 with position 0,0 occurring in the button's top left corner. The coordinates map the intersection closest to the mouse pointer's position.

Syntax

```
<coordinates>
   <x>X_coordinate</x>
   <y>Y_coordinate</y>
</coordinates>
```

X_coordinate	*coordinate*	the coordinate on the X axis
Y_coordinate	*coordinate*	the coordinate on the Y axis

Available In

button

Example

When a user clicks on a button containing an image, a *coordinates* option is inserted into the *button* item. The following coordindates option sets a position of 180 on the x-axis and 255 on the y-axis.

```
<coordinates>
   <x>180</x>
   <y>255</y>
</coordinates>
```

Usage Details

1. Default: **none**

data

Associates an *action*, *button*, or *cell* item with a single *data* item. The *data* option is valid only in items with a type setting of enclose, display, extract, or remove.

Syntax

```
<data>data item</data>
```

data item	*string*	the item sid of the data item to associate with the action, button, or cell

Available In

action, button, cell

Example

The button below is an enclosure button associated with a single data item.

```
<button sid="encloseImageButton">
   <value>Update Image</value>
   <type>enclose</type>
   <data>displayImage</data>
</button>
```

If a user enclosed another file, then the *data* item referred to in the button's *data* option would be replaced with the new *data* item. (The *data* item would use the same item sid - the one that's referred to in the *data* option.)

Usage Details

1. Default: **none**
2. A *data* option specifies only zero or one data items.
3. If an item with a type setting of enclose and a *data* option is used to enclose a second *data* item, then the second *data* item will replace the first.
4. If an enclosure mechanism is used to replace an image stored in a *data* item with a new image (see above), then buttons and labels whose *image* option is set to the identifier of the image *data* item will be updated to display the new image.
5. A *data* item referred to in a *data* option might also have a *datagroup* option and thus belong to the datagroups of other actions, buttons, or cells.

datagroup

Provides a way of associating related *data* items to each other and to certain other items. There are two ways of using this option. In the first case, it enables you to create a group of *data* items, called a datagroup. In the second case, this option enables you to reference such a datagroup from *button*, *action*, or *cell* items.

This option is most often used to group file enclosures. For example, you can use this feature to create folders with which users can organize their enclosures. Each enclosed file can belong to several datagroups, and each datagroup can contain several enclosed files.

Syntax

```
<datagroup>
   <datagroupref>datagroup reference_1</datagroupref>
   ...
   <datagroupref>datagroup reference_n</datagroupref>
</datagroup>
```

Note:

- Include a *datagroup reference* entry for each datagroup this item accesses.

datagroup reference	*string*	identifies a data group. This can done in one of two formats: • *datagroup_name* for datagroups on the same page • *page_sid.datagroup_name* for datagroups on a different page.

Available In

action, button, cell, data

Example

If this sample were part of a data item definition, it would mean the data item belonged to the datagroups Business_Letters, Personal_Letters, and Form_Letters.

If this sample were part of an *action*, *button*, or *cell* item, it would mean the user could store the enclosure in one of the three datagroups.

```
<datagroup>
   <datagroupref>Business_Letters</datagroupref>
   <datagroupref>Personal_Letters</datagroupref>
   <datagroupref>Form_Letters</datagroupref>
</datagroup>
```

In the following example, the Enclose button references the datagroups Business_Letters and Personal_Letters. As a result, when users click this button, they can choose to place the file they are enclosing into one of these folders. Because the datagroup Form_Letters is not specified in the *datagroup reference*, it is not available to "BUTTON1".

```
<button sid="BUTTON1">
   <value>Click to Enclose File</value>
   <type>enclose</type>
   <datagroup>
      <datagroupref>Business_Letters</datagroupref>
      <datagroupref>Personal_Letters</datagroupref>
   </datagroup>
</button>
```

Usage Details

1. Default: none
2. The grouping of *data* items into datagroups cannot span multiple pages. That is, all the *data* items assigned to a given datagroup must belong to the same page.

On the other hand, buttons, actions, and cells from one page can reference datagroups from another page, provided that you specify the page sid in the datagroup reference.
3. Used with items handling enclosures, datagroup lists the datagroups the item can access. Used with a *data* item, datagroup lists the datagroups to which the enclosure belongs. Enclosures are stored in *data* items.
4. Items that handle enclosed files perform enclose, extract, remove, and display actions. These actions types are set using the *type* option.
5. When a user selects an item that handles enclosed files, the list of datagroups appears. The user chooses the datagroup (or folder) with which to work. If the action is enclosing, the enclosed file is added to that datagroup. Otherwise, a list of files in the datagroup appears. The user chooses a file from the list.
6. The action of enclosing a file creates the *data* item, and stores the user's choice of *datagroup* (or folder) in the data item's *datagroup* option.

delay

Delays the execution of an automatic action or specifies an automatic action repeat factor. Repeated actions stop when the page containing the action definition closes. Define automatic actions using an *action* item.

Syntax

```
<delay>
   <type>repeat factor</type>
   <interval>interval</interval>
</delay>
```

repeat factor	*repeat*	queue the action to repeat at the <*interval*> specified
	once	perform the action once after the <*interval*> specified
interval	*integer*	the frequency of repeated actions or the delay before performing single occurrence actions. The unit of measurement is seconds.
	-1	perform the action before the page displays. Only valid with a repeat factor of once.

Available In

action

Example

This sample sets the action to occur once, 15 minutes (900 seconds) after the page opens.

```
<delay>
   <type>once</type>
   <interval>900</interval>
</delay>
```

Usage Details

1. Defaults:
   - repeat factor: **once**
   - interval: **zero seconds**
2. This means the action will occur when the page appears.
3. Repeating automatic actions is one method of creating a sparse-stated connection. It allows the form to indicate periodically to a server application that it is still running.
4. All actions with the same interval occur in the order they are defined in the page.
5. The page does not display while actions with an interval of -1 are running.

dirtyflag

Records whether the form has been updated since the last save or submission. If the user attempts to close the form when the *dirtyflag* is set to on, the user will first be prompted to save their changes.

The *dirtyflag* is set to on whenever the user makes a change to the form. Such changes include typing information into the form, selecting choices in lists or with radio buttons, and so on. The *dirtyflag* is set to off whenever the user saves or submits the form.

Note that the *dirtyflag* is not set by computed changes to the form. For example, if the user clicks a button that triggers a compute, and that compute copies information to a field in the form, the *dirtyflag* would not be set. In these cases, the form should include additional computes that set the *dirtyflag*.

If necessary, the save prompt can be disabled by using a compute to set the *dirtyflag* to off.

This option is not saved or transmitted as part of the form. Instead, it is automatically created each time the form is read into memory, and is maintained only during display or processing.

Syntax

```
<dirtyflag>status<dirtyflag>
```

status	*on*	indicates that the user has changed the form since the last save or network submission
	off	indicates that the user has not changed the form since the last save or network submission

Available In

form global

Example

This example shows how the dirtyflag can be set to off.

```
<button sid="ApplicationSpecificSave">
   <value>Save</value>
   <type>cancel</type>
   <custom:myoption compute="toggle(activated, 'off', 'on') &#xA;
      == '1' ? &#xA;
      MySaveFunction() + set('global.global.dirtyflag', &#xA;
      'off') : ""></custom:myoption>
</button>
```

Usage Details

1. Defaults: **none**
2. The *dirtyflag* option is not saved or transmitted with the form description.

excludedmetadata

This option allows additional data about a signature to be included, but never signed. This makes it possible to store the notarization of signatures without interfering with other, overlapping signatures.

For example, if signer1 signs a form and then signer2 affixes an overlapping signature, you could not modify the first signature without breaking the second. In this case, you would not be able to notarize the first signature, since affixing the notarization would change the mimedata of that signature and break the second signature.

The *excludedmetadata* provides a place to store the notarization for the first signature without breaking the second signature. You can add information to this option at any time, since the *excludedmetadata* option is never signed.

Syntax

```
<excludedmetadata>
   <servernotarizations>
      <notarization>notarization1</notarization>
      ...
      <notarization>notarizationn</notarization>
   </servernotarizations>
</excludedmetadata>
```

Notarization	*string*	a compressed base64 encoded PKCS-7 signature that signs the hash of the mimedata option and the details of the signature that is being notarized

Available In

signature

Example

The following example shows an *excludedmetadata* option with two notarizing signatures. Note that the base64 blocks would be much larger in practice.

```
<excludedmetadata>
   <servernotarizations encoding="base64-gzip">
      <notarization>asdfkj439fgasdf81hgb</notarization>
      <notarization>opkbt1ed7f8y3476p294</notarization>
   </servernotarizations>
</excludedmetadata>
```

Usage Details

1. Default: **none**

filename

Identifies the name of an enclosed file. This name appears in the list of enclosed files.

Syntax

```
<filename>name of file</filename>
```

name of file	*string*	the name of the enclosed file

Available In

data

Example

This sample specifies the name of an enclosed file:

```
<filename>std_logo.xfd</filename>
```

Usage Details

1. Default: **none**
2. To ensure cross-platform compatibility, limit filenames to the following set of characters: lowercase letters from a to z, uppercase letters from A to Z, the integers 0 through 9, and the underscore (_).
3. To ensure cross-platform compatibility, limit form names to a maximum of eight characters, followed by a *.xfd* extension.

first

Identifies the first item in a repeat, group or switch. This is the item that first receives the focus when the user tabs into a group, a particular case in a switch, or a new row in a repeat.

This option affects the tab order in the following ways:

- When the user tabs forward into a table or pane, the focus goes to this item first. In the case of a table, the focus goes to this item in the first row.
- When the user tabs forward from the end of a row, the focus goes to this item in the next row (if there is one), or to the item stipulated by the table's *next* option.
- When the user tabs backward from this item, the focus goes to the preceding row, or to the item that precedes the table or pane.

Syntax

```
<first>item reference</first>
```

item reference	*string*	an XFDL reference to the first item in a row of a table or the first item in a pane.

Available In

pane, table

Example

The following example shows a table in which the *first* option contains a reference to the second item in the repeat structure:

```
<table sid="itemsTable">
   <first>Product</first>
   <last>Qty</last>
   <xforms:repeat nodeset="order/row">
      <field sid="Qty">
         <previous>Product</previous>
         <xforms:input ref="qty">
            <xforms:label></xforms:label>
         </xforms:input>
      </field>
      <popup sid="Product">
         <next>Qty</next>
         <xforms:select1 appearance="minimal" ref="product">
            <xforms:label>Choose product</xforms:label>
            <xforms:item>
               <xforms:label>Widget</xforms:label>
               <xforms:value>widget</xforms:value>
            </xforms:item>
            <xforms:item>
               <xforms:label>Gadget</xforms:label>
               <xforms:value>gadget</xforms:value>
            </xforms:item>
         </xforms:select1>
      </popup>
      <label sid="LineTotal">
         <xforms:output ref="lineTotal"/>
      </label>
   </xforms:repeat>
</table>
```

In this case, when the user first tabs into the table, the focus goes to the popup in the first row. When the user tabs to the next row, the focus goes to the popup in the second row.

Usage Details

1. Default: the first item in the <xforms:repeat> or <xforms:group> element, or the first item in the selected case of the <xforms:switch> element.
2. When a pane contians an xforms:switch, this option is not effective unless all cases contain an element with the same sid that is identified as *first*.

focused

Specifies whether an item, page, or form currently has the input focus. This option is usually set by code outside XFDL, but can also be set by a compute, provided that the compute is setting the focus of an item to on, the item is on the same page, and the item receiving the focus is capable of doing so.

This option is not saved or transmitted as part of the form. Instead, it is automatically created each time the form is read into memory, and is maintained only during display or processing.

Syntax

```
<focused>status</focused>
```

status	*on*	item, page, or form has input focus
	off	item, page, or form does not have input focus

Available In

button, check, checkgroup, combobox, field, list, popup, radio, radiogroup, slider, page global, form global

Example

The following example shows a button that changes its color to white if it has the input focus, and to blue if it does not.

```
<button sid="saveButton">
   <type>saveas</type>
   <value>Save</value>
   <focused>off</focused>
   <bgcolor compute="focused=='on' ? 'white' : 'blue'"><bgcolor>
</button>
```

The following example shows how the focus can be moved to a different item based on user input.

```
<check sid="CHECK1">
   <value>off<value>
   <label>Check here to skip next section</label>
   <custom:myoption compute="toggle(value, 'off', 'on') == &#xA;
      '1' ? set('FIELD14.focused','on'): ''"></custom:myoption>
</check>
```

Usage Details

1. Default: **off**
2. The *focused* option is set to on when an item, page, or form receives the input focus, and is set to off when the focus is moved to another item, page or form.
3. An object's *focused* option does not change when the form application displaying it becomes active or inactive on a desktop. For example, a page that is open on screen will have a *focused* option set to on, even if the page is minimized or is not the currently active application on the desktop.

4. In objects that are hierarchical, it is possible for more than one object to have the focus at one time. For example, a form, a page, and a field can all be focused at the same time.
5. When a form viewing application is closing a form, it should set all *focused* options to off and then resolve all formulas before shutting down.
6. The *focused* option is not included in form descriptions that are saved or transmitted.

focuseditem

Specifies which item in the page currently has the focus.

This option is not saved or transmitted as part of the form. Instead, it is automatically created each time the form is read into memory, and is maintained only during display or processing.

Syntax

```
<focuseditem>sid</focuseditem>
```

sid	*string*	the sid of the item that currently has the focus

Available In

page global

Example

The following example shows a label that uses a compute to set its value. The compute copies the name of the item that currently has the focus from the *focuseditem* option. The label then displays this name.

```
<label sid="itemWithFocus">
   <value compute="PAGE1.global.focuseditem></value>
</label>
```

Usage Details

1. When the page changes, the focuseditem for the previous page is set to empty. For example, if the user changed from ″PAGE1″ to ″PAGE2″, the *focuseditem* option for ″PAGE1″ would be set to an empty string.
2. The *focuseditem* option is maintained by the form viewing application, and is not included in form descriptions that are saved or transmitted.

fontcolor

Defines the font color for the text or filler portion of an item. In *radio* and *check* items, fontcolor defines the color of the bullet and check, respectively. In *line* items, fontcolor defines the color of the line. In other items, it defines the text color.

Syntax

```
<fontcolor>color</fontcolor>
```

color	*special*	The color may be expressed in any of the following formats:
		• Comma-separated RGB values. For example:
		`192,192,192`
		• Hexadecimal-based RGB values. For example:
		`#336699`
		• Color name. For example:
		`blue`

Available In

button, check, combobox, field, label, line, list, popup, radio, slider, page global, form global

Examples

These samples all set the font color to chocolate.

```
<fontcolor>chocolate</fontcolor>
<fontcolor>210,105,30</fontcolor>
<fontcolor>#993300</fontcolor>
```

Usage Details

1. Defaults behave as follows:
   - If there is no fontcolor setting at the item level, the item will derive its *fontcolor* setting from the page or form global level. If no fontcolor setting is specified at the page or form level, the default is **black**.
   - For *check* and *radio* items, the default is not inherited from the page or global level, and is always **red**.

fontinfo

Defines the character set, font name, point size, and font characteristics for the text portion of an item. Note that the font selected for an item influences the item's size.

Syntax

```
<fontinfo>
   <fontname>font name</fontname>
   <size>point size</size>
   <effect>effect1</effect>
   ...
   <effect>effectn</effect>
</fontinfo>
```

Note:

- fontname and size must appear first, and in the order shown.
- weight, effects, form, and character set are optional.

font name	*string*	the name of the font.
point size	*unsigned byte*	the size of the font.
effect	*string*	Can be any of the following: • **plain** — use plain face font. • **bold** — use bold face font. • **underline** — use underlined font. • **italic** — use italic font.

Available In

box, button, check, combobox, field, label, line, list, popup, radio, slider, spacer, page global, form global

Example

This sample sets the font information to Times 14, bold italic:

```
<fontinfo>
   <fontname>Times</fontname>
   <size>14</size>
   <effect>bold</effect>
   <effect>italic</effect>
</fontinfo>
```

Usage Details

1. Defaults first to the page setting for *fontinfo*, then to the global setting for *foninfo*. If neither setting exists, then the defaults are: **Helvetica**, **8**, **plain**.
2. If any of the fontinfo settings are invalid, then the defaults will be used.
3. The *size* option calculates item size using the font's average character width. Therefore, choice of font affects item width.
4. XFDL supports the following fonts and font sizes:
   - Fonts: Courier, Times, Symbol (symbol), Helvetica, and Palatino.
   - Sizes: 8, 9, 10, 11, 12, 14, 16, 18, 24, 36, 48.
5. Other fonts and font sizes may be used. However, especially for cross-platform Internet applications, it is best to choose from the ones cited above since they are guaranteed to work.

format

Allows you to apply formatting to the contents of an item, or to create edit checks for that item. It also allows you to set *button* items to be mandatory.

Syntax

```
<format>
   <datatype>data type</datatype>
   <presentation>presentation settings</presentation>
   <constraints>constraint settings</constraints>
</format>
```

Note:
- *Datatype* is mandatory and must appear first; the other settings are optional.

data type	*(see below)*	the type of data the item can contain.
presentation settings	*(see below)*	the formatting to apply to the data in this item.
constraint settings	*(see below)*	the constraints to apply to user input.

Available In

button, checkgroup, combobox, field, label, list, popup, radiogroup, slider

Data Types

You can only declare one data type for each item. If you do not set the data type, XFDL will default to string. XFDL supports the following data types:

Data Type	Description	Format Defaults To:
`currency`	a fixed point decimal number with a scale of 2 and a range equal to the range of a float	any number. Automatically adds .00 to end, if no decimal value specified.
`date`	a date including day-of-month, month, and year	3 Mar 2005
`day_of_month`	the number of a day of the month	12
`day_of_week`	the name or number of a day of the week	Thu
`date_time`	the date, including year, month, day, and the time, including at least hours and minutes.	5 Oct 2005 6:45:21 PM
`float`	a positive or negative floating point decimal number in the range of 1.7 * 10^{-308} to 1.7 * 10^{308}	any decimal number

Data Type	Description	Format Defaults To:
`integer`	a positive or negative whole number in the range of -2,147,483,648 to +2,147,483,647	any whole number
`month`	the name or number of a month	Mar
`string`	free form character data up to 32K long	any group of characters
`time`	a time value containing hours and minutes. Represents either the 12 hour or the 24 hour clock.	11:23:21 PM
`void`	disable entire format option (including data type, presentation, and constraints)	no effect on contents of the item
`year`	a numeric year designation	2005

Presentation Settings

Presentation settings control the output of the text. For example, you can specify that the text should include a currency symbol, or that it should round all numbers up. You can specify any number of presentation settings.

Presentation settings follow this syntax:

```
<format>
   <presentation>
      <settingname1>setting</settingname1>
      ...
      <settingnamen>setting</settingnamen>
   </presentation>
</format>
```

Note:
- You can define any number of settings.

The following list defines the valid settings:

calendar

This sets which calendar is used for formatting dates. It supports the following settings:

- **gregorian** — A solar calendar. The primary calendar for North America.
- **japanese** — A lunisolar calendar (based on lunar and solar cycles). Used in Japan, along with gregorian calendar.

Using this setting forces the calendar to be type selected, regardless of the locale of the form. This setting is only valid for date types, such as date, time, and so on.

Default for en_US locale: **gregorian**.

Default for other locales: not documented.

casetype

Forces the value to be set to a particular case. Valid settings are:

- **upper —** Sets all letters to upper case.
- **lower —** Sets all letters to lower case.
- **title —** Capitalizes the first letter of each word.
- **none —** Leaves the capitalization unchanged.

This setting is only valid for string and date (date, time, and so on) data types.

Default: **none**.

currencylocale

Allows you to set a different locale for a particular currency field. For example, you could set one field to use US dollars and another field to use British pounds.

Valid settings include all valid locales. For a complete list of locales and their corresponding codes, see the "*IBM Workplace Forms™ Locale Specification for XFDL*".

Note that this setting does not convert currencies in any way.

Default: the locale of the form.

decimalseparator

The symbol(s) used to separate the decimal place. This is often a period, as shown:

```
100.00
```

You can use any string. This setting is only valid for int, float, and currency data types.

Default for en_US locale: a period.

Default for other locales: refer to *IBM Workplace Forms Locale Specification for XFDL*.

Note: Must be used in conjunction with the *decimalseparators* constraint.

fractiondigits

Sets the number of digits shown after the decimal place. For example, a setting of 3 would allow three digits after the decimal place, as shown:

```
13.764
```

All values are rounded according to the *round* setting. If no *round* setting is specified, all values are rounded up. (See the *round* setting for an explanation of rounding up.)

Fractiondigits is only valid for float and currency data types. Note that setting both *fractiondigits* and *significantdigits* may cause conflicting formats. In this case, *significantdigits* takes precedence.

Default: the maximum number of digits allowed for the data type.

groupingseparator

The symbol(s) used to separate groups of numbers (for example, thousands in North America). This is often a comma, as shown:

```
1,000,000
```

You can use any string, with the keyword **none** representing no separators at all. However, you should not use strings that already have a meaning, such as the period.

This setting is only valid for int, float, and currency data types.

Default for en_US locale: a comma.

Default for other locales: refer to the *IBM Workplace Forms Locale Specification for XFDL*.

Note: Must be used in conjunction with the *groupingseparators* constraint.

keepformatindata
Sets whether formatting, such as dollar signs and other ″decoration″, is maintained when the value is copied to the XForms model. Valid values are:

- **on —** maintain all formatting.
- **off —** strip all formatting.

Default: **off**.

negativeindicator
Sets the symbols that are used to indicate a negative value. You can place symbols both before and after the number by setting a prefix and a suffix. To do this, you must include the <prefix> and <suffix> tags in your definition, as shown:

```
<negativeindicator>
   <prefix>prefix</prefix>
   <suffix>suffix</suffix>
</negativeindicator>
```

The prefix and suffix are defined as strings. For exampe, if you set the prefix to an open bracket and the suffix to a close bracket, you will get a bracket negative. The following shows the bracket notation for negative 100:

```
(100)
```

You can also leave either the prefix or suffix blank, so long as the other setting has a value.

Note:

- The *pattern* setting overrides the *negativeindicator* setting.
- Do not use this setting with currency data types.
- Do not use symbols that already have meanings, such as the period.

Default for en_US locale: minus sign (-).

Default for other locales: refer to the *IBM Workplace Forms Locale Specification for XFDL*.

pad Sets the number of digits to show, regardless of the value. For example, setting a pad of 5 would result in all numbers having five digits, as shown:

```
00002
00100
```

If the value has more characters than dictated by the *pad* setting, the value is not changed and is displayed as entered.

Pad is only valid for integer, float, and currency data types. Use the *padcharacter* setting to control which character is used to pad the value.

Default : **0** (no padding imposed).

padcharacter

Sets the character to use for padding. For example, if you set the padcharacter to a zero and the *pad* setting was 5, numbers would be displayed as follows:

```
00010
01245
```

You may only specify a single character as the pad character. Furthermore, you must use a pad character that is valid for your data type. For example, you cannot use a Z in an integer value.

Padcharacter is only valid for integer, float, and currency data types. Use the *pad* setting to control how many pad characters are used.

Default for en_US locale: **0**.

Default for other locales: refer to the *IBM Workplace Forms Locale Specification for XFDL*.

pattern

Allows you to set a pattern for number and date data types. This pattern is used to display the data. For example, you might want all numbers to be formatted with two digits after the decimal place.

To learn how to represent number and date patterns, refer to “Defining Patterns” on page 101.

Note that the *pattern* setting overrides both the *style* and *negativeindicator* settings.

patternrefs

Allows you to define one or more patterns for string data types that are used to display the data. For example, you may want to ensure that all phone numbers are displayed with dashes, as shown: 250-604-8734.

You must define each pattern in its own <patternref> tag, as shown:

```
<patternrefs>
   <patternref1>pattern</patternref1>
   ...
   <patternrefn>pattern</patternrefn>
</patternrefs>
```

If you define only one pattern, that pattern is used for all input regardless of the number of constraints you define.

If you define more than one *patternref*, you must define an equal number of patterns in the constraints. Each pattern is then matched to the corresponding constraint. For example, the first pattern is matched to the first constraint, the second pattern is matched to the second constraint, and so on. This allows you to define a different pattern for each constraint.

To learn how to represent string patterns, refer to page 103.

To review best practices for creating phone number, postal code, or e-mail address patterns, see the*Workplace Forms Best Practices for Form Design* document.

Note that the *patternrefs* setting overrides the *style* setting.

Default: as dictated by the *style* setting.

round Determines how values are rounded. Valid settings are:

- **floor** — Always rounds down. For example, 46.9 becomes 46.
- **ceiling** — Always rounds up. For example, 46.1 becomes 47.
- **up** — Rounds values greater than 5 up, and values less than 5 down. For values equal to 5, it rounds up. For example, 46.5 becomes 47, while 46.4 becomes 46.
- **down** — Rounds values greater than 5 up, and values less than 5 down. For values equal to 5, it rounds down. For example, 46.5 becomes 46, while 46.6 becomes 47.
- **half_even** — Rounds values greater than 5 up, and values less than 5 down. For values equal to 5, rounds up if the preceding digit is even, and down if the preceding digit is odd. For example, 46.5 becomes 47, while 45.5 becomes 45.

Round is only valid for integer, float, and currency data types.

Note that if the *significantdigits* setting is used, then the *round* setting is reset to **half_even**.

Default: **half_even**.

showcurrency

Sets whether the appropriate currency symbol is shown. This is only valid for a currency data type. Valid settings are on, which shows the symbol, and off, which does not.

The symbol used is determined by the *currencylocale* setting. If there is no *currencylocale* setting, it defaults to normal currency symbol for the current locale.

Default: **on**.

significantdigits

Sets the number of significant digits allowed. This is generally the total number of digits allowed in the number. For example, 134.56 has five significant digits.

If the data entered exceeds the number of significant digits allowed, then only the least significant digits are shown. For example, if you allow five significant digits and 12,345.56 is entered, then only 345.56 is shown.

significantdigits is only valid for integer, float, and currency data types.

Note that setting both *fractiondigits* and *significantdigits* may cause conflicting formats. In this case, *significantdigits* takes precedence.

Default: the maximum number of digits allowed for the data type.

style Sets how various data types are displayed. For example, you can use the style to set whether times include seconds, and whether dates are spelled out or numeric.

Valid settings are:

- **numeric**
- **short**
- **medium**
- **long**
- **full**

For more information about how the styles affect the different data types, see the "Data Type Styles."

Note that both the pattern and patternrefs settings override the style setting.

Default: **medium**.

Data Type Styles

The following table shows how the style affects the presentation of various data types in the en_US locale. For other locales, refer to the *IBM Workplace Forms Locale Specification for XFDL*. The symbols used to define each format are explained on page 101.

Data Type	Style	Format	Example
`date`	numeric	yyyyMMdd	20041123
	short	yyyy-MM-dd	2004-11-23
	medium	d MMM yyyy	23 Nov 2004
	long	MMMM d, yyyy	November 23, 2004
	full	EEEE, MMMM d, yyyy	Tuesday, November 23, 2004
`day_of_month`	numeric	d	3
	short	d	3
	medium	d	3
	long	d	3
	full	d	3
`day_of_week`	numeric	e	3
	short	e	3
	medium	EEE	Wed
	long	EEEE	Wednesday
	full	EEEE	Wednesday
`date_time`	numeric	yyyyMMdd h:mm	20041123 8:15
	short	yyyy-MM-dd h:mm a	11/23/04 8:15 AM
	medium	d MMM yyyy h:mm:ss a	Nov 23, 2004 8:15:23 AM
	long	MMMM d, yyyy h:mm:ss a	November 23, 2004 8:15:23 AM
`month`	numeric	M	9
	short	M	9
	medium	MMM	Sep
	long	MMMM	September

Data Type	Style	Format	Example
	full	MMMM	September
`time`	numeric	H.mm	17.30
	short	H:mm a	17:30 PM
	medium	h:mm:ss a	5:30:14 AM
	long	h:mm:ss a	5:30:14 AM
`year`	numeric	yyyy	2004
	short	yy	04
	medium	yyyy	2004
	long	yyyy	2004
	full	yyyy G	2004 AD

Constraint Settings

Constraint settings control the text that the user is allowed to input. For example, you can limit input to a range of numbers, to a particular length, or to a specific pattern, such as the common ###-####.

Constraint settings follow this syntax:

```
<format>
   <constraints>
      <settingname1>setting</settingname1>
      ...
      <settingnamen>setting</settingnamen>
   </constraints>
</format>
```

Note:
- You can define any number of settings.

The following list defines that valid settings:

casesensitive
Sets whether the data entered must match the case of the defined pattern constraints. Valid settings are:
- **on** — The data entered must match the case of the defined templates.
- **off** — The data entered does not need to match the case of any defined templates.

Default: **off**.

checks
Allows you to force the format check to fail, or to ignore all constraints settings. Valid settings are:
- **fail** — Forces the format check to fail.
- **ignore** — Ignores all constraint settings. Note that the data type and the presentation settings are still respected.
- **none** — Has no effect.

Default: none.

decimalseparators

Defines one or more symbols that are allowed to indicate the decimal place. This is often a period, as shown:

```
100.00
```

List each separator in its own separator tag, as shown:

```
<decimalseparators>
   <decimalseparator1
      >separator</decimalseparator1>
   ...
   <decimalseparatorn
      >separator</decimalseparatorn>
</decimalseparators>
```

You can use any string, such as a comma or a comma followed by a space. This setting is only valid for integer, float, and currency data types.

Note:

- The user must use the same separator in a given string. For example, if you define both comma and space as valid separators, the user must type either 1,000,000 or 1 000 000. Mixing the separators, as in 1,000 000, is not allowed.
- If this setting is empty, it inherits the decimalseparator defined in the presentation settings.

Default: a comma.

groupingseparators

Defines one or more symbols that are allowed to separate groups of numbers (such as thousands in North America) during input. This is often a comma, as shown:

```
1,000,000
```

List each separator in its own separator tag, as shown:

```
<groupingseparators>
   <groupingseparator1
      >separator</groupingseparator1>
   ...
   <groupingseparatorn
      >separator</groupingseparatorn>
</groupingseparators>
```

You can use any string, such as a comma or a comma followed by a space, with the keyword none representing no separator at all. This setting is only valid for integer, float, and currency data types.

Usage Details

- The user must use the same separator in a given string. For example, if you define both comma and space as valid separators, the user must type either 1,000,000 or 1 000 000. Mixing the separators, as in 1,000 000, is not allowed.
- If this setting is left empty, it inherits the groupingseparator defined in the presentation settings.

Default: a comma.

length Sets a range of lengths that the data entered must fall within. To do this, you must include the <min> and <max> tags in your definition, as shown:

```
<length>
   <min>shortest length allowed</min>
   <max>longest length allowed</max>
</length>
```

For example, if you wanted all values to be between 4 and 7 characters in length, you would set the min to 4 and the max to 7. This allows the user to enter a value that is either 4 characters or 7 characters in length, as well as all lengths in between.

The length is calculated after all formatting has been applied, and will include all formatting characters such as the negative sign, currency symbols, and so on.

If you add a *length* setting to a field, the field is treated as mandatory.

Default: the maximum range of lengths allowed for the data type.

mandatory
Sets whether the user must enter a value. Valid settings are:

- **on —** The user must enter a value.
- **off —** The user need not enter a value.

This value works in conjunction with the *required* property for the linked element in the XForms model. If either setting indicates that input is mandatory, then it is mandatory.

Default: the *required* property of the linked XForms data element, or **off**.

message
Sets the message that is displayed when the input is invalid. This can be any text.

Default: **none**.

patterns
Allows you to set one or more patterns for strings, date, or numbers that are are valid as input. For example, you might want to constrain dates to the following format: YYYY-MM-DD.

You must define each pattern in its own <pattern> tag, as shown:

```
<patterns>
   <pattern1>pattern</pattern1>
   ...
   <patternn>pattern</patternn>
</patterns>
```

To learn how to represent different patterns, refer to “Defining Patterns” on page 101.

If you define more than one *patternref* in the presentation settings, you must define an equal number of patterns in the constraints. Each *patternref* is then matched to the corresponding constraint *pattern*. For example, the first *patternref* is matched to the first constraint *pattern*, the second *patternref* is matched to the second constraint *pattern*, and so on. This allows you to define a different pattern for each constraint.

Note that unlike the *template* setting, the *pattern* setting will not show users any of the text you include in your patterns, since there is no way to tell which pattern the user will follow.

range Sets a numerical range that the data entered must fall within. To do this, you must include the <low> and <high> tags in your definition, as shown:

```
<range>
   <min>smalled number allowed</min>
   <max>highest number allowed</max>
</range>
```

The low and high values are inclusive. For example, if you wanted to create a range from 1 to 100, you would set the low value to 1 and the high value to 100. This allows the user to enter either 1 or 100, as well as all values in between.

If you set a range for a string, the data is evaluated on a character by character basis. For example, you might set your low value to "fg" and your high value to "jk". I this case, the first character entered would have to be in the f-j range, and the second character would have to be in the g-k range. This check ignores case.

If you add a *range* setting to a field, the field is treated as mandatory.

Default: the maximum range allowed for the data type.

template

Allows you to display symbols in the input area before the user enters their data. This is useful if you want to show formatting placeholders, such as parentheses for the area code in a phone number.

To create a template, use a period to represent any 1 character that the user types in. All other characters are shown to the user as typed.

For example, if you create the following template:

```
(...)...-....
```

The user will see the following:

```
(   )   -
```

Setting a template in no way limits the user input. If you want to limit the user input, you must also use the *patterns* setting. Futhermore, you can only set one template.

yearwindow

Sets how to interpret two digit dates. This provides two options for interpreting dates:

- **Fixed Date —** You can specify a specific year, such as 70. All numbers from that year and up are assumed to be in the 20th century (for example, 1975). All numbers before that are assumed to be in the 21st century (for example, 2004).

  To set a fixed date, you must include the <fixedyear> tag as shown:

  ```
  <yearwindow>
    <fixedyear>the year</fixedyear>
  </yearwindow>
  ```

- **Sliding Date —** You can specify a range rather than a fixed date. This means that the date on which the decision is based changes as time passes. The date is calculated by taking the current date and subtracting a number you specify. For example, if you set your range to 30 years and it is 2004, your decision date would be 2004 - 30 = 1974. In this case, all numbers from 74 and up would be in the 20th century, and all numbers below 74 would be in the 21st century.

To set a sliding date, you must include the <factor> tag as shown:

```
<yearwindow>
   <factor>range</factor>
</yearwindow>
```

You can set either a fixed date or a sliding date, but not both. If you do set both, the sliding date will override the fixed date.

Default: a sliding date with a factor of 80.

Defining Patterns

When defining a pattern or patternref, you must create a template for that pattern. For example, phone numbers commonly follow this template: (000)000-0000. The following sections explain how to create the following patterns:

- Date Patterns
- Number Patterns
- String Patterns

Date Patterns

The following symbols are used to create date patterns:

Symbol	Description	Example
`G`	The era, expressed as AD or BC.	AD
`y`	The year.	1997
`u`	The extended year.	4601
`M`	The month.	11
`d`	The day of the month.	23
`h`	The hour for a twelve-hour clock (1-12).	11
`H`	The hour for a twenty-four hour clock (0-23).	23
`m`	The minute of the hour (1-59)	34
`s`	The second of the minute (1-59).	12
`S`	The fractional second, expressed as a decimal value.	234
`E`	The day of the week, as text.	Tuesday
`e`	The day of the week, as a number (1-7).	2
`D`	The day of the year (1-366).	234
`F`	The occurrence of that weekday in the month (1-5). For example, the second Wednesday in the month.	2
`w`	The week in the year (1-52).	27
`W`	The week in the month (1-5).	3
`a`	The meridiem, expressed as AM or PM.	AM

Symbol	Description	Example
`k`	The hour in the day (1-24).	23
`K`	The hour in the day (0-11)	3
`g`	The Julian day.	2451334
`A`	The millisecond in the day.	69540000
`'`	Use to enclose text you want to display.	'Date='
`''`	Use to write a single quote as part of text.	'o''clock'

When creating date patterns, you can repeat the placeholder to determine which format to use. For example, a single e represents the day of the week as a single digit, such as 3. Two e's (ee) represents the day of the week as two digits, such as 03. Three E's (EEE) represents the day of the week as short text, such as Wed. And finally, four E's (EEEE) represents the day of the week as full text, such as Wednesday.

Number Patterns

The following symbols are used to create number patterns:

Symbol	Description	Example
`0`	Use to specify a digit that must appear. For example, 0.00 would require input with a single digit before the decimal place, and two digits after. Similarly, #0.00 would allow one or more digits before the decimal place, and two digits after.	#0.00
`@`	Use to specify the number of significant digits to show. significant digits are the largest value digits in the number. For example, in the number 12345, the 1 is the most significant, the 2 is the second most, and so on. Typing that number into a template of @@@ would produce the number 12300. A significant digit is always shown, even if its value is zero. Furthermore, you cannot use this symbol with a decimal value.	@@@
`#`	Represents zero or more digits. For example, #.# would accept any of the following values: 1, 1.1, 0.1, or 123.34.	#.#
`.`	Decimal separator.	#.#
`1-9`	Each number represents a digit that must appear, and is used to set the increment for rounding. This means that #5 would round the number to the nearest five. Similarly, #29 would round the number to the nearest multiple of 29. For example, if you set a pattern of #35 and the user typed 138, the number would be rounded to 140 (the nearest multiple of 35).	#5

Symbol	Description	Example
-	A negative indicator. Note that this is a placeholder for the characters defined in the *negativeindicator* setting. For example, if you defined your negative indicator as parentheses, then -#.# would result a value like: (123.45).	-#.#
,	A separator indicator, representing the character used to separate increments of one thousand in numbers. Note that this is a placeholder for the characters defined in the separator setting. For example, if you declared your separator as a comma followed by a space, then 0,000 would result in a value like: 4, 000.	0,000
`\u00A4`	A currency indicator. Note that this is a placeholder for the indicator defined in the *currencylocale* setting. For example, if you declared your currencylocale to be the US, and your template was \u00A4#0.00, you would get a value like: $534.23. If this symbol appears twice, it is replaced by the international currency symbol.	\u00A4#0.0
E	Separates the mantissa from the exponent in scientific notation. For example, 0.#E# would result in a value like: 1.23E4 Note that when using # in scientific notation, this represents the number of digits that will always appear after the decimal. So 0.# will result in one digit after the decimal, while 0.### will result in three digits.	0.#E#
+	Use this to prefix positive exponents with the plus sign. For example, 0.#E+# would result in a value like: 1.34E+4.	0.#E+#
;	Separates the positive and negative versions of a pattern. For example, if you wanted a pattern of #.# or -#.#, you would declare: #.#;-#.#	#.#;-#.#
%	Multiply the data by 100 and show as a percentage. For example, if you set a template of #% and entered a value of 0.12, you woud get: 12%.	#%
`\u2030`	Multiply the data by 1000 and show as per mille. For example, if you set a template of #.#\u2030 and entered a value of .123, you would get: 123 per mille.	#.#/u2030
*	Precedes a pad character, which you can use to insert specific symbols. For example, *0##.## would result in a value like: 012.23.	*0##.##

String Patterns

All string patterns are written with Unix style regular expressions. Regular expressions are well-defined through a variety of public sources (such as www.regular-expressions.info), and as such are not discussed in detail in this document.

When using regular expressions, be aware that the patternrefs you set for the presentation are intended to match corresponding patterns in your constraints. This means that you can define groups in your constraints, and then refer to those groups from the presentation setting using the standard $# notation.

Examples

This example specifies a field containing integer data with a range of values from 10 to 1,000 inclusive, and formatted with commas separating the thousands:

```
<format>
   <datatype>integer</datatype>
   <presentation>
      <groupingseparator>,<groupingseparator>
   </presentation>
   <constraints>
      <range>
         <min>10</min>
         <max>1000</max>
      </range>
   </constraints>
</format>
```

This example specifies a field that contains currency data that is mandatory. An error message appears if the data is not entered correctly.

```
<format>
   <datatype>currency</datatype>
   <constraints>
      <mandatory>on</mandatory>
      <message>Entry incorrect -- try again.</message>
   </constraints>
</format>
```

This example specifies a field in which date data will be formatted as month, day-of-month, and year (for example, November 23, 2004):

```
<format>
   <datatype>date</datatype>
   <presentation>
      <style>long</style>
   </presentation>
</format>
```

This example sets up a template and patterns for both presentation and constraints. The template sets up a format of (###) ###-#### for a telephone number. This means that when the field is first displayed, it will show the parentheses and the dash to the user. The constraint pattern uses a regular expression to create the same pattern, thereby limiting the input to match the template. Finally, the presentation patternref uses a regular expression to define how the input should be formatted when displayed on the screen. This expression refers to the groups defined in the constraint pattern.

```
<format>
   <datatype>string</datatype>
   <constraints>
      <template>(...) ...-....</template>
      <patterns>
         <pattern>\((\d{3})\)\s(\d{3})-(\d{4})</pattern>
      </patterns>
   </constraints>
   <presentation>
      <patternrefs>
         <patternref>($1) $2-$3</patternref>
      </patternrefs>
   </presentation>
</format>
```

Usage Details

1. Default datatype: **string**.
2. Default presentation:
   - calendar — **gregorian** (en_US locale)
   - casetype — **none**
   - currencylocale — the locale of the form
   - decimalseparator — period (**.**) (en_US locale)
   - fractiondigits — maximum number of digits allowed by data type
   - negativeindicator — minus sign (-) (en_US locale)
   - pad — **0**
   - padcharacter — **0** (en_US locale)
   - pattern — n/a
   - round — **up**
   - groupingseparator — comma (**,**) (en_US locale)
   - showcurrency — **on**
   - significantdigits — the maximum number of digits allowed for the data type
   - style — **medium**

   Note: The default values for other locales are listed in the *IBM Workplace Forms Locale Specification for XFDL*.
3. Default constraints:
   - casesensitive — **off**
   - checks — **none**
   - decimalseparator — period (**.**) (en_US locale)
   - length — maximum range of lengths allowed for the data type
   - mandatory — the *required* property of the linked XForms data element, or **off**
   - message — the <xforms:alert> setting for the item, if present.
   - patterns — n/a
   - range — the maximum range allowed by the data type
   - groupingseparators — comma (**,**)
   - template — n/a
   - yearwindow — a sliding date with a factor of 30
4. In some cases, it's possible to create formatting that will have unpredictable results. For example, if you specify that the grouping separator should be a period, this may cause problems since the decimal separator is also a period. Use good judgement when defining your formats.

5. All constraints are applied to the input data. This may create an item the user cannot complete. For example, the combination of data type date and constraint pattern of #.# creates such a situation. A date type cannot be formatted as a decimal number.
6. You should use caution if you are designing forms that use two digit dates. While the *yearwindow* setting provides a mechanism for interpreting two digit dates, the best solution is to use four digit dates.
7. When applying a format to a *combobox*, *list*, or *popup*, the formatting will be applied to the value of each cell linked to the item. Those cells that do not pass the check will be flagged or filtered. If a cell passes the checks, its value will be replaced with a formatted value before the item is displayed. The *label* option for these cells will remain unaffected.
8. When applying a format to a *combobox*, *list*, or *popup* item, a cell with an empty value will fail all format checks but will still be selectable, even if input is mandatory. This allows users to erase their previous choice (which will also reset all formulas based on that choice). However, users will still need to select a valid cell before they can submit the form.
9. If any two *comboboxes*, *lists*, or *popups* use the same set of cells, they must apply the same formatting.
10. The void data type disables a format line completely through the use of a compute. Void formats never fail regardless of the checks in the format statement.
11. For details on using the *format* option in buttons, see the Usage Details in the button item description.
12. The message constraint overrides the <xforms:alert> setting for the item.
13. An item is mandatory if either the mandatory constraint is set to true or the required property for a bound data element is set to true.
14. If an element in the XForms data model is both empty and invalid, then any item on the form that is bound to that element is set to be mandatory.

formid

Defines a unique identifier for the form.

Syntax

```
<formid>
   <title>form title</title>
   <serialnumber>serial number</serialnumber>
   <version>version number</version>
</formid>
```

Note:

- serialnumber and version are mandatory; title is optional

form title	*string*	provides a title for the form
serialnumber	*string*	provides a unique identifying string for the form. This is generated by an external program such as a form design program.

version number	*number in the format AA.Bb.cc*	shows the major (AA), minor (Bb) and maintenance (cc) numbers for the form. This is generated by an external program such as a form design program, but can also be set manually.

Available In

form global

Example

This sample shows how the formid option appears in the global characteristics of a form.

```
<formid>
   <title>Admin_Form</title>
   <serialnumber>{94EC8BA0-7D33-11D2-B5E3-0060}</serialnumber>
   <version>4.8.2</version>
</formid>
```

Usage Details

1. Defaults:
   - title: **none**
   - serialnumber: **none**
   - version : **1.0.0**
2. This option is intended for use with form design programs and licensign models.

fullname

Used in a signature item to record the fully qualified name of the signer. This name is retrieved from the digital certificate used to the sign the form.

Syntax

```
<fullname>name</fullname>
```

name	*string*	the fully qualified name of the signer, as supplied by the digital certificate used to sign the form

Available In

signature

Example

This sample shows a fullname option as part of a signature item:

```
<signature sid="empsignature">
   <signer>Jane D Smith, jsmith@insurance.com</signer>
   <fullname>
      "Verisign, Inc.", Verisign Trust Network,
      "www.verisign.com/repository/RPA Incorp. by
      Ref.,LIAB.LTD(c)98", Persona Not Validated,
      Digital ID Class 1 - Microsoft, Jane D Smith,
      jsmith@insurance.com
   </fullname>
</signature>
```

Usage Details

1. This option is added to the form during the signing process, and is created by the software that enables signing (such as a form viewer).

group

Provides a way of associating related items. There are two ways of using this option. In the first case, it enables you to to create groups of cells or radio buttons. In the second case, the *group* option enables you to populate lists, popups, and comboboxes by referencing a group of cells. Items with the same group reference are considered members of the same group.

Syntax

```
<group>group reference</group>
```

group reference	*string*	identifies the group. Can be one of: • *group_name* for groups on the current page • *page_sid.group_name* for groups on other pages

Available In

cell, combobox, list, popup, radio

Example

In the following sample, the group option creates a group called ″LIST1_GROUP″ containing the cell items ″CELL1″ and ″CELL2″.

```
<cell sid="CELL1">
   <group>LIST1_GROUP</group>
   <value>red</value>
   <type>select</type>
</cell>
<cell sid="CELL2">
   <group>LIST1_GROUP</group>
   <value>green</value>
   <type>select</type>
</cell>
```

In the following code, the *group* option is used to populate the list item with the cells from the the preceding sample.

```
<list sid="LIST1">
   <group>LIST1_GROUP</group>
   <label>Colors</label>
   <value></value>
</list>
```

Usage Details

1. Default: **none**
2. The association of cells or radio buttons into groups cannot span multiple pages. That is, all the cells or radio buttons assigned to a given group must belong to the same page. On the other hand, you are allowed to populate a list, popup, or combobox with a group of cells defined on another page, as long as you specify the page sid in the group reference.
3. *List* and *popup* items are populated with cells that have the same group reference as the item. It is possible to have multiple *list* and *popup* items with the same group reference. In this way, the same group of cells can populate more than one list or popup.
4. All *radio* items having the same group reference will form a mutually exclusive group.

help

Points to the help message for the item. The *item reference* identifies the *help* item containing the help message. There can be many items pointing to the same help message.

Syntax

```
<help>item reference</help>
```

item reference	*string*	a reference to the help item that contains the help message.

Available In

button, check, checkgroup, combobox, field, label, list, popup, radio, radiogroup, slider

Example

This sample points to the help item general_help defined on the page called page_1.

```
<help>page_1.general_help</help>
```

Usage Details

1. Default
   - XFDL: **none**
   - XForms: if no help item is referenced, then the concatenated value of the <xforms:hint> and <xforms:help> settings are used to create a help message that is shown for the item. For more information about these settings, refer to "Metadata Sub-Options" on page 210.
2. The *help* option overrides the <xforms:hint> and <xforms:help> settings for the item.

image

Associates an image with an item. The *item reference* identifies the *data* item containing the image. This image replaces any text label if the viewer is able to display images.

Syntax

```
<image>item reference</image>
```

item reference	*string*	identifies the data item

Available In

button, label

Example

This sample points to the data item company_logo defined on the page called page_lst.

```
<image>page_1st.company_logo</image>
```

Usage Details

1. Default: **none**
2. Use this option to associate images with *button* and *label* items.
3. If an enclosure mechanism is used to replace an image stored in a *data* item with a new image, then buttons and labels whose *image* option is set to the identifier of the image data item will be updated to display the new image. For details, see the *data* option description.
4. Use the *imagemode* option to control the display behavior of the image.

imagemode

Defines how the image will be displayed in the item.

Syntax

```
<imagemode>image mode</imagemode>
```

image mode	*clip*	if the image is smaller than the item, the image is centered in the item. Otherwise, the image is placed in the item from the top left corner and the parts of the image that extend past the item's bounding box are cut off.
	resize	image is placed in the item from the top left corner. The image is then expanded or contracted in both directions so that it fits the item exactly.

scale	if the image is smaller than the item, the image is centered in the item. Otherwise, the image is placed in the item from the top left corner. The image is expanded or contracted, keeping the original aspect ratio, to the point at which one of the sides fits snugly in the item and the other side is smaller than the item.

Available In

button, label

Example

This sample displays a company logo as a resized image.

```
<image>page_1st.company_logo</image>
<imagemode>resize</imagemode>
```

Usage Details

1. Default: **resize**
2. An imagemode of **clip** draws the image in the upper left corner of the item. If the image is too big for the item's space, and the image is clipped at the item's edge. If the image is smaller than the item in either horizontally or vertically, the image is centered appropriately.
3. An imagemode of **resize** resizes the image to be the exact size of the item, whether that means an increase in size, decrease in size, or an increase in one dimension and decrease in the other.
4. An imagemode of **scale** also resizes the image to fit the item, except that it will preserve the aspect ratio of the original image. If either dimension of the image is larger than the item, then the image will be made small enough to fit in the given space as follows:
   - The larger dimension will fit snugly in the space.
   - The other dimension will be scaled by the same factor.

     Likewise, if both dimensions of the image are smaller than the space given by the button or label, then the image will be expanded to fit the given space as follows:
   - The larger dimension will fit snugly in the space.
   - The other dimension will be scaled by the same factor.
5. Use this option with the *image* option to control the image's appearance with button and label items.

itemfirst

Identifies the first item on the page, excluding the global item. An item is *first* when it appears first in the build order (in other words, it is first in the XFDL text).

This option is not saved or transmitted as part of the form. Instead, it is automatically created each time the form is read into memory, and is maintained only during display or processing.

Syntax

```
<itemfirst>item reference</itemfirst>
```

item reference	*string*	a reference to the first item on the page, excluding the global item.

Available In

page global

Example

This sample shows how the *itemfirst* option appears in the page globals of a form.

```
<page sid="PAGE1">
   <global sid="global">
      <itemfirst>LABEL1</itemfirst>
   </global>
</page>
```

Usage Details

1. Defaults: **none**
2. The *itemfirst* option is not saved or transmitted with form descriptions.

itemlast

Identifies the last item on the page, excluding the global item. An item is *last* when it appears last in the build order (in other words, it is last in the XFDL text).

This option is not saved or transmitted as part of the form. Instead, it is automatically created each time the form is read into memory, and is maintained only during display or processing.

Syntax

```
<itemlast>item reference</itemlast>
```

item reference	*string*	a reference to the last item on the page, excluding the global item.

Available In

page global

Example

This sample shows how the *itemlast* option appears in the page globals of a form.

```
<page sid="PAGE1">
   <global sid="global">
      <itemlast>BUTTON2</itemlast>
   </global>
</page>
```

Usage Details

1. Defaults: **none**
2. The *itemlast* option is not saved or transmitted with form descriptions.

itemlocation

Serves two purposes:

- It specifies the location of an item in the page layout.
- It allows you set the size of the item, either in relation to another item, or in absolute terms.

Itemlocation offers three ways to position items on the page: absolute positioning, relative positioning, and offset positioning. Absolute positioning anchors the top left corner of an item to a particular location on the page, using an x-y coordinate. For example, you might place an item 10 pixels in from the left margin, and 10 pixels down from the top of the page. Relative positioning places items on the page in relation to one another. For example, it might place one item below another. Finally, offset positioning allows you to place an item on the page relative to another item, and then move it a set amount. For example, you might place an item below another, and then move it 10 pixels to the right.

Itemlocation also provides two ways to the set the size for an item: relative positioning and extent sizing. Relative positioning allows you set the size of an item relative to another item on the page. For example, you might expand an item so that it's right edge lines up with the right edge of a different item. Extent sizing allows you to set the absolute size of an item, to the pixels. For example, you might set an item to be 100 pixels wide and 30 pixels tall.

Note that you can also combine these methods for positioning and sizing. For example, you might place an item on the form using absolute positioning, and then place a second item below the first using relative positioning.

Syntax

```
<itemlocation>
   settings
</itemlocation>
```

settings	*(see below)*	• the setting describes where to position the item and how to size it. This can take four forms: absolute positioning, relative positioning, offset positioning, and extent sizing. These methods are described in more detail below.

Available In

box, button, check, checkgroup, field, label, line, list, popup, combobox, radio, radiogroup, slider, spacer

Absolute Positioning

Absolute positioning places the item at a specific x-y coordinate on the page. This location is measured from the top left cornner of the page to the top left corner of the item, and is expressed in pixels. When using absolute positioning, the syntax is:

```
<itemlocation>
   <x>x-coordinate</x>
   <y>y-coordinate</y>
</itemlocation>
```

x-coordinate	*short (must be positive)*	• the horizontal distance in pixels from the form's top left corner
y-coordinate	*short (must be positive)*	• the vertical distance in pixels from the form's top left corner

Example

This sample places a label on the page so that its top left corner is 20 pixels in from the page's left edge, and 30 pixels down from the top of the page.

```
<label sid="persInfo_label">
   <value>Personal Information</value>
   <itemlocation>
      <x>20</x>
      <y>30</y>
   </itemlocation>
</label>
```

Relative Positioning

Relative positioning allows an item to be placed relative to the location of another item. It also allows for the specification of an item's size relative to the size and location of other items. The other items (called reference points or anchor items) must be defined earlier in the XFDL form description before they can be used in an *itemlocation* statement.

When using the relative positioning scheme, the first external item placed on the form appears in the top left corner. It cannot be placed in relation to any other item, since no other items exist. All subsequent items can be placed in relation to items that appear before them in the form's description. If no relational position for an item is specified, it will appear below the previous item, with its left edge against the page's left edge.

When using relative positioning, the syntax is:

```
   <itemlocation>
      <modifier>item reference</modifier>
   </itemlocation>
or
   <itemlocation>
      <modifier>
         <itemref>item reference</itemref>
         <itemref>item reference</itemref>
      </modifier>
   </itemlocation>
```

Note:

- the second syntax is used when an item is positioned between two other items.

modifier	*(see below)*	• determines where in relation to the reference point the item is placed. For example, you can specify that the item goes "below" the reference item. The available modifiers are described in more detail below.
item reference	*string*	• identifies the reference point item. This item must be on the same page, or within the same toolbar, as the item you are placing.

Modifiers

There are three types of modifiers:

- position modifiers - used to position an item
- alignment modifiers - used to align one edge of an item (relative positioning only)
- expansion modifiers - used to alter an item's size (relative positioning only)

Position Modifiers

above Places item a small distance above reference point item; aligns left edges.

after Places item a small distance after reference point item; aligns top edges.

before Places item a small distance before reference point item; aligns top edges.

below places item a small distance below reference point item; aligns left edges.

offset Places item so that it is offset from its original location by the measurement specified in the x-coordinate and y-coordinate settings.

within
Assigns item to the toolbar. Note that the within modifier must appear before any other modifiers.

Alignment Modifiers

alignb2b
Aligns bottom edge of item with bottom edge of reference point item.

alignb2c
Aligns bottom edge of item with vertical center of reference point item.

alignb2t
Aligns bottom edge of item with top edge of reference point item.

alignc2b
: Aligns vertical center of item with bottom edge of reference point item.

alignc2l
: Aligns horizontal center of item with left edge of reference point item.

alignc2r
: Aligns horizontal center of item with right edge of reference point item.

alignc2t
: aligns vertical center of item with top edge of reference point item.

alignhoriz between
: Aligns horizontal center of item between right edge of first reference point item and left edge of second reference point item.

alignhorizc2c
: Aligns horizontal center of item with horizontal center of reference point item; center below. Note that this modifier requires two reference points.

alignl2c
: Aligns left edge of item with horizontal center of reference point item.

alignl2l
: Aligns left edge of item with left edge of reference point item.

alignl2r
: Aligns left edge of item with right edge of reference point item.

alignr2c
: Aligns right edge of item with horizontal center of reference point item.

alignr2l
: Aligns right edge of item with left edge of reference point item.

alignr2r
: Aligns right edge of item with right edge of reference point item.

alignt2b
: aligns top edge of item with bottom edge of reference point item.

alignt2c
: Aligns top edge of item with vertical center of reference point item.

alignt2t
: Aligns top edge of item with top edge of reference point item.

alignvertbetween
: Aligns vertical center of item between bottom edge of first reference point item and top edge of second reference point item. Note that this modifier requires two reference points.

alignvertc2c
: Aligns vertical center of item with vertical center of reference point item.

Expansion Modifiers

expandb2b
: Aligns bottom edge of item with bottom edge of reference point item.

expandb2c
: Holds top edge of item constant and expands bottom edge to align with vertical center of reference point item.

expandb2t
: Holds top edge of item constant and expands bottom edge to align with top edge of reference point item.

expandl2c
: Holds right edge of item constant and expands left edge to align with horizontal center of reference point item.

expandl2l
: Holds right edge of item constant and expands left edge to align with left edge of reference point item.

expandl2r
: Holds right edge of item constant and expands left edge to align with right edge of reference point item.

expandr2c
: Holds left edge of item constant and expands right edge to align with horizontal center of reference point item.

expandr2l
: Holds left edge of item constant and expands right edge to align with left edge of reference point item.

expandr2r
: Holds left edge of item constant and expands right edge to align with right edge of reference point item.

expandt2b
: Holds bottom edge of item constant and expands top edge to align with bottom edge of reference point item.

expandt2c
: Holds bottom edge of item constant and expands top edge to align with vertical center of reference point item.

expandt2t
: Holds bottom edge of item constant and expands top edge to align with top edge of reference point item.

Examples

This sample aligns the vertical center of an item between the bottom edge of the item label_one and the top edge of the item label_two.

```
<itemlocation>
   <alignvertbetween>
      <itemref>label_one</itemref>
      <itemref>label_two</itemref>
   </alignvertbetween>
</itemlocation>
```

This sample aligns the item's left edge with the center of item the_firm and expands the right edge to align with the right edge of the same reference item (the_firm).

```
<itemlocation>
   <alignl2c>the_firm</alignl2c>
   <expandr2r>the_firm</expandr2r>
</itemlocation>
```

This sample assigns an item to the toolbar main_toolbar and positions it under the company logo company_logo.

```
<itemlocation>
   <within>main_toolbar</within>
   <below>company_logo</below>
</itemlocation>
```

Offset Positioning

The relative positioning scheme also allows an item to be offset from its original position, by a particular number of pixels. This is a quick way to create an indented layout on a form. You can offset an item in any of these four directions: right, left, up, down. The offset is expressed as an x and y value, and follows the relative positioning statement, as shown:

```
<itemlocation>
   relative positioning
   <offsetx>x-offset</offsetx>
   <offsety>y-offset</offsety>
</itemlocation>
```

relative positioning	*(see above)*	• sets the position of the item relative to one or more items on the form.
x-offset	*integer*	• the number of pixels to move the item along the x axis. A positive number moves the item to the right, a negative number moves the item to the left.
y-offset	*integer*	• the number of pixels to move the item along the y axis. A positive number moves the item down, a negative number moves the item up.

Example

This sample places an item below a label called persInfo_label, and then uses offset to move the item 15 pixels to the left and 20 pixels down:

```
<itemlocation>
   <below>persInfo_label</below>
   <offsetx>-15</offsetx>
   <offsety>20</offsety>
</itemlocation>
```

Extent Sizing

Extent sizing provides a different way to set the size of an item, and works in conjunction with either absolute or relative positioning. Unlike the *size* option, which sets the size of an item in terms of the number of internal characters, you can use extent sizing to set the absolute size of the exterior, or bounding box, of an item. Futhermore, using extent sizing overrides the *size* option.

Extent sizing sets the size of an item in pixels, using separate values for the width and the height. When an extent is specified, the item's top left corner will stay where it is, and the item will be resized so that it is as many pixels wide as the width value and as many pixels tall as the height value.

Extent sizing uses the following syntax:

```
<itemlocation>
   positioning
   <width>width</width>
   <height>height</height>
</itemlocation>
```

positioning	*(see above)*	• sets the position of the item on the form. This can be either absolute or relative positioning.
width	*integer*	• the width of the item, in pixels.
height	*integer*	• the height of the item, in pixels.

Example

This sample shows an extent setting on a field that has been placed using absolute positioning. The field is first placed at an x-y coordinate of 10, 10. It is then sized to be 300 pixels wide and 30 pixels high.

```
<itemlocation>
   <x>10</x>
   <y>10</y>
   <width>300</width>
   <height>30</height>
</itemlocation>
```

Usage Details

1. Default item location:
   - in the body of the page
   - under the previous item in the page definition
   - aligned along the left margin of the page
2. Itemlocation overrides size. If the itemlocation affects the size of the item (using extent sizing) and the *size* option has also been set for the item, the itemlocation will determine the size.
3. There are two measurements for sizing items when using absolute positoning: in pixels (using extent sizing) or in characters (using the *size* option). If you choose to size items using characters, you should be aware that different platforms and video cards use differently sized fonts. Even with so-called cross-platform fonts, an item's actual size (in pixels) might change from one platform to the next if it is sized using character height and width. As a result, absolutely positioned items sized with characters may have overlap or sizing problems if displayed on different platforms, different video cards, or in both small font and large font modes. To ensure that forms appear the same on any platform, and under any video card or font mode while using absolute positioning, size items in pixels or inches.
4. An item's vertical center is halfway between the top and bottom edges. The horizontal center is halfway between the left and right edges.

itemnext

Identifies the next item on the page, excluding the global item. An item is *next* when it appears next in the build order (in other words, it is next in the XFDL text).

This option is not saved or transmitted as part of the form. Instead, it is automatically created each time the form is read into memory, and is maintained only during display or processing.

Syntax

```
<itemnext>item reference</itemnext>
```

item reference	*string*	a reference to the next item on the page, excluding the global item.

Available In

action, box, button, cell, check, checkgroup, combobox, data, field, help, label, line, list, popup, radio, radiogroup, signature, slider, spacer, toolbar

Example

The following example shows what two labels look like in memory, with the *itemnext* option inserted in each:

```
<label sid="LABEL2">
   <value>This is a label.</value>
   <itemnext>LABEL3</itemnext>
</label>
<label sid="LABEL3">
   <value>This is a label.</value>
   <itemnext>LABEL4</itemnext>
</label>
```

You can use computes to determine the next item on the form. For example, the following code shows a label that uses a compute set its value. The compute goes to the item that currently has the focus, then copies the value of the *itemnext* option from that item:

```
<label sid="nextItemName">
   <value compute="PAGE1.global.focuseditem->itemnext"></value>
</label>
```

Usage Details

1. Defaults: **none**
2. If the *itemnext* option is in the last item on the page, it points to the first item on the page (excluding the global item).
3. When the *itemnext* option is used in items that appear in table rows:
   - Each item points to the next item in the row.
   - The last item in the row points to the first item in the next row.
   - The last item in the table it points to the first item following the table.
4. When working with panes, tables, radiogroups, or checkgroups:
   - The item that precedes a table, pane, radiogroup, or checkgroup points to the table, pane, radiogroup, or checkgroup.
   - The last item generated by a pane, table, radiogroup, or checkgroup points to the first item following the containing item.
5. The *itemnext* option is not saved or transmitted with form descriptions.

itemprevious

Identifies the previous item on the page, excluding the global item. An item is *previous* when it immediately precedes the current item in the build order (in other words, it comes immediately before the current item in the XFDL text).

This option is not saved or transmitted as part of the form. Instead, it is automatically created each time the form is read into memory, and is maintained only during display or processing.

Syntax

```
<itemprevious>item reference</itemprevious>
```

item reference	*string*	a reference to the previous item on the page, excluding the global item.

Available In

action, box, button, cell, check, checkgroup, combobox, data, field, help, label, line, list, popup, radio, radiogroup, signature, slider, spacer, toolbar

Example

The following example shows what two labels look like in memory, with the *itemprevious* option inserted in each:

```
<label sid="LABEL2">
   <value>This is a label.</value>
   <itemprevious>LABEL1</itemprevious>
</label>
<label sid="LABEL3">
   <value>This is a label.</value>
   <itemprevious>LABEL2</itemprevious>
</label>
```

You can use computes to determine the previous item on the form. For example, the following code shows a label that uses a compute to set its value. The compute goes to the item that currently has the focus, then copies the value of the *itemprevious* option from that item:

```
<label sid="nextItemName">
   <value compute="PAGE1.global.focuseditem->itemprevious"></value>
</label>
```

Usage Details

1. Defaults: **none**
2. If the *itemprevious* option is in the first item on the page (excluding the global item), it points to the last item on the page.
3. When the *itemprevious* option is used in items that appear in table rows:
   - Each item points to the previous item in the row.
   - The first item in a row points to the last item in the previous row.
   - The first item in the table points to the table.
4. The *itemprevious* option is not saved or transmitted with form descriptions.

justify

Aligns lines of text within the space an item occupies.

Syntax

```
<justify>alignment</justify>
```

alignment	*left*	align each line's left edge along the left margin
	right	align each line's right edge along the right margin
	center	align the center of each line with the center of the item

Available In

button, combobox, field, label, popup

Example

This sample aligns the text in the center of the item.

```
<justify>center</justify>
```

Usage Details

1. Default: varies depending on the item
   - *button* and *popup* items: **center**
   - *combobox, label* and *field* items: **left**
2. The built-in labels (*label* option) for items do not support a *justification* option.
3. If you center or right justify a field with a *scrollhoriz* of never, the field will wordwrap if you type beyond the edge of the field. However, the new line of text will continue to be right or center justified.

keypress

Contains the last keystroke made by the user in the focused item, page, or form. A keypress option is ignored if no keypress has been established at the level of focus. If the value of a keypress option is ignored at the item level, it passes up to the page level, and if ignored at the page level, it passes up to the form level. This option allows for the creation of a default button (shortcut key) on a page or a form.

This option is not saved or transmitted as part of the form. Instead, it is automatically created each time the form is read into memory, and is maintained only during display or processing.

Syntax

```
<keypress>key pressed</keypress>
```

Key pressed	*ESC*	escape key

ENTER	enter key
NUMPAD_ ENTER	number pad enter key
" "	space bar
F[1-12]	function key (can be any key from F1 to F12)
any keyboard char	any key on the keyboard

Available In

button, field, combobox, popup, list, radio, check, page global, form global

Example

The following example shows how you can set the keypress option to respond to the F7 key. When the form's user presses the F7 key, it activates the CancelButton's cancel action and cancels the form.

```
<button sid="CancelButton">
   <type>cancel</type>
   <value>Cancel</value>
   <custom:myoption compute="toggle(global.global.keypress, &#xA;
      '', 'F7') == '1' ? set ('CancelButton.activated','on') &#xA;
      : ''"></custom:myoption>
</button>
```

Usage Details

1. Default: **none**
2. The *keypress* option enables the use of a default button or action on the form.
3. You cannot place a compute in the activated option of a button. You must place it in a custom option.
4. The *keypress* option is not saved or transmitted with form descriptions.

label

Specifies a text label for an item. The label usually appears above the item and aligned with its left margin. For *popup* items, the label appears inside the item when no selection has been made.

Syntax

```
<label>label text</label>
```

label text	*string*	the text of the label

Available In

cell, check, checkgroup, combobox, field, list, pane, popup, radio, radiogroup, slider, spacer, page global, form global

Example

This sample defines a typical label.

```
<label>Student Registration Form</label>
```

Usage Details

1. Default:
   - XFDL: **none**
   - XForms: the value of the *xforms:label* option.
2. If an item contains both a *label* option and an *xforms:label* option, the XFDL *label* option takes precedence.
3. The label defined in a *label* option has a transparent background by default. To display a particular color behind the label, set the *labelbgcolor* option.
4. If used in the page global, the *label* option sets the title of the page as it appears in the title bar of the window displaying the form. If used in the form global, the *label* option sets the default title for all pages.
5. Multiple line labels require a carriage return in the code where you want it to appear in the label. For example:

   ```
   <label>This label spans
   two lines.</label>
   ```

   Note that not all items allows multi-line labels.

labelbgcolor

Defines the background color for the label specified in the label option.

Syntax

```
<labelbgcolor>color</labelbgcolor>
```

color	*special*	The color may be expressed in any of the following formats:
		• Comma-separated RGB values. For example:
		`192,192,192`
		• Hexadecimal-based RGB values. For example:
		`#336699`
		• Color name. For example:
		`blue`

Available In

check, checkgroup, combobox, field, list, pane, radio, radiogroup, slider

Examples

These samples all set the background color to red.

```
<labelbgcolor>red</labelbgcolor>
<labelbgcolor>255,0,0</labelbgcolor>
<labelbgcolor>#FF0000</labelbgcolor>
```

Usage Details

1. Default: **transparent**

   This means that a *label* option will always be transparent unless a color is specified (if no color is specified, the label background color will be the same as the page background color or an underlying box, if there is one).

labelborder

Defines whether there is a border around the label specified in the *label* option.

Syntax

```
<labelborder>status</labelborder>
```

border	*on*	item has a border
	off	item does not have a border

Available In

check, checkgroup, combobox, field, list, radio, radiogroup, slider

Example

This sample sets the the label to have a border:

```
<labelborder>on</labelborder>
```

Usage Details

1. Default: **off**
2. The border is always one pixel in width.

labelfontcolor

Defines the font color for the label specified in the *label* option.

Syntax

```
<labelfontcolor>color</labelfontcolor>
```

color	*special*	The color may be expressed in any of the following formats: • Comma-separated RGB values. For example: `192,192,192` • Hexadecimal-based RGB values. For example: `#336699` • Color name. For example: `blue`

Available In

check, checkgroup, combobox, field, list, pane, radio, radiogroup, slider

Examples

These samples both set the font color to green1:

```
<labelfontcolor>green</labelfontcolor>
<labelfontcolor>0,255,0</labelfontcolor>
<labelfontcolor>#008000</labelfontcolor>
```

Usage Details

1. Defaults first to page setting for *fontcolor*, then to the form setting for *fontcolor*. If there is no *fontcolor* setting at the page or form level, the default is black.

labelfontinfo

Defines the font name, point size, and font characteristics for the label specified in the label option.

Syntax

```
<labelfontinfo>
   <fontname>font name</fontname>
   <size>point size</size>
   <effect>effect1</effect>
   ...
   <effect>effectn</effect>
</labelfontinfo>
```

Note:

- effects are optional.

font name	*string*	the name of the font
point size	*unsigned byte*	the size of the font

effect	*string*	Can be any of the following: • **plain** — use plain face font. • **bold** — use bold face font. • **underline** — use underlined font. • **italic** — use italic font.

Available In

check, checkgroup, combobox, field, list, pane, radio, radiogroup, slider

Example

This sample sets the font information to Palatino 12, plain (the default), underlined in the ANSI character set.

```
<labelfontinfo>
   <fontname>Palatino</fontname>
   <size>12</size>
   <effect>underline</effect>
</labelfontinfo>
```

Usage Details

1. Defaults first to page setting for *fontinfo*, then to the form setting for *fontinfo*. If neither setting exists, then the defaults are:
   - font name: **Helvetica**
   - point size: **8**
   - weight: **plain**
   - effects: **not underlined**
   - form: **not italics**
2. If any of the font info settings are invalid, then the defaults are used.
3. The *size* option calculates item size using the font's average character width. Therefore, choice of font affects item width.
4. XFDL supports the following fonts and font sizes:
   - Fonts: Courier, Times, Symbol (symbol), Helvetica, and Palatino.
   - Sizes: 8, 9, 10, 11, 12, 14, 16, 18, 24, 36, 48.
5. Other fonts and font sizes may be used. However, especially for cross-platform Internet applications, it is best to choose from the ones cited above since they are guaranteed to work.

last

Identifies the last item in a repeat, group, or switch. This is the item that receives the focus when the user tabs backward into a group, a particular case, or a new row in a repeat.

This option affects the tab order in the following ways:

- When the user tabs backward into a table or pane, the focus goes to this item. In the case of a table, the focus goes to this item in the last row.
- When the user tabs backward from the beginning of a row, the focus goes to this item in the previous row or to the item that precedes the table or pane.

- When the user tabs forward from this item, the focus goes to the next row or to the item that follows the table or pane.

Syntax

```
<last>item reference</last>
```

item reference	*string*	an XFDL reference to the last item in the last row of a table or the last item in a pane.

Available In

pane, table

Example

The following example shows what a table might look like in memory, once the *first* option has been created. In this case, the *first* option contains a reference to the last LineTotal field in the table:

```
<table sid="itemsTable">
   <first>Product</first>
   <last>Qty</last>
   <xforms:repeat nodeset="order/row">
      <field sid="Qty">
         <previous>Product</previous>
         <xforms:input ref="qty">
            <xforms:label></xforms:label>
         </xforms:input>
      </field>
      <popup sid="Product">
         <next>Qty</next>
         <xforms:select1 appearance="minimal" ref="product">
            <xforms:label>Choose product</xforms:label>
            <xforms:item>
               <xforms:label>Widget</xforms:label>
               <xforms:value>widget</xforms:value>
            </xforms:item>
            <xforms:item>
               <xforms:label>Gadget</xforms:label>
               <xforms:value>gadget</xforms:value>
            </xforms:item>
         </xforms:select1>
      </popup>
      <label sid="LineTotal">
         <xforms:output ref="lineTotal"/>
      </label>
   </xforms:repeat>
</table>
```

In this case, if the user tabs backward into the table, the focus goes to the Qty field on the last row. When the user tabs to the previous row, the focus goes to the Qty field in that row.

Usage Details

1. Default: last item in the build order of a repeat, group, or case
2. When a page contains an xforms:switch, this option is not effective unless all cases contain an elmement with the same sid that is identified as *last*.

layoutinfo

This option records location information for all visible signed items. A hash is taken of each page containing a signed item, and this hash includes positioning information for all the signed items relative to each other in those pages.

Syntax

```
<layoutinfo>
   <pagehashes>
      <pagehash>
         <pageref>page sid_1</pageref>
         <hash>pagehash_1</hash>
      </pagehash>
      ...
      <pagehash>
         <pageref>page sid_n</pageref>
         <hash>pagehash_n</hash>
      </pagehash>
   </pagehashes>
</layoutinfo>
```

page sid	*string*	the scope ID of the page you are hashing.
pagehash	*string*	the hash of the page.

Available In

signature

Example

In the following example, the signature item includes hashes for the three pages on which items were signed:

```
<signature sid="signature1">
   <layoutinfo>
      <pagehashes>
         <pagehash>
            <pageref>PAGE1</pageref>
            <hash encoding="base64">pJAdX7+zh9+zEe</hash>
         </pagehash>
         <pagehash>
            <pageref>PAGE2</pageref>
            <hash encoding="base64">s+tHT+SElktqw4sod5</hash>
         </pagehash>
         <pagehash>
            <pageref>PAGE3</pageref>
            <hash encoding="base64">Jo13Ds+eth3EsSGE</hash>
         </pagehash>
      </pagehashes>
   </layoutinfo>
</signature>
```

Usage Details

1. This option helps ensure the security of your forms when you have signed items that are positioned using relative positioning, and unsigned *reference items*. Reference items are items on which the relative positioning of other items is based. Without layoutinfo, it is possible to move signed items on the form -

without breaking the signature - by moving their unsigned reference items. By using layoutinfo to record the position of the signed items relative to each other, you can detect this form of tampering.

linespacing

This option adjusts the spacing between lines of text. This sets on offset value, which will add to or subtract from the default spacing. For example, a value of 1 will add one pixel to the space between each line, while a value of -1 will remove one pixel from the space between each line.

Syntax

```
<linespacing>offset</linespacing>
```

offset	*integer*	the offset measured in pixels. This can be a positive or negative integer, or a value of 0 for no offset.

Available In

button, label, spacer

Example

In the following example, the linespacing for a label is set to 12, which will create a large amount of space between lines:

```
<label sid="LABEL1">
   <size>
      <width>10</width>
      <height>5</height>
   </size>
   <value>
   </value>
   <linespacing>12</linespacing>
</label>
```

Usage Details

1. Default: **0**.
2. When using a negative offset, the offset cannot be larger than the size of the font (that is, the second line of text cannot be moved higher than the first line). For example, if the font was 12 pixels high, the offset could not be larger than -12.
3. When using a positive offset, there is no limit to the size of the offset.
4. For buttons and labels, the linespacing affects the text in the *value* option. For spacers, the linespacing affects the text in the *label* option.

mimedata

Contains the actual data associated with a *data* item or a *signature* item. It can be binary data or the contents of an enclosed file. The data is encoded in base64 format, so that even forms containing binary data can be viewed in a text editor. When the data is needed by the form, it is decoded automatically from base64 back to its native format.

Data may also be compressed before base64 encoding, allowing an item to store a larger block of data.

About MIME data in signature items

The MIME data contains the contents of a signature. An XFDL generator must create it as follows:
1. Using the signature filter instructions in the associated signature button, create a plain-text version of the form or portion of the form to be signed.
2. Using the instructions in the signature button's *signformat* option, create a hash of the plain-text description.
3. sign the hash with the signer's private key.
4. Include a binary represenation of the signature (as generated by the signature engine) in the *mimedata* option.

Syntax

```
<mimedata encoding="format">MIME data</mimedata>
```

format	*string*	the format in which to encode the data. Valid formats are: • **base64** — a base 64 textual representation of the data. • **base64-gzip** — a base 64 textual represenation of the data after it has been gzip compressed.
MIME data	*string*	the binary data or enclosed file contents

Available In

data, signature

Example

This sample assigns some encoded data to the mimedata option:

```
<mimedata encoding="base64-gzip">
   R0lGODdhYABPAPAAAP///wAAACwAAAAAYABPAAAC/4SPqcvtD02Y
   Art68+Y7im7ku2KkzXnOzh9v7qNw+k+TbDoLFTvCSPzMrS2YzmTE
</mimedata>
```

This sample shows a mimedata option in a digital signature:

```
<signature sid="empsignature">
   <signformat>application/vnd.xfdl</signformat>
   <signer>Jane D Smith, jsmith@insurance.com</signer>
   <signature>Page1.empsignature</signature>
   <signitemrefs>
      <filter>omit</filter>
```

```
            <itemref>Page1.mgrSigButton</itemref>
            <itemref>Page1.admSigButton</itemref>
            <itemref>Page1.empsignature</itemref>
            <itemref>Page1.mgrsignature</itemref>
            <itemref>Page1.admsignature</itemref>
         </signitemrefs>
      <!-- The items listed above MUST have itemlocation
        options with absolute and extent as the last
        settings in order for the filter below to
        be sufficient in terms of security -->
         <signoptionrefs>
            <filter>keep</filter>
            <optionref>PAGE1.mgrSigButton.itemlocation</optionref>
            <optionref>PAGE1.admSigButton.itemlocation</optionref>
            <optionref>PAGE1.empsignature.itemlocation</optionref>
            <optionref>PAGE1.mgrsignature.itemlocation</optionref>
            <optionref>PAGE1.admsignature.itemlocation</optionref>
         </signoptionrefs>
         <signoptions>
            <filter>omit</filter>
            <optiontype>triggeritem</optiontype>
            <optiontype>coordinates</optiontype>
         </signoptions>
         <mimedata encoding="base64">
            MIIFMgYJKoZIhvcNACooIIFIzCCBR8CAQExDzANBgkgAQUFADA
            LB\ngkqhkiG9w0BBwGgggQZMCA36gAwSRiADjdhfHJl6hMrc5D
            ySSP+X5j\nANfBGSOI\n9w0BAQQwDwYDVQQHEwhJbmRlcm5lqw
            dDEXMBUGA1UEChM\nOVmVyaVNpZ24sIEluYy4xNDAKnqweaftn
            1ZlcmlTaWduIENsYXNzIDEgQ0Eg\nLSJbmRdWFsIFN1YnNjcml
            iyZXIwHhcNOTgwMTI3MwMDAwOTgwM\M1OTU5WjCCARExETA
         </mimedata>
      </signature>
```

Usage Details

1. Default: **none**
2. Base64 encoding transforms the data into a format that can be processed easily by text editors, e-mail applications, and so on. Converting data to base64 format ensures the resulting string contains no characters requiring an escape sequence.
3. Base64-gzip encoding compresses the data before transforming it to base64 format.
4. For signatures: because the signer's public key is included in the MIME data, a subsequent program can verify a signature without requiring that the signer's key be previously installed.
5. For signatures: the *mimedata* option in a *signature* item is always omitted from the signature represented by that item, regardless of the signature filters in use. This is done because the *mimedata* is not populated with the signature information until after the signature has been applied. (In other words, the signature can't include itself because it hasn't been generated yet.)
6. For signatures: the *mimedata* option in a data item used to store a signature image (see *signatureimage* option) is always omitted from the signature represented by that image, regardless of the signature filters in use. This is done because the *mimedata* is not populated with the signature image until after the signature has been applied. (In other words, the signature can't include its own image because it hasn't been added to the form yet.)

mimetype

Defines the MIME type of the data stored in a *data* item.

Syntax

```
<mimetype>MIME type</mimetype>
```

MIME type	*string*	the MIME type of the data item

Available In

data

Example

This sample sets the MIME type to indicate image data:

```
<mimetype>image/gif</mimetype>
```

Usage Details

1. Default: **application/vnd.xfdl**
2. The following are examples of MIME types. For full information on MIME types, read the MIME rfcs (1521, 1522, 1867 and 822), available on the World Wide Web.

   application/vnd.xfdl
   XFDL form item

   image/jpeg
   image item

   image/rast
   image item

   image/bmp
   image item

   image/gif
   image item

   text/plain
   ASCII text item

mouseover

Specifies whether the mouse pointer is currently over an item or page.

This option is set by an external program such as a parser, and is not saved or transmitted as part of the form. Instead, it is automatically created each time the form is read into memory, and is maintained only during display or processing.

Syntax

```
<mouseover>status</mouseover>
```

status	*on*	mouse pointer is over item or page

off	mouse pointer is not over item or page

Available In

button, check, checkgroup, combobox, field, list, popup, radio, radiogroup, toolbar, page global

Example

The following example shows a button that changes its color to white if it the mouse pointer is over it, and to blue if the pointer is not over it.

```
<button sid="saveButton">
   <type>saveas</type>
   <value>Save</value>
   <bgcolor compute="mouseover == 'on' ? 'white' :
      'blue'"></bgcolor>
</button>
```

Usage Details

1. Default: **off**
2. An object's *mouseover* option is set to on when the mouse pointer is over the object, and to off when the mouse pointer is not over the object.
3. A page global *mouseover* option is set to on when the mouse pointer is over the page (even if it is also over an item on the page).
4. A *mouseover* option in a toolbar is set to on when the mouse pointer is over the toolbar (even if it is also over an item in the toolbar).
5. The *mouseover* option is not included in form descriptions that are saved or transmitted.

next

Identifies the item to receive focus when a user tabs ahead from the current item. If a user tabs ahead from the last item on the page, the tab cycles within the same page, beginning with the first item on the page. Only modifiable or readonly items can receive focus.

Syntax

```
<next>item reference</next>
```

item reference	*string*	identifies the item to receive focus next

Available In

button, check, checkgroup, combobox, field, list, popup, radio, radiogroup, slider, page global

Example

This sample points to the item ″address_field″. When users tab ahead from the current item, the item identified as ″address_field″ will receive focus.

```
<next>address_field</next>
```

Usage Details

1. Default: **none**
2. The first page defined in the form is always the first page displayed. The default tabbing order depends on the order in which page and item definitions occur within the form definition. The sequence is as follows:
   - First item to receive focus: first modifiable item defined for the body of the first page
   - Subsequent items to receive focus: each modifiable item on the page in the order in which they are defined
3. If the last item on the page is tabbed past, the first modifiable item in the page's toolbar receives focus. If there is no toolbar, focus returns to the first item.
4. Placing next in page globals defines the first item to receive focus when the page appears.
5. If the *next* option identifies page globals, focus moves to the item defined to receive focus when the page appears. The page globals reference is *global* for the current page or *page_tag.global* for another page.

pagefirst

Stores a reference to the global item on the first page of the form, excluding the global page. A page is *first* when it appears first in the build order (in other words, it is first in the XFDL text).

This option is not saved or transmitted as part of the form. Instead, it is automatically created each time the form is read into memory, and is maintained only during display or processing.

Syntax

```
<pagefirst>page reference</pagefirst>
```

page reference	*string*	a reference to the first page in the form, excluding the global page.

Available In

page global

Example

This sample shows how the *pagefirst* option appears in the page globals of a form.

```
<page sid="PAGE1">
   <global sid="global">
      <pagefirst>PAGE1.global</pagefirst>
   </global>
</page>
```

Usage Details

1. Defaults: **none**
2. The *pagefirst* option stores a reference to the global item of the first page. For example, *PAGE1.global*.
3. The *pagefirst* option is not saved or transmitted with the form description.

pageid

Defines a unique identifier for the page.

Syntax

```
<pageid>
   <serialnumber>serial number</serialnumber>
</pageid>
```

serial number	*string*	provides a unique identifying string for the form. This is generated by an external program such as a form design program.

Available In

page global

Example

This sample shows how the *pageid* option appears in the page globals of a form.

```
<pageid>
   <serialnumber>{94EC2BA4-7D34-B5E4-0060-9947}</serialnumber>
</pageid>
```

Usage Details

1. Defaults: **none**
2. This option is intended for use with form design programs and licensign models.

pagelast

Stores a reference to the global item in the last page of the form, excluding the global page. A page is *last* when it appears last in the build order (in other words, it is last in the XFDL text).

This option is not saved or transmitted as part of the form. Instead, it is automatically created each time the form is read into memory, and is maintained only during display or processing.

Syntax

```
<pagelast>page reference</pagelast>
```

page reference	*string*	a reference to the last page in the form, excluding the global page.

Available In

page global

Example

This sample shows how the *pagelast* option appears in the page globals of a form.

```
<page sid="PAGE1">
   <global sid="global">
      <pagelast>PAGE4.global</pagelast>
   </global>
</page>
```

Usage Details

1. Defaults: **none**
2. The *pagelast* option stores a reference to the global item of the last page. For example, *PAGE4.global*.
3. The *pagelast* option is not saved or transmitted with form descriptions.

pagenext

Stores a reference to the global item in the next page in the form, excluding the global page. A page is *next* when it appears next in the build order (in other words, it is next in the XFDL text).

This option is not saved or transmitted as part of the form. Instead, it is automatically created each time the form is read into memory, and is maintained only during display or processing.

Syntax

```
<pagenext>page reference</pagenext>
```

page reference	*string*	a reference to the next page in the form, excluding the global page.

Available In

page global

Example

This example shows the *pagenext* option for PAGE2 of a form.

```
<page sid="PAGE2">
   <global sid="global.global">
      <pagenext>PAGE3.global</pagenext>
   </global>
</page>
```

Usage Details

1. Defaults: **none**
2. The *pagenext* option stores a reference to the global item of the next page. For example, PAGE1 might contain the following reference: *PAGE2.global*.
3. If the *pagenext* option is on the last page of the form, it points to the first page of the form (excluding the global page).
4. The *pagenext* option is not saved or transmitted with form descriptions.

pageprevious

Stores a reference to the global item in the previous page in the form, excluding the global page. A page is *previous* when it immediately precedes the current page in the build order (in other words, it is immediately previous in the XFDL text).

This option is not saved or transmitted as part of the form. Instead, it is automatically created each time the form is read into memory, and is maintained only during display or processing.

Syntax

```
<pageprevious>page reference</pageprevious>
```

page reference	*string*	a reference to the previous page in the form, excluding the global page.

Available In

page global

Example

This example shows the *pageprevious* option for PAGE2 of a form.

```
<page sid="PAGE2">
   <global sid="global.global">
      <pageprevious>PAGE1.global</pageprevious>
   </global>
</page>
```

Usage Details

1. Defaults: **none**
2. The *pageprevious* option stores a reference to the global item of the previous page. For example, PAGE2 might contain the following reference: *PAGE1.global*.
3. If the *pageprevious* option is on the first page of the form (excluding the global page), it points to the last page of the form.
4. The *pageprevious* option is not saved or transmitted with form descriptions.

previous

Identifies the item to receive focus when a user tabs backwards, using SHIFT + TAB, from the current item. If the current item has a *previous* option, the item indicated in that option is next in the reverse tab order. If the current item has no *previous* option, the previous item in the build order that can receive the input focus is next in the reverse tab order.

Syntax

```
<previous>item reference</previous>
```

item reference	*string*	identifies the item to receive focus next

Available In

button, check, checkgroup, combobox, field, list, pane, popup, radio, radiogroup, slider, table

Example

This sample points to the item ″date_field″. When users tab back from the current item, the item identified as ″date_field″ will receive focus.

```
<previous>date_field</previous>
```

Usage Details

1. The first page defined in the form is always the first page displayed. The default tabbing order depends on the order in which page and item definitions occur within the form definition. The sequence is as follows:
   - First item to receive focus: first modifiable item defined for the body of the first page
   - Subsequent items to receive focus: each modifiable item on the page in the reverse order in which they are defined
2. When tabbing back past the first item on the page, the last modifiable item in the page's toolbar receives focus. If there is no toolbar, focus returns to the last item defined in the page.

printbgcolor

Enables the form to be printed with a specific background color on a color printer. This color can be the same as or different from the background color shown on the screen. On black and white printers, grayscaling is used.

See also: *printlabelbgcolor*, *printfontcolor*, *printlabelfontcolor*.

Syntax

```
<printbgcolor>color</printbgcolor>
```

color	*special*	The color may be expressed in any of the following formats: • Comma-separated RGB values. For example: `192,192,192` • Hexadecimal-based RGB values. For example: `#336699` • Color name. For example: `blue`

Available In

box, button, check, checkgroup, combobox, field, label, list, popup, radio, radiogroup, slider, page global, form global

Examples

These samples both set the printed background color to forest green.

```
<printbgcolor>forest green</printbgcolor>
<printbgcolor>34,139,34</printbgcolor>
<printbgcolor>#228B22</printbgcolor>
```

Usage Details

1. Defaults behave as follows:
   - Form: **white**
   - Page: the form global setting or default (**white**)
   - Item: when no *printbgcolor* is specified, the *printbgcolor* default is derived from the item's specified or default *bgcolor* (which may be further derived from the page or form global *bgcolor* settings or defaults).

printfontcolor

Enables the item to be printed with a specific font color on a color printer. This color can be the same as or different from the font color shown on the screen. On black and white printers, grayscaling is used.

Syntax

```
<printfontcolor>color</printfontcolor>
```

color	*special*	The color may be expressed in any of the following formats: • Comma-separated RGB values. For example: `192,192,192` • Hexadecimal-based RGB values. For example: `#336699` • Color name. For example: `blue`

Available In

button, check, combobox, field, label, line, list, popup, radio, slider, page global, form global

Examples

These samples both set the printed font color to forest green.

```
<printfontcolor>forest green</printfontcolor>
<printfontcolor>34, 139, 34</printfontcolor>
<printfontcolor>#228B22</printfontcolor>
```

Usage Details

1. The default *printfontcolor* is derived from the *fontcolor* setting or default for the item, except in the case of checks or radios, which default to **black**. (The *fontcolor* default may be further derived from page or form settings or defaults.)
2. For *check* items, the *printfontcolor* option describes what color the check in the check box will be when the form is printed. Likewise, for *radio* items, the *printfontcolor* option describes what color the on dot will be when the form is printed.
3. For *line* items, the *printfontcolor* option describes what color a line will be when the form is printed.

printing

Indicates whether the form is currently printing. This value toggles from off to on just before printing. Any computes that rely on this option are updated before the form prints. This allows you to make computed changes to the form just before it is printed.

This option is not saved or transmitted as part of the form. Instead, it is automatically created each time the form is read into memory, and is maintained only during display or processing.

Syntax

```
<printing>setting</printing>
```

setting	*on*	the form is currently printing.
	off	the form is not printing.

Available In

form global

Example

This example shows a form that is currently printing:

```
<globalpage sid="global">
   <global sid="global">
      <printing>on</printing>
   </global>
</globalpage>
```

Usage Details

1. The *printing* option is not saved or transmitted with the form description.

printlabelbgcolor

Enables a item's built-in label to be printed with a specific background color on a color printer. This color can be the same as or different from the background color shown on the screen. On black and white printers, grayscaling is used.

Syntax

```
<printlabelbgcolor>color</printlabelbgcolor>
```

color *special* The color may be expressed in any of the following formats:

- Comma-separated RGB values. For example:
  `192,192,192`
- Hexadecimal-based RGB values. For example:
  `#336699`
- Color name. For example:
  `blue`

Available In

check, checkgroup, combobox, field, list, pane, radio, radiogroup, slider

Examples

These samples both set the printed label background color to forest green.

```
<printlabelbgcolor>forest green</printlabelbgcolor>
<printlabelbgcolor>34,139,34</printlabelbgcolor>
<printlabelbgcolor>#228B22</printlabelbgcolor>
```

Usage Details

1. Default: the *printlabelbgcolor* default is derived from the item's specified or default *labelbgcolor*.

printlabelfontcolor

Enables an item's built-in label to be printed with a specific font color on a color printer. This color can be the same as or different from the font color shown on the screen. On black and white printers, grayscaling is used.

Syntax

```
<printlabelfontcolor>color</printlabelfontcolor>
```

color	*special*	The color may be expressed in any of the following formats:
		• Comma-separated RGB values. For example:
		`192,192,192`
		• Hexadecimal-based RGB values. For example:
		`#336699`
		• Color name. For example:
		`blue`

Available In

check, checkgroup, combobox, field, list, pane, radio, radiogroup, slider

Examples

These samples both set the printed label font color to forest green.

```
<printlabelfontcolor>forest green</printlabelfontcolor>
<printlabelfontcolor>34,139,34</printlabelfontcolor>
<printlabelfontcolor>#228B22</printlabelfontcolor>
```

Usage Details

1. Default: the *printlabelfontcolor* default is derived from the item's specified or default *labelfontcolor*.

printsettings

Determines the settings that will be used when the form is printed. The user can be allowed to change these defaults, or the form can be set so that it will always follow the defaults.

Syntax

```
<printsettings>
   <pages>page list</pages>
   <dialog>dialog settings</dialog>
   <header>header settings</header>
   <footer>footer settings</footer>
   <border>on|off</border>
   <singlelinefieldsaslines>on|off</singlelinefieldsaslines>
   <scroll barsonfields>on|off</scroll barsonfields>
   <radioswithoutvalues>on|off</radioswithoutvalues>
   <radiosaschecks>on|off</radiosaschecks>
   <pagelayout>layout setting</pagelayout>
</printsettings>
```

Note:

- All settings are optional.

page list	*(see below)*	the list of pages that print
dialog settings	*(see below)*	determines whether the print dialog is shown, and which settings must be used when printing (for example, paper orientation and number of copies)
header/footer settings	*(see below)*	
border	*on*	prints a border around form pages.
	off	does not print a border around form pages.
signlinefieldsaslines	*on*	print a single line field as a line.
	off	prints a single line field as a rectangle.
scroll barsonfields	*on*	prints scroll bars on fields.
	off	removes scroll bars from fields when printing.
radioswithoutvalues	*on*	prints radio buttons without the selected value.
	off	prints radio buttons with the selected value.
radiosaschecks	*on*	prints radio buttons as check boxes.
	off	prints radio buttons as radio buttons.

layout setting	*string*	controls whether the print job is scaled to fit a page, or tiled across multiple pages. Valid settings are: • **fittopage** — scales the form to fit on a single page, without maintaining the aspect ratio. • **shrinktopage** — scales the form to fit on a single page, and maintains the aspect ratio of the form. • **tileonedirection** — sizes the smallest dimension (either width or length) to fit to a single page, and tiles the other dimension across multiple pages. • **tiletwodirections** — does not resize the form, and tiles both the width and the height across multiple pages if necessary.

Available In

action, button, cell, page global, form global

Page List

The page list uses the following syntax:

```
<pages>
   <filter>keep|omit</filter>
   <pageref>page sid1</pageref>
   ...
   <pageref>page sidn</pageref>
</pages>
```

The settings for the page list work as follows:

Setting	Description
filter	Whether to omit or keep the pages listed. Using keep will print all pages listed, and exclude all other pages in the form. Using omit will print all pages in the form except those listed.
pageref	a list of page sids that indicates which pages to keep or omit.

Dialog Settings

The dialog settings use the following syntax:

```
<dialog>
   <active>on/off</active>
   <copies>#</copies>
   <orientation>portrait|landscape</orientation>
   <printpages>print page settings</printpages>
</dialog>
```

The settings work as follows:

Setting	Description
active	when on, the print dialog will be displayed before the form is printed, allowing the user to change the settings. When off, the dialog will not be shown and the form will be printed immediately.
copies	a positive integer that determines the number of copies that will be printed; defaults to 1.
orientation	determines whether the form will be printed in landscape or portrait orientation.
printpage settings	(see below)

Print Page Settings

The print page settings use the following syntax:

```
<printpages>
   <active>on|off</active>
   <choice>all|current</choice>
</printpages>
```

The settings work as follows:

Setting	Description
active	when on, the print dialog will allow the user to choose between printing ″All″ pages or the ″Current Page″; defaults to on.
choice	all sets the printpages default to ″All″; current sets the printpages default to ″Current Page″; defaults to all.

Header/Footer Settings

The header and footer settings use the following syntax:

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

The settings work as follows:

Setting	Description
left text	this text is left justified in the header/footer.
center text	this text is centered in the header/footer.
right text	this text is right justified in the header/footer.

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 section (that is, left, center, or right) can contain different text. For example, you might put a date in the left section, a title in the middle section, and a document 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.

Example

This sample prevents ″page2″ from being printed, sets the form to print in landscape orientation, strips the scollbars from all fields, and prints two copies of the form:

```
<printsettings>
   <pages>
      <filter>omit</filter>
      <pageref>page2</pageref>
   </pages>
   <dialog>
      <active>on</active>
      <orientation>landscape</orientation>
      <copies>2</copies>
   </dialog>
   <scroll barsonfields>off</scroll barsonfields>
</printsettings>
```

Usage Details

1. Defaults:
   - **Page List —** the page list will default to keeping **all** pages in the form.
   - **Dialog Settings —** the dialog will default to being **on**, and will print one copy of all pages in the form in a portrait orientation. By default, the user will be able to change all of these settings.
   - **border — off**.
   - **singlelinefieldsaslines, scroll barsonfields, radioswithoutvalues, radiosaschecks, pagelayout —** the equivalent setting in the form rendering software.

2. Those settings that inherit their default value from the form rendering software will override the form rendering software if they are set differently.
3. Print settings set on a specific page override the global printsettings. For example, if you set the number of copies globally, then set the **orientation** for page two, page two will not inherit the copies setting. If you want to add page specific settings to your form, you must repeat the form's global settings in the settings for that page.

printvisible

Determines whether an item is visible when the form is printed. To set whether the item is visible in the Viewer, use the *visible* option.

Syntax

```
<printvisible>setting</printvisible>
```

setting	*on*	the item is visible when the form is printed.
	off	the item is not visible when the form is printed.

Available In

box, button, check, checkgroup, combobox, field, label, line, list, popup, radio, radiogroup, slider

Examples

In this example, the *nameField* does not print due to the *printvisible* setting, but the *addressField* does because it defaults to the *visible* setting, which defaults to on.

```
<field sid="nameField">
   <value>John Doe</value>
   <printvisible>off</printvisible>
</field>
<field sid="addressField">
   <value>123 Home Street</value>
</field>
```

Usage Details

1. Default: defaults to the *visible* option for the item.
2. This option overrides the *visible* option for the purposes of printing.

readonly

Sets the item to be readonly, so that user's can read information in the item but cannot change that information.

Syntax

```
<readonly>setting</readonly>
```

setting	*on*	the item is readonly.
	off	the item accepts input.

Available In

check, checkgroup, combobox, field, list, popup, radio, radiogroup, slider

Example

The following example shows a field that is set to be readonly:

```
<field sid="readOnlyField">
   <value>You cannot type into this field.</value>
   <readonly>on</readonly>
</field>
```

Usage Details

1. Default:
   - XFDL: **off**.
   - XForms: defaults to the *readonly* property for the data element to which the containing item is bound.
2. If an item has either the *readonly* option or the XForms *readonly* property set, then the item is readonly. For more information about the readonly property, refer to “Property Settings” on page 213.
3. The *readonly* setting permits users to scroll an item even though they cannot update the item’s contents.

requirements

Specifies one or more requirements that must be satisfied before the form will function properly. For example, a form may require a Java™ Virtual Machine to run correctly.

You can use the requirements feature to check for the availability of a particular class, function call, or Java Virtual Machine. If the requirement is not met, you can display a message and then either continue or fail.

Syntax

```
<requirements>
   <requirement>requirement settings1</requirement>
      ...
   <requirement>requirement settingsn</requirement>
   <detected>status</detected>
</requirements>
```

requirement settings	*(see below)*	
status	*on*	all requirements were detected successfully. This is a controlled option that is not written out with the form.
	off	one or more requirements were not detected successfully. This is a controlled option that is not written out with the form. This is the default value.

Available In

form global

Requirement Settings

The requirement settings use the following syntax:

```
<requirement>
   <component>component setting</component>
   <detected>on|off</detected>
   <actions>
      actions
   </actions>
</requirement>
```

The settings work as follows:

Setting	**Description**
component	identifies the type of requirement. This can be a class, a function call, or a java virtual machine (*see below*).
detected	When on, the component for this requirement was successfully detected. When off, the component for this requirement was not successfully detected. The default is off. This is a controlled option that is not written out with the form.
actions	*optional. Specifies the action to perform if the component is not available. (see below).*
	optional. Contains a message that is displayed when the component is not successfully detected. If this option is not set, then no message is displayed to the user.

Class Component

When you list a class as a requirement, that class must be available for the requirement to be fulfilled. The class component uses the following syntax:

```
<component>
   <class>
      <name>class name</name>
      <minversion>version number</minversion>
   </class>
</component>
```

The settings work as follows:

Setting	Description
class name	the name of the class. For example: `com.PureEdge.WebServices`
version number	*optional*. The minimum version of the instance that must be registered. For example, 1.0.0.

Class Instance Component

When you list a class instance as a requirement, that class instance must be available for the requirement to be fulfilled. The class instance component uses the following syntax:

```
<component>
   <class>
      <name>class instance name</name>
      <minversion>version number</minversion
      <criteria>
         <value>value1</value>
         ...
         <value>valuen</value>
      </criteria>
   </class>
</component>
```

The settings work as follows:

Setting	Description
class instance name	the name of the class instance. For example: `com.PureEdge.WebServices`
version number	*optional*. The minimum version of the instance that must be registered. For example, 1.0.0.
value	a string that was used to register the class. For example, the Sun JVM (com.uwi.java.JavaInvocationEngineFactory class) is registered with the following string: ″Sun VM 1.4″ You may include any number of strings, and all strings must match for the requirement to be met.

Function Call Component

When you list a function call as a requirement, that function call must exist as an XFDL function for the requirement to be fulfilled. The function call can be made available through an extension or a WSDL. The function call component uses the following syntax:

```
<component>
   <functioncall>
      <name>name</name>
   </functioncall>
</component>
```

The settings work as follows:

Setting	**Description**
name	the name of the function call. This is the name of the function that must be registered and it must include the package name.

Java VM Component

When you list a Java Virtual Machine as a requirement, the correct version of the Java VM must be available on the local computer. The Java VM component uses the following syntax:

```
<component>
   <javavm>
      <minversion>minimum version</minversion>
      <maxversion>maximum version</maxversion>
   </javavm>
</component>
```

The settings work as follows:

Setting	**Description**
minimum version	*optional*. The minimum acceptable version of the Java VM. For example, 1.2. You must list either a minumum version or a maximum version. You may also list both.
maximum version	*optional*. The maximum acceptable version of the Java VM. For example, 1.4. You must list either a minumum version or a maximum version. You may also list both.

Example

The following example sets up the following requirements for a form: (1) a Java Virtual Machine version 1.4.0 or greater must be available, and (2) the Web Services class must be available.

```
<requirements>
   <requirement>
      <component>
         <javavm>
            <minversion>1.4.0</minversion>
         </javavm>
      </component>
      <actions>
         <message>Your computer does not have the required Java
            Virtual Machine installed. Because of this, the Web
            Services functions will not execute properly. To
            correct this, you must install the Sun Java VM version
            1.4 or higher.</message>
      </actions>
   </requirement>
   <requirement>
      <component>
         <class>
            <name>com.PureEdge.WebServices</name>
            <minversion>1.0.0</minversion>
         </class>
      </component>
      <actions>
         <message>The Web Services system is not available on your
            computer. Please contact your System Administrator for
            assistance.</message>
      </actions>
   </requirement>
</requirements>
```

Usage Details

1. The *detected* option is not written out with the form description that is saved or transmitted.
2. The default for the *detected* options is **off**.

rtf

Stores the rich text value for rich text fields.

Syntax

```
<rtf>rich text</rtf>
```

rich text	*string*	the rich text string.

Available In

field

Example

This sample shows a rich text field:

```
<field sid="richTextField">
   <texttype>text/rtf</texttype>
   <value>Hello</value>
   <rtf>rich text version of Hello</rtf>
</field>
```

Usage Details

1. Default: if the *rtf* option is empty or does not exist, a default rich text string is created as follows:
   - Text: set to equal the *value* option.
   - Font: set as the *fontinfo* and *fontcolor* options.
   - Justification: set as the *justify* option.

   If the required options are not present, then the default settings for those options are used. Note that the background color is controlled by the *bgcolor* option, not the rich text.
2. To use rich text in a field, the *texttype* option must be set to *text/rtf*.
3. A plain text version of the rich text string is stored in the *value* option, with the following considerations:
   - If the *value* option has a compute, that compute is lost when the text is copied from the *rtf* option. (However, external changes to the *value* are still made. For example, a *set*.)
   - Once the text is copied to the *value* option, the format system may impose a change based on the *format* option. If a change occurs, then the *rtf* option is reset based on the formatted *value* option and the *fontinfo*, *fontcolor*, *justify*, and *bgcolor* options.
   - If the plain text version does not match the rich text, then the rich text takes precedence. In this case, the *value* option is updated to reflect the rich text.
4. Computed changes to the *fontinfo*, *fontcolor*, *bgcolor*, *justify*, *texttype*, or *value* options will cause the *rtf* option to update if necessary.
5. Changes to the *rtf* option do not cause changes to the *fontinfo*, *fontcolor*, *bgcolor*, or *justify* options.
6. Dynamically changing the *texttype* from rtf to plain text is not supported.
7. When using XForms, the *rtf* option is bound to the data model (as opposed to the *value* option).

saveformat

Specifies the format a form will be saved in. An XFDL form may be saved in XFDL format or HTML format. Furthermore, the XFDL format may be compressed using ASCII compression.

The formats work as follows:
- XFDL format saves the entire form definition, including the user input.
- HTML format saves the form as a series of assignment statements for each modifiable item, equating the item reference with the item's value. The only items included in the save are custom items and the following modifiable items: *check*, *field*, *list*, *popup*, *combobox* and *radio*.

Syntax

```
<saveformat>MIME type</saveformat>
```

MIME type	*application/vnd.xfdl*	use XFDL format
	application/vnd.xfdl;content-encoding="base64-gzip"	use compressed XFDL format

application/x-www-form-urlencoded	use HTML format

Available In

action, button, cell, page global, form global

Examples

This example shows how to use saveformat in a save button:

```
<button sid="save_button">
   <type>saveas</type>
   <saveformat>application/x-www-form-urlencoded</saveformat>
</button>
```

When a user clicks this button, the form will be converted to HTML format (see Usage Note 3 below) and saved to the user's drive.

XFDL format in form globals

This example shows how to use saveformat as a form global characteristic.

```
<?xml version="1.0"?>
<XFDL xmlns="http://www.ibm.com/xmlns/prod/XFDL/7.0"
   xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
   xmlns:xforms="http://www.w3.org/2002/xforms">
   <globalpage sid="global">
      <global sid="global">
         <bgcolor>ivory</bgcolor>
         <saveformat>application/vnd.xfdl</saveformat>
      </global>
   </globalpage>
   <page sid="page_1">
      ...
```

Any time a user saves this form, it will be saved in XFDL format.

Usage Details

1. Default: The default format is the format that the form was in before it was parsed. For example, a form written in XFDL will be transmitted in XFDL, unless otherwise specified by this option.
2. This option can also be included as a form global option and in the definitions of items that trigger save actions. These are button or cell items that have a *type* option setting of **save**.

HTML Format by Item Type

The general syntax of a form saved in HTML format is:

```
itemreference=value&item reference=value&...
```

Note:

- The ampersand separates form items.

The syntax of items saved in HTML format by type:

Item Type	HTML Format
check	item sid=value option setting
field	item sid=value option setting
list	item sid=value option setting of selected cell Note that the item reference identifies the list.
popup	item sid= value option setting of selected cell Note that the item reference identifies the popup.
combobox	item sid= value option setting
radio	group option setting=item sid of selected radio
custom	item sid=value option setting

Substitutions and Omissions:

- Only modifiable items are saved as HTML data. A form cannot be saved in HTML format and expected to be viewed as a form again. It is saved as a string of item tags and their associated values.
- Spaces in the value are replaced by the plus sign (+). For example, 'Two words' becomes 'Two+words'.
- The membership operator in item and group references is replaced by a minus sign.
- *page_one.age_group* becomes *page_one-age_group*.
- Page tags are removed from item and group references in single page forms.
- Check boxes and radio buttons with a value option setting of **off** are omitted.
- Entries resulting in an empty string on the right hand side of the assignment statement are omitted. This occurs when the referenced option setting is empty or the option definition is missing.

scrollhoriz

Defines horizontal scrolling options for a *field* item.

Syntax

```
<scrollhoriz>option</scrollhoriz>
```

option	*never*	permit scrolling using the cursor but display no horizontal scroll bar
	always	permit scrolling and display a horizontal scroll bar
	wordwrap	wrap field contents from line to line, inhibit scrolling and display no horizontal scroll bar

Available In

field

Example

This sample sets the horizontal scrolling option to permit scrolling and to display the horizontal scroll bar.

```
<scrollhoriz>always</scrollhoriz>
```

Usage Details

1. Default: **never**
2. The scroll bar displays along the field's bottom edge.

scrollvert

Defines vertical scrolling options for a *field* item.

Syntax

```
<scrollvert>option</scrollvert>
```

option	*never*	permit scrolling using the cursor but display no vertical scroll bar
	always	permit scrolling and display a vertical scroll bar
	fixed	inhibit scrolling and display no vertical scroll bars

Available In

field

Example

This sample sets the vertical scrolling option to inhibit all scrolling.

```
<scrollvert>fixed</scrollvert>
```

Usage Details

1. Default: **never**
2. The scroll bar displays along the field's right edge.

signature

Used in conjunction with the button item to establish the XFDL item name by which a particular signature will be identified.

Syntax

```
<signature>name of signature</signature>
```

name of signature	*string*	the name of the signature

Available In

button, signature

Example

This sample identifies the signature item for a particular button as ″mysig″.

```
<signature>mysig</signature>
```

Usage Details

1. Default: **none**
2. The signature option must be included in each signature button that is set up.

signatureimage

Points to a data item, identifying it as the data item into which the captured signature image is placed. Used only with image-based digital signatures (such as CIC InkTools).

Syntax

```
<signatureimage>data item</signatureimage>
```

data item	*string*	the itemref for the data item that contains an image associated with a signature

Available In

button

Example

This sample identifies the data item ″SIGIMAGE″. When the user signs a form with a pen/pad device, the mimedata of the captured image is placed in the data item ″SIGIMAGE″.

```
<signatureimage>SIGIMAGE</signatureimage>
```

Usage Details

1. Default: **none**
2. This option is used with signature types that utilize a digital image, such as InkTools signatures.

signdatagroups

Specifies which datagroups are filtered for a particular signature. Filtering a datagroup means keeping or omitting all of the *data* items that are in the specified datagroup.

For example, if a *signdatagroups* option specifies that the ″attachments″ datagroup should be kept, then all *data* items within the ″attachments″ group will be signed.

This filter applies to all *data* items present at the time of signing, including those added as enclosures.

Syntax

```
<signdatagroups>
   <filter>datagroup filter</filter>
   <datagroupref>datagroup reference1</datagroupref>
      ...
   <datagroupref>datagroup referencen</datagroupref>
</signdatagroups>
```

Note:

- There may be any number of *datagroup reference* entries.

datagroup filter	*keep*	include datagroups in the *datagroup reference* list with the signature; omit those not in the list
	omit	omit datagroups in the *datagroup reference* list from the signature; include those not in the list
datagroup reference	*string*	identifies a datagroup whose *data* items will be filtered

Available In

button, signature

Example

This example specifies a *signdatagroups* option that keeps the datagroup called "Business_Letters".

```
<signdatagoups>
   <filter>keep</filter>
   <datagroupref>Business_Letters</datagroupref>
</signdatagroups>
```

Usage Details

1. Default: **omit nothing (keep all data items), unless the containing page is omitted**
2. Since enclosed files can belong to several datagroups, and datagroups can contain several enclosed files, care must be exercised when setting up *signdatagroups* options to ensure that only the desired datagroups are filtered.
3. Other filters may take precedence over the *signdatagroups* option. Refer to "Order of Precedence of Filters" on page 395 for more information on the order of precedence of filters.

signdetails

Specifies which certificate attributes are shown to the user when they are choosign a certificate to sign the form, and defines the filters used to select the available certificates when the user is signing a form.

For example, the *signdetails* option could specify that only those certificates with a common name that begins with "Bob" are shown, and that only the owner's common name and e-mail address are shown.

Syntax

```
<signdetails>
   dialogcolumns
   filteridentity
</signdetails>
```

Note:

- Both *dialogcolumns* and *filteridentity* are optional.

Expression	Setting	Description
dialogcolumns	(see below)	a list of certificate attributes that should be shown to the user when they are selecting a certificate to sign
filteridentity	(see below)	a list of certificate attributes and values that are used to filter which certificates are available to the user for signing

dialogcolumns

The *dialogcolumns* element uses the following syntax:

```
<dialogcolumns>
   <property>attribute1</property>
      ...
   <property>attributen</property>
</dialogcolumns>
```

Note:

- The number of attributes is optional.

Each certificate attribute listed is shown to the user when they view the certificates available for signing. For example, if you wanted the user to see the owner's common name and e-mail address for each certificate, you would use the following setting:

```
<dialogcolumns>
   <property>Subject: CN</property>
   <property>Subject: E</property>
</dialogcolumns>
```

For a list of available attributes, see "Certificate Attributes" below.

filteridentity

The *filteridentity* element uses the following syntax:

```
<filteridentity>
     <filter>
        <tag>attribute_1</tag>
        <value>value_1</value>
     </filter>
     ...
     <filter>
        <tag>attribute_n</tag>
        <value>value_n</value>
     </filter>
  </filteridentity>
```

Note:

- The number of attributes and filters is optional.

attribute	*string*	the name of the attribute you want to user to filter the available certificates
value	*string*	the value to which you want to compare the attribute. Use an asterisk (*) as a wildcard or multiple characters, or a question mark (?) as a wildcard for a single character.

If the value of the attribute matches the filter, then the certificate will be available to the user. For example, to restrict the available certificates to those with a common name beginning with ″Bob″, you would use the following filter:

```
<filteridentity>
   <filter>
      <tag>Subject: CN</tag>
      <value>Bob*</value>
   </filter>
</filteridentity>
```

For a list of available attributes, see ″Certificate Attributes″ below.

Available In

button, signature

Example

This example specifies a *signdetails* option that makes those certificates with an e-mail address in the ibm domain available, and shows the serial number and the owner’s common name for each certificate.

```
<signdetails>
   <dialogcolumns>
      <property>Serial</property>
      <property>Subject: CN</property>
   </dialogcolumns>
   <filteridentity>
      <filter>
         <tag>Subject: E</ae>
         <value>*@ibm.com</value>
      </filter>
   </filteridentity>
</signdetails>
```

Usage Details

1. Default: all certificates are available, and the certificate's common name and expiry date are shown to the user.

Certificate Attributes

The following is a list of attributes that are common to X.509 certificates. Note that the names of certificate attributes are case sensitive.

Attribute	Description
Version	the version of the X.509 specification that the certificate follows
Serial	the certificate's serial number
signatureAlg	the algorithm used by the Certificate Authority to sign the certificate
BeginDate	the date at which the certificate became valid
EndDate	the certificate's expiry date
PublicKey	the certificate's public key
FriendlyName	the certificate's friendly name
Subject: CN	the certificate owner's common name
Subject: E	the certificate owner's e-mail address
Subject: T	the certificate owner's title
Subject: L	the certificate owner's locality
Subject: ST	the certificate owner's state of residence
Subject: O	the organization to which the certificate owner belongs
Subject: OU	the name of the organizational unit to which the certificate owner belongs
Subject: C	the certificate owner's country of residence
Subject: STREET	the certificate owner's street address
Subject: ALL	the certificate owner's complete distinguished name
Issuer: CN	the certificate issuer's common name
Issuer: E	the certificate issuer's e-mail address
Issuer: T	the certificate issuer's title
Issuer: L	the certificate issuer's locality
Issuer: ST	the certificate issuer's state of residence
Issuer: O	the organization to which the certificate issuer belongs
Issuer: OU	the organizational unit to which the certificate issuer belongs
Issuer: C	the certificate issuer's country of residence
Issuer: STREET	the certificate issuer's street address
Issuer: ALL	the certificate issuer's complete distinguished name

signer

Identifies who signed a particular form.

Syntax

```
<signer>identity of user</signer>
```

Identity of user	*string*	identity of user

Available In

button, signature

Example

In this example, signer is similar to a user's e-mail signature, clearly identifying who signed the form.

```
<signer>John Smith, jsmith@acme.org</signer>
```

Usage Details

1. The setting of the *signer* option varies, depending on the signing engine used:

signing Engine	**signer Setting**
Generic RSA	common name, e-mail
CryptoAPI	common name, e-mail
Netscape	common name, e-mail
Entrust	signer's login identity
CIC	signer's name as entered during signing ceremony
Clickwrap	*Accepted*
HMAC Clickwrap	the value of the answer indicated by the HMACsigner tag in the *signformat* option Note that if the HMACsigner tag includes more than one answer, they are combined in a comma delimited list. For example, ″answer1, answer2″.

2. The *signer* option is automatically generated by the signature button when the user signs the form. It is added to both the signature button code and the signature code. No manual coding is required.

signformat

Records the type of encoding that a form viewing program must use to create the mimedata setting in a signature. Specifically, the parameters in signformat specify:

- The MIME type of the data from which the mimedata setting is created (see below for an explanation).
- The signature engine to use.
- Settings specific to the engine used.

About the mimedata setting:

To create the mimedata setting, a form viewer takes the signer's certificate and a plaintext representation of the form or portion of the form that the signature applies to, and encodes them according to the settings in signformat. For details, see the *mimedata* option.

Syntax

```
<signformat>MIME type;
   engine="signature engine";
   verifier;
   cval;
   delete;
   parameters
</signformat>
```

MIMEtype	*string*	the MIME type of the signed data. May be: • XFDL — application/vnd.xfdl
signature engine	*string*	the type of signature. Valid types are: • ClickWrap • CryptoAPI • Entrust • Generic RSA (includes CryptoAPI and Netscape) • HMAC-ClickWrap • Netscape • signaturePad • Silanis Default: Generic RSA
verifier	*string*	an optional flag that indicates which verifier should be used when verifying certificate chains during digital signature operations. Valid verifiers are: • Basic — Performs basic certificate verification. • DODJ12 — Performs strict certificate verification that complies with US Department of Defense requirements. Default: Basic
cval	*string*	an optional flag that indicates whether the current value of computed options is signed. This is useful if you want to sign the compute, but not the value calculated by the compute (for example, if you are signing the presentation layer of a form). If you want to sign the current values, do not use this flag. If you do not want to sign the current values, use: `cval="off"` Default: current values are signed.
delete	*string*	an optional flag that indicates whether the signature can be deleted by the user. By default, all signatures can be deleted. If you want to prevent a signature from being deleted, use: `delete="off"`

parameters	*depends on engine*	additional parameters required by the signature engine (see below)

Available In

button, signature

About the signature Engines

The following table describes the signature engines that are available:

signature Engine	Description
ClickWrap	The ClickWrap signature allows users to sign a form without requiring a digital certificate. Instead, the signer may have to answer some questions about themself and echo some text to indicate their intent to agree to the document.
CryptoAPI	The CryptoAPI signature uses digital certificates that are stored in your Internet Explorer certificate store to create a digital signature.
Entrust	The Entrust signature allows the end-user to sign the form using the Entrust brand of products.
Generic RSA	The Generic RSA signature uses digital certificates from either your Internet Explorer or your Netscape certificate store.
HMAC-ClickWrap	The HMAC-ClickWrap signature is similar to the ClickWrap signature, in that the end-user does not require a digital certificate to create a signature. Instead, the end-user provides a *shared secret* (such as a password) that can be verified by a server. Once the signature is verified, the server then uses its own digital certificate to sign the form and validate the end-user's non-digital signature.
Netscape	The Netscape signature uses digital certificates that are stored in your Netscape certificate store to create a digital signature.
signaturePad	The signaturePad signature allows the end-user to sign the form using a variety of pad-style hardware. This signature type includes support for signature pads from Interlink and Topaz, as well as pads that support the WinTab standard.
Silanis	A Silanis signature allows the end-user to sign the form using the Silanis brand of products. It is also the only signature type that can create non-locking signatures, in which the data that is signed is not locked to prevent changes.

Additional Parameters for Common signature Engines

The following table details the additional parameters you must use for each signature type when defining the *signformat* option. Note that the Generic RSA signature does not require any additional parameters.

Engine	Parameter	Valid Settings	Description
Clickwrap	hashalg	sha1	hash algorithm with 160-bit message digest. This is the default setting.
		md5	hash algorithm with 128-bit message digest

Engine	Parameter	Valid Settings	Description
	titleText	string	text to display for the main title of the signature dialog box
	mainPrompt	string	text to display for the main prompt
	mainText	string	text to display for the main text
	question1Text	string	label for the first question
	answer1Text	string	default answer to the first question; user can overwrite
	question2Text	string	label for the second question
	answer2Text	string	default answer to the second question; user can overwrite
	question3Text	string	label for the third question
	answer3Text	string	default answer to the third question; user can overwrite
	question4Text	string	label for the forth question
	answer4Text	string	default answer to the forth question; user can overwrite
	question5Text	string	label for the fifth question
	answer5Text	string	default answer to the fifth question; user can overwrite
	echoPrompt	string	text to display for the echo prompt
	echoText	string	text to display for the signer to echo
	buttonPrompt	string	text to display above the accept and reject buttons
	acceptText	string	text to display on the accept button
	rejectText	string	text to display on the reject button
	HMACsigner	string	a comma delimited list of the the answers that store the signer's identity. This is written as *answer*$_n$. For example, answer$_1$, answer$_2$, and so on. Note that this parameter applies only to HMAC-Clickwrap signatures
	HMACSecret	string	a comma delimited list of the the answers that store the shared secret. This is written as *answer*$_n$. For example, answer$_1$, answer$_2$, and so on. Note that this parameter applies only to HMAC-Clickwrap signatures
	readonly	string	a comma delimited list of the the answers that should be read-only. This is written as *answer*$_n$. For example, answer$_1$, answer$_2$, and so on. Note that this parameter applies only to HMAC-Clickwrap signatures
CryptoAPI	csp	determined by form viewing program	Cryptographic Service Provider (ie Microsoft® Base Cryptographic Provider v1.0)

Engine	Parameter	Valid Settings	Description
	csptype	rsa_full	full RSA implementation (this is the default)
		rsa_sig	for a CSP that supplies only RSA signature algorithms
		dss	for a CSP that supplies algorithms compliant with the Digital signature Standard
		dss_dh	for a CSP that supplies DSS compliant algorithms and Diffie-Hellman encryption
		fortezza	for a CSP that supplies Fortezza algorithms
	hashalg	sha1	hash algorithm with 160-bit message digest. This is the default setting.
		md5	hash algorithm with 128-bit message digest
Entrust	hashalg	sha1	hash algorithm with 160-bit message digest. This is the default setting.
		md5	hash algorithm with 128-bit message digest
Netscape	hashalg	sha1	hash algorithm with 160-bit message digest. This is the default setting.
		md5	hash algorithm with 128-bit message digest
signaturePad	TSP	string	defines the preferred signature pad software/hardware to use. Valid settings are: • Interlink • Topaz • WinTab If no setting is specified, the form viewing software will determine which hardware is available, and will default to either Interlink or Topaz first and WinTab second.
	hashalg	sha1	hash algorithm with 160-bit message digest. This is the default setting.
		md5	hash algorithm with 128-bit message digest
	titleText	string	text to display for the main title of the signature dialog box
	mainPrompt	string	text to display for the main prompt
	mainText	string	text to display for the main text
	question1Text	string	label for the first question

Engine	Parameter	Valid Settings	Description
	answer1Text	string	default answer to the first question; user can overwrite
	question2Text	string	label for the second question
	answer2Text	string	default answer to the second question; user can overwrite
	question3Text	string	label for the third question
	answer3Text	string	default answer to the third question; user can overwrite
	question4Text	string	label for the forth question
	answer4Text	string	default answer to the forth question; user can overwrite
	question5Text	string	label for the fifth question
	answer5Text	string	default answer to the fifth question; user can overwrite
	echoPrompt	string	text to display for the echo prompt
	echoText	string	text to display for the signer to echo
	buttonPrompt	string	text to display above the accept and reject buttons
	acceptText	string	text to display on the accept button
	rejectText	string	text to display on the reject button
	readonly	string	a comma delimited list of the the answers that should be read-only. This is written as *answer*$_n$. For example, answer$_1$, answer$_2$, and so on.
	startText	string	text to display on the button that starts the signature capture
	endText	string	text to display on the button that ends the signature capture
	penColor	string	the color to use when drawing the signature on the screen. This is either a color name or a comma separated list of RGB values.
	backgroundColor	string	the color to use for the background of the signature graphic. This is either a color name or a comma separated list of RGB values.
Silanis	lock	on	sets the signature to lock all signed data. This prevents the user from changing data once it has been signed. By default, all signature types lock the data.
		off	prevents the signature from locking the data. This means users will be able to change signed data. Note that changes to signed data will still break the signature.

Note: Instead of using one of the above settings for csptype, the numeric value that is defined for it in the cryptographic API may be used. For example, csptype=dss and csptype=3 produce the same result.

Example

This example shows a button configured for a CryptoAPI signature.

```
<button sid="empSigButton">
   <type>signature</type>
   <value compute="signer"></value>
   <signer></signer>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <signformat>         application/vnd.xfdl;
      csp="Microsoft Base Cryptographic Provider v1.0";
      csptype=rsa_full; hashalg=sha1
   </signformat>
   <signoptions>
      <filter>omit</filter>
      <optiontype>triggeritem</optiontype>
      <optiontype>coordinates</optiontype>
   </signoptions>
   <signitemrefs>
      <filter>omit</filter>
      <itemref>PAGE1.mgrSigButton</itemref>
      <itemref>PAGE1.admSigButton</itemref>
      <itemref>PAGE1.empsignature</itemref>
      <itemref>PAGE1.mgrsignature</itemref>
      <itemref>PAGE1.admsignature</itemref>
   </signitemrefs>
<!-- The items listed above MUST have itemlocation
options with absolute and extent as the last
settings in order for the filter below to
be sufficient in terms of security -->
   <signoptionrefs>
      <filter>keep</filter>
      <optionref>PAGE1.mgrSigButton.itemlocation</optionref>
      <optionref>PAGE1.admSigButton.itemlocation</optionref>
      <optionref>PAGE1.empsignature.itemlocation</optionref>
      <optionref>PAGE1.mgrsignature.itemlocation</optionref>
      <optionref>PAGE1.admsignature.itemlocation</optionref>
   </signoptionrefs>
   <signature>empsignature</signature>
</button>
```

This example shows a button configured for a Clickwrap signature.

```
<button sid="empSigButton">
   <type>signature</type>
   <value compute="signer"></value>
   <signer></signer>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <signformat>application/vnd.xfdl;
      engine="ClickWrap"; hashalg="md5";
      titleText="Document Acceptance";
      echoPrompt="Type the following:";
      echoText="I agree";question1Text="Name:";
```

```
         question2Text="Employee ID #:"
      </signformat>
      <signoptions>
         <filter>omit</filter>
         <optiontype>triggeritem</optiontype>
         <optiontype>coordinates</optiontype>
      </signoptions>
      <signitemrefs>
         <filter>omit</filter>
         <itemref>PAGE1.mgrSigButton</itemref>
         <itemref>PAGE1.admSigButton</itemref>
         <itemref>PAGE1.empsignature</itemref>
         <itemref>PAGE1.mgrsignature</itemref>
         <itemref>PAGE1.admsignature</itemref>
      </signitemrefs>
      <!-- The items listed above MUST have itemlocation
      options with absolute and extent as the last
      settings in order for the filter below to
      be sufficient in terms of security -->
      <signoptionrefs>
         <filter>keep</filter>
         <optionref>PAGE1.mgrSigButton.itemlocation</optionref>
         <optionref>PAGE1.admSigButton.itemlocation</optionref>
         <optionref>PAGE1.empsignature.itemlocation</optionref>
         <optionref>PAGE1.mgrsignature.itemlocation</optionref>
         <optionref>PAGE1.admsignature.itemlocation</optionref>
      </signoptionrefs>
      <signature>empsignature</signature>
   </button>
```

Usage Details

1. An XFDL Viewer automatically copies the *signformat* option from a signature button to its associated signature item.
2. signformat is an optional setting for a button item, but is mandatory for a signature item.

signgroups

Specifies which groups are filtered for a particular signature. Filtering a group means keeping or omitting all the *cell* items that are in the specified group.

For example, if a *signgroups* option specifies that the "colorcells" group should be kept, then all cells within the "colorcells" group will be signed.

Syntax

```
<signgroups>
   <filter>group filter</filter>
   <groupref>group reference1</groupref>
      ...
   <groupref>group referencen</groupref>
</signgroups>
```

Note:

- There may be any number of *group reference* entries.

group filter	*keep*	include groups of cells in the *group reference* list with the signature; omit those not in the list

	omit	omit groups of cells in the *group reference* list from the signature; include those not in the list
group reference	*string*	identifies a group whose *cell* items will be filtered

Available In

button, signature

Example

This example shows a signgroups setting that omits the group of cells named ″monthlyPayCells″.

```
<signgroups>
   <filter>omit</filter>
   <groupref>monthlyPayCells</groupref>
</signgroups>
```

Usage Details

1. Default: **omit nothing (keep all cell items), unless the containing page is omitted**
2. It is possible to have several *list* or *popup* items with the same group reference, as these are populated with cells that have the same group reference as the item which contains them. Therefore, when setting up *signgroups* options, caution must be exercised in making group references to *list* or *popup* items which might be populated by the same group of cells.
3. Other filters may take precedence over the *signgroups* option. Refer to "Order of Precedence of Filters" on page 395 for more information on the order of precedence of filters.

signinstance

Specifies what XForms instance data is filtered for a particular signature. Filtering instances means keeping or omitting specific data from each data instance.

When instance data is omitted from a signature but the associated user interface elements are signed, the user can still enter data into those elements. Furthermore, the overlap and layout tests are not performed on those items. This leaves them free to change certain characteristics, such as size (for expanding tables or fields), to accommodate the user input. This facilitates signing the presentation layer of a form while leaving the actual data open to change.

Syntax

```
<signinstance>
   <filter>instance filter</filter>
   <dataref_1>
      <model>model ID</model>
      <ref>XPath</ref>
   </dataref>
   ...
   <dataref_n>
      ...
   </dataref>
</signinstance>
```

Note:

- There may be any number of *group reference* entries.

instance filter	*keep*	include groups of cells in the *group reference* list with the signature
	omit	omit groups of cells in the *group reference* list from the signature; include those not in the list
model ID	*string*	the ID of the <xforms:model> that the contains the data you want to filter. Set to empty to default to the first model in the form.
XPath	*XPath*	an XPath reference to the root node of the data you want to filter. All children of this node are filtered in the same manner. This reference is evaluated relative to the root node of the first instance in the model.

Available In

button, signature

Example

The following code shows an XForms model for a purchase order:

```
<xforms:model>
   <xforms:instance id="po" xmlns="">
      <po>
         <order>
         <row>
            <product/>
            <unitCost>0</unitCost>
            <qty></qty>
            <lineTotal></lineTotal>
         </row>
         </order>
         <subtotal>0</subtotal>
         <tax>0</tax>
         <total>0</total>
      </po>
   </xforms:instance>
   <xforms:instance id="temps" xmlns="">
      <root>
         <productCode/>
         <submitting>false</submitting>
      </root>
   </xforms:instance>
</xforms:model>
```

In this case, you might want to omit the temporary information that is stored in the temps instance. To do this, you would use the following filter:

```
<signinstance>
   <filter>omit</filter>
   <dataref>
      <model></model>
      <ref>instance('temps')</ref>
   </dataref>
</signinstance>
```

Usage Details

1. Default: **omit nothing (keep instance data)**
2. Avoid using other signature filters with *signinstance* except when absolutely necessary. Because signed items still accept input so long as the associated data elements are not signed, you do not need to worry about filtering most user interface elements out of the signature. However, you must still omit some elements, such as additional signature buttons, signature items, data items, and the *triggeritem* option.

signitemrefs

Specifies individual items that are filtered for a particular signature. Filtering an item reference means keeping or omitting specific items, rather than all items of a particular type (see *signitems*).

Syntax

```
<signitemrefs>
   <filter>item filter</filter>
   <itemref>item reference₁</itemref>
   ...
   <itemref>item referenceₙ</itemref>
</signitemrefs>
```

Note:

- There may be any number of *item reference* entries.

item filter	*keep*	include items in the *item reference* list with the signature; omit those not in the list
	omit	omit items in the *item reference* list from the signature; include those not in the list
item reference	*string*	specifies the item to be filtered

Available In

button, signature

Example

This sample sets the *signitemrefs* option to omit two fields from the signature:

```
<signitemrefs>
   <filter>omit</filter>
   <itemref>field1</itemref>
   <itemref>page1.field2</itemref>
</signitemrefs>
```

Usage Details

1. Default: **omit nothing (keep all items), unless the containing page is omitted**.
2. Since all items have a name and type, signitemrefs filters are always applicable.
3. When not signing the entire form, it is *strongly recommended* that an **omit** signitemrefs filter be used to exclude unwanted items, and that a **keep** signoptionrefs filter should be used to cover the layout of omitted items. See the Usage Details for the *signoptionrefs* option for details.
4. If this filter is used with the keep setting, then it is easy to add or delete items that would obscure or unobscure signed items without breaking a signature. The keep setting is useful as an optimization when you want to create a co-signature (i.e., a signature which signs another signature that uses omission logic).
5. Other filters may take precedence over the *signitemrefs* option. Refer to "Order of Precedence of Filters" on page 395 for more information on the order of precedence of filters.

signitems

Specifies which types of items filtered for a particular signature. Filtering an item means keeping or omitting all items of a particular type, rather than specific items (see *signitemrefs*).

Syntax

```
<signitems>
   <filter>item filter</filter>
   <itemtype>item type1</itemtype>
   ...
   <itemtype>item typen</itemtype>
</signitems>
```

Note:

- There may be any number of *item type* entries.

item filter	*keep*	include types of items in the *item type* list with the signature; omit those not in the list
	omit	omit types of items in the *item type* list from the signature; include those not in the list
item type	*string*	specifies the type (element tag name) of items to be filtered

Available In

button, signature

Example

This sample sets the signitems option to keep the following types of items with the signature: boxes, buttons, and fields.

```
<signitems>
   <filter>keep</filter>
   <itemtype>box</itemtype>
   <itemtype>button</itemtype>
   <itemtype>field</itemtype>
</signitems>
```

Usage Details

1. Default: **omit nothing (keep all items), unless the containing page is omitted**.
2. The only recommended use of this filter is to omit *data* items in support of the usage of the signdatagroups filter.
3. Element tag names can be specified with or without a namespace prefix. The default namespace URI is the XFDL namespace URI if no namespace prefix is given. If a namespace prefix is given, then the namespace URI to which the prefix is bound is used in determining whether or not each element matches the given filter.

   For example, the filter component <itemtype>*box*</itemtype> could equivalently be written <itemtype>*xfdl:box*</itemtype> if the prefix *xfdl* is mapped to the XFDL namespace URI.
4. Other filters may take precedence over the *signitems* option. Refer to “Order of Precedence of Filters” on page 395 for more information on the order of precedence of filters.

signnamespaces

Specifies which namespaces are filtered for a particular signature. Filtering a namespace means keeping or omitting all of the form elements and attributes that are in the specified namespace.

For example, if a *signnamespaces* option specifies that the http://www.ibm.com/xmlns/prod/XFDL/Custom namespace should be kept, then all elements in that namespace are signed.

Syntax

```
<signnamespaces>
   <filter>namespace filter</filter>
   <uri>namespace URI1</uri>
   ...
   <uri>namespace URIn</uri>
</signnamespaces>
```

Note:

- There may be any number of *namespace URI* entries.

namespace filter	*keep*	include all form elements in the *namespace URI* list with the signature; omit those not in the list
	omit	omit all form element that are in the namespaces in the *namespace URI* list from the signature; include those not in the list
namespace URI	*string*	identifies a namespace whose elements will be filtered

Available In

button, signature

Example

This example shows a *signnamespaces* setting that omits the http://www.ibm.com/xmlns/prod/XFDL/Custom namespace.

```
<signnamespaces>
   <filter>omit</filter>
   <uri>http://www.ibm.com/xmlns/prod/XFDL/Custom</uri>
</signnamespaces>
```

Usage Details

1. Default: **omit nothing (keep all namespaces)**.
2. Other filters may take precedence over the *signnamespaces* option. Refer to "Order of Precedence of Filters" on page 395 for more information on the order of precedence of filters.

signoptionrefs

Specifies individual options that are filtered for a particular signature. Filtering option references means keeping or omitting specific options, rather than all options of a particular type (see *signoptions*).

Syntax

```
<signoptionrefs>
   <filter>option filter</filter>
   <optionref>option reference₁</optionref>
   ...
   <optionref>option referenceₙ</optionref>
</signoptionrefs>
```

Note:

- There may be any number of *option reference* entries.

option filter	*keep*	include options in the *option reference* list with the signature; omit those not in the list
	omit	omit options in the *option reference* list from the signature; include those not in the list
option reference	*string*	specifies the option to be filtered

Available In

button, signature

Example

This example specifies a signoptionrefs setting that keeps a particular field with the signature.

```
<signoptionrefs>
   <filter>keep</filter>
   <optionref>page1.field1.value</optionref>
</signoptionrefs>
```

Note: The page name may be dropped if the option in question is on the same page, but the item name must not be dropped.

Usage Details

1. Default: **keep all options (omit nothing), unless the containing item is omitted**.
2. Note that, unlike signoptions, the signoptionrefs filter can cause an item to be included even if the item filters would normally omit the item. This is necessary in order to ensure that the hashed text of a signature is in valid XFDL format.
3. It is *strongly recommended* that signoptionrefs be used to **keep** the itemlocation of items that have been omitted, and that all omitted items have an itemlocation with absolute and extent settings as the last two settings.
4. Element tag names can be specified with or without a namespace prefix. The default namespace URI is the XFDL namespace URI if no namespace prefix is given. If a namespace prefix is given, then the namespace URI to which the prefix is bound is used in determining whether or not each element matches the given filter.

   For exampe, the filter component <optionref>*page1.field1.value*</optionref> could equivalently be written <optionref>*page1.field1.xfdl:value*</optionref> if the prefix *xfdl* is mapped to the XFDL namespace URI.

5. Other filters may take precedence over the *signoptionrefs* option. Refer to "Order of Precedence of Filters" on page 395 for more information on the order of precedence of filters.

signoptions

Specifies which types of options are filtered for a particular signature. Filtering options means keeping or omitting all options of a particular type, rather than specific options (see *signoptionrefs*).

Syntax

```
<signoptions>
   <filter>option filter</filter>
   <optiontype>option type1</optiontype>
   ...
   <optiontype>option typen</optiontype>
</signoptions>
```

Note:

- There may be any number of *option type* entries.

option filter	*keep*	include types of options in the *option type* list with the signature; omit those not in the list
	omit	omit types of options in the *option type* list from the signature; include those not in the list
option type	*string*	specifies the type (element tag name) of options to be filtered

Available In

button, signature

Example

This example shows a signoptions setting that omits two types of options from the signature.

```
<signoptions>
   <filter>omit</filter>
   <optiontype>url</optiontype>
   <optiontype>printsettings</optiontype>
</signoptions>
```

Usage Details

1. Default: **keep all options (omit nothing), unless the containing item is omitted**.
2. One signoptions setting must always be specified in the following way:

```
<signoptions>
   <filter>omit</filter>
   <optiontype>triggeritem</optiontype>
   <optiontype>coordinates</optiontype>
</signoptions>
```

- This setting ensures that the signature will not be broken due to an alteration to the form.
- If the this option must be used as a keep filter, then consider omitting the *global.global.triggeritem* and the coordinates of image buttons using a signoptionrefs filter.

3. Element tag names can be specified with or without a namespace prefix. The default namespace URI is the XFDL namespace URI if no namespace prefix is given. If a namespace prefix is given, then the namespace URI to which the prefix is bound is used in determining whether or not each element matches the given filter component.

   For example, the filter component <optiontype>*triggeritem*</optiontype> could be equivalently written as <optiontype>*xfdl:triggeritem*</optiontype> if the prefix *xfdl* is mapped to the XFDL namespace URI.
4. Other filters may take precedence over the *signoptions* option. Refer to "Order of Precedence of Filters" on page 395 for more information on the order of precedence of filters.

signpagerefs

Specifies individual pages that are filtered for a particular signature. Filtering pages means keeping or omitting a page and all of its contents.

Syntax

```
<signpagerefs>
   <filter>page filter</filter>
   <pageref>page reference1</pageref>
   ...
   <pageref>page referencen</pageref>
</signpagerefs>
```

Note:

- There may be any number of *page reference* entries.

page filter	*keep*	include pages in the *page reference* list with the signature; omit those not in the list
	omit	omit pages in the *page reference* list from the signature; include those not in the list
page reference	*string*	specifies the page to be filtered

Available In

button, signature

Example

This sample sets the signpagerefs option to omit two pages from the signature.

```
<signpagerefs>
   <filter>omit</filter>
   <pageref>page1</pageref>
   <pageref>page2</pageref>
</signpagerefs>
```

Usage Details

1. Default: **keep all pages**.
2. Other filters may take precedence over the *signpagerefs* option. Refer to "Order of Precedence of Filters" on page 395 for more information on the order of precedence of filters.

size

Specifies an item's size. It does not include external labels, borders, or scroll bars. These are part of the bounding box size which is calculated automatically.

Examples of item size are the input area in a *field* item or the height and width of the label in *label* and *button* items.

Syntax

```
<size>
   <width>width</width>
   <height>height</height>
</size>
```

width	*unsigned byte*	the horizontal dimension of the item, measured in characters
height	*unsigned byte*	the vertical dimension of the item, measured in lines

Available In

box, button, check, combobox, field, label, line, list, popup, radio, slider, spacer

Example

This sample sets the item's size to 80 characters wide by five lines high.

```
<size>
   <width>80</width>
   <height>5</height>
</size>
```

Usage Details

1. Default: refer to "Default Sizes" on page 393 for a complete list of default sizes.
2. Size and Font:
   - The width might not always accommodate the number of characters specified. The calculation to determine actual width is:
   - *width* * (average character width for the item's font)
   - The width will only exactly match the number of characters the item can display horizontally when the font is mono-spaced (like Courier).
3. If either the height or the width is invalid, the default item size will be used. A dimension of zero (0) is invalid for all items except the *line* item.
4. The item and bounding box sizes can be changed by using *itemlocation* with an **expansion** or **extent** modifier. This will override the *size* option.

suppresslabel

Suppresses the built-in label for some items, so that the label is not shown even if the *label* option or *xforms:label* option is set.

This is most useful when you are using XFDL to wrap an XForm control that includes labels that are not necessary in the visual presentation. For example, you might not want to display the labels of items in a table.

When the label is suppressed, the item is both displayed and printed as if no label were present at all. This means that both the appearance and size match an equivalent item with no label.

Syntax

```
<suppresslabel>status</suppresslabel>
```

status	*on*	suppress the item's built-in label.
	off	do not suppress the item's built-in label.

Available In

button, check, checkgroup, combobox, field, list, label, pane, radio, radiogroup, slider

Example

This example shows a table with one field in each row. The *suppresslabel* option has a compute that deterimines whether the field is in the first row, and suppresses the label if the field is not. This effectively creates a title row of labels on the first row of the table, then suppresses the labels for all subsequent rows.

```
<table sid="customerTable">
   <xforms:repeat nodeset="customer">
      <field sid="nameField">
         <xforms:input ref="name">
            <xforms:label>Name:</xforms:label>
         </xforms:input>
         <suppresslabel compute="xforms.getPosInSet() == '1' ?
            'off' : 'on'"/>
      </field>
   </xforms:repeat>
</table>
```

Usage Details

1. Default: **off**.

texttype

Specifies whether a field uses plain text or rich text.

Syntax

```
<texttype>type</texttype>
```

type	*string*	the type of text to use. Valid options are: • text/plain • text/rtf

Available In

field

Example

This sample sets a field to use rich text.

```
<field sid="richTextField">
   <texttype>text/rtf</texttype>
   <value>Hello</value>
   <rtf>rich text version of Hello</rtf>
</field>
```

Usage Details

1. Default: **text/plain**
2. If using rich text:
   - The rich text is stored in the *rtf* option.
   - A plain text version of the rich text is stored in the *value* option (refer to the *rtf* option for more information).

thickness

Specifies the thickness of a *line* item. The unit of measurement is pixels.

Syntax

```
<thickness>thickness</thickness>
```

thickness	*unsigned byte*	the thickness of the line

Available In

line

Example

This sample defines a horizontal line 40 characters long and five pixels thick:

```
<size>
   <width>40</width>
   <height>0</height>
</size>
<thickness>5</thickness>
```

Usage Details

1. Default: **one pixel**
2. Use size to define the dimension of a line in one direction (height or width) and thickness to define the dimension in the other direction. The dimension thickness defines must be set to zero in size.
3. The line's thickness may be changed by using itemlocation with an expansion modifier for the dimension that thickness describes.

transmitdatagroups

Specifies which datagroups are filtered when the form is transmitted. Filtering a datagroup means keeping or omitting all the *data* items that are in the specified datagroup.

For example, if a *transmitdatagroups* option specifies that the ″attachments″ datagroup should be kept, then all data items within the ″attachments″ group will be transmitted.

This filter applies to all *data* items present when transmitted, including those added as enclosures.

Syntax

```
<transmitdatagroups>
   <filter>datagroup filter</filter>
   <datagroupref>datagroup reference1</datagroupref>
   ...
   <datagroupref>datagroup referencen</datagroupref>
</transmitdatagroups>
```

Note:

- There may be any number of *datagroup reference* entries.

datagroup filter	*keep*	include datagroups in the *datagroup reference* list with the transmission; omit those not in the list
	omit	omit datagroups in the *datagroup reference* list from the transmission; keep those not in the list
datagroup reference	*string*	identifies a datagroup whose *data* items will be filtered

Available In

action, button, cell

Examples

This sample specifies that only items of type data with a datagroup setting of enclosures should be transmitted:

```
<transmitdatagroups>
   <filter>keep</filter>
   <datagroupref>enclosures</datagroupref>
</transmitdatagroups>
```

This sample specifies that all items of type data except those with a datagroup setting of others should be kept in the transmission:

```
<transmitdatagroups>
   <filter>omit</filter>
   <datagroupref>others</datagroupref>
</transmitdatagroups>
```

Usage Details

1. Default: **omit nothing (keep all data items), unless the containing page is omitted**.
2. Since enclosed files can belong to several datagroups, and datagroups can contain several enclosed files, care must be exercised when setting up transmitdatagroups options to ensure that only the desired datagroups are filtered.
3. Other filters may take precedence over the *transmitdatagroups* option. Refer to "Order of Precedence of Filters" on page 395 for more information on the order of precedence of filters.

transmitformat

Specifies the format of the form data submitted to a processing application. An XFDL form can submit data in XFDL format or in HTML format. Furthermore, the XFDL format may be compressed using ASCII compression.

XFDL format submits the entire form definition, including user input.

HTML format submits just an assignment statement for each item equating the item reference with the item's value. The only items included are modifiable items, custom items, and items with a *transmit* option setting of all.

Note: Form and page globals are sent only if the format is XFDL.

Syntax

```
<transmitformat>MIME type</transmitformat>
```

MIME type	*application/vnd.xfdl*	use XFDL format
	application/vnd.xfdl; content-encoding="base64-gzip"	use compressed XFDL format
	application/x-www-form-urlencoded	use HTML form format

Available In

action, button, cell, page global, form global

Examples

XFDL format

This example shows a button which, when clicked, will submit the form in XFDL format.

```
<button sid="send_button">
   <type>done</type>
   <url>mailto:rrunner@acme.com</url>
   <transmitformat>application/vnd.xfdl</transmitformat>
</button>
```

When a user clicks the button, the entire form definition will be submitted, unless other transmit options specify a partial submission.

HTML form format

This sample shows an automatic action that submits form data in HTML form format.

```
<action sid="status_action">
   <type>submit</type>
   <url>http://www.host.domain/cgi-bin/recvStatus</url>
   <transmitformat>
      application/x-www-form-urlencoded
   </transmitformat>
   <delay>
      <type>repeat</type>
      <interval>180</interval>
   </delay>
</action>
```

Every 180 seconds, the form definition will be converted to HTML form format.

HTML Format by Item Type

The general syntax of a submitted HTML form is:

```
item reference=value&item reference=value&...
```

Note:

- The ampersand separates form items.

The syntax of an HTML form entry by item type:

Item Type	HTML Format
field	item sid=value option setting
list	item sid=value option setting of selected cell Note that the item reference identifies the list.
popup	item sid=value option setting of selected cell Note that the item reference identifies the popup.
combobox	item sid=value option setting
check	item sid=value option setting
radio	group option setting=item sid of selected radio
custom	item sid=value option setting
all other items	item sid=value option setting

Substitutions and Omissions:

- Spaces in the value are replaced by the plus sign (+). For example, ’Two words’ becomes ’Two+words’.
- The membership operator in item and group references is replaced by a minus sign.
- *page_one.age_group* becomes *page_one-age_group*.
- Page tags are removed from item and group references in single page forms.
- Check boxes and radio buttons with a *value* option setting of off are omitted.
- Entries resulting in an empty string on the right hand side of the assignment statement are omitted. This occurs when the referenced option setting is empty or the option definition is missing.

HTML Considerations

The functionality of XFDL forms differs somewhat from HTML forms. Those differences are:

- **Enclosures —** HTML does not support enclosures. To submit enclosed form data, use XFDL format.
- **Item tags —** XFDL allows a smaller set of characters in item tags than HTML does. XFDL item tags support the following characters: a-z, A-Z, 0-9, and the underscore (_).
- **Check boxes —** XFDL check boxes vary slightly from HTML check boxes. XFDL check boxes are independent items; HTML check boxes are grouped together using the same format as *radio* items. When an XFDL form is submitted in HTML format, the submission will contain an entry for each check box.

Usage Details

1. Default: The default is the format that the form was in before it was parsed. For example, a form written in XFDL will be transmitted in XFDL unless otherwise specified by this option.
2. This option can be included as a form global option and in the definitions of items that trigger form submissions. These items have a *type* option setting of **submit** or **done**.

transmitgroups

Specifies which groups are filtered for a particular trasmission. Filtering a group means keeping or omitting all the *cell* items that are in the specified group.

For example, if a *transmitgroups* option specifies that the ″colorcell″ group should be kept, then all cells within the ″colorcell″ group will be transmitted.

Syntax

```
<transmitgroups>
   <filter>transmit flag</filter>
   <groupref>group reference₁</groupref>
   ...
   <groupref>group referenceₙ</groupref>
</transmitgroups>
```

Note:

- There may be any number of *group reference* entries.

transmit flag	*keep*	include groups of cells in the *group reference* list with the transmission; omit those not in the list
	omit	omit groups of cells in the *group reference* list from the transmission; include those not in the list
group reference	*string*	identifies a group whose *cell* items will be filtered

Available In

action, button, cell

Examples

This sample specifies that only the items in the ″countryCells″ and ″departmentCells″ groups should be kept in the transmission.

```
<transmitgroups>
   <filter>keep</filter>
   <groupref>countryCells</groupref>
   <groupref>departmentCells</groupref>
</transmitgroups>
```

This sample specifies that all groups of cells should be kept in the transmission except for ″firstNameCells″ group.

```
<transmitgroups>
   <filter>omit</filter>
   <groupref>firstNameCells</groupref>
</transmitgroups>
```

Usage Details

1. Default: **omit nothing (keep all cell items), unless the containing page is omitted**.
2. It is possible to have several *list* or *popup* items with the same group reference, as these are populated with cells that have the same group reference as the item which contains them. Therefore, when setting up a *transmitgroups* option, caution must be exercised in making group references to *list* or *popup* items which might be populated by the same group of cells.
3. Other filters may take precedence over the *transmitgroups* option. Refer to “Order of Precedence of Filters” on page 395 for more information on the order of precedence of filters.

transmititemrefs

Specifies individual items that are filtered for a particular transmission. Filtering item references means keeping or omitting individual items, rather than all items of a particular type (see *transmititems*).

Syntax

```
<transmititemrefs>
   <filter>transmit flag</filter>
   <itemref>item reference1</itemref>
   ...
   <itemref>item referencen</itemref>
</transmititemrefs>
```

Note:

- There may be any number of item reference entries.

transmit flag	*keep*	include items in the *item reference* list with the transmission; omit those not in the list
	omit	omit items in the *item reference* list from the transmission; include those not in the list
item reference	*string*	identifies the item to be filtered

Available In

action, button, cell

Examples

This sample specifies that only the item on page1 called ″MgrSignButton″ should be transmitted, and that all other items should be omitted.

```
<transmititemrefs>
   <filter>keep</filter>
   <itemref>page1.MgrSignButton</itemref>
</transmititemrefs>
```

This sample shows how you would use transmititemrefs in conjunction with transmititems: although all items that are buttons are omitted, the button on page1 called ″MgrSignButton″ will be kept.

```
<transmititems>
   <filter>omit</filter>
   <itemtype>button</itemtype>
</transmititems>
<transmititemrefs>
   <filter>keep</filter>
   <itemref>page1.MgrSignButton</itemref>
</transmititemrefs>
```

Usage Details

1. Default: omit nothing (keep all items), unles the containing page is omitted

2. Other filters may take precedence over the *transmititemrefs* option. Refer to "Order of Precedence of Filters" on page 395 for more information on the order of precedence of filters.

transmititems

Specifies types of items that are filtered for a particular transmission. Filtering items means keeping or omitting all items of a particular type, rather than individual items (see *transmititemrefs*).

Syntax

```
<transmititems>
   <filter>transmit flag</filter>
   <itemtype>item type1</itemtype>
   ...
   <itemtype>item typen</itemtype>
</transmititems>
```

Note:

- There may be any number of *item type* entries.

transmit flag	*keep*	include types of items in the *item type* list with the transmission; omit those not in the list
	omit	omit types of items in the *item type* list from the transmission; include those not in the list
item type	*string*	identifies the type (element tag name) of items to be filtered

Available In

action, button, cell

Example

This sample specifies that *box*, *help*, *label*, *spacer*, and *toolbar* items should be omitted from the form data submitted to the form processing application.

```
<transmititems>
   <filter>omit</filter>
   <itemtype>box</itemtype>
   <itemtype>help</itemtype>
   <itemtype>spacer</itemtype>
   <itemtype>toolbar</itemtype>
</transmititems>
```

Usage Details

1. Default: **omit nothing (keep all items), unless the containing page is omitted**.
2. Element tag names can be specified with or without a namespace prefix. The default namespace URI is the XFDL namespace URI if no namespace prefix is given. If a namespace prefix is given, then the namespace URI to which the prefix is bound is used in determining whether or not each element matches the given filter.

For example, the filter component <itemtype>*box*</itemtype> could equivalently be written <itemtype>*xfdl:box*</itemtype> if the prefix *xfdl* is mapped to the XFDL namespace URI.

3. Other filters may take precedence over the *transmititems* option. Refer to "Order of Precedence of Filters" on page 395 for more information on the order of precedence of filters.

transmitnamespaces

Specifies which namespaces are filtered for a particular transmission. Filtering a namespace means keeping or omitting all of the form elements and attributes that are in the specified namespace.

For example, if a *transmitnamespaces* option specifies that the http://www.ibm.com/xmlns/prod/XFDL/Custom namespace should be kept, then all elements in that namespace are transmitted.

Syntax

```
<transmitnamespaces>
   <filter>namespace filter</filter>
   <uri>namespace URI1</uri>
   ...
   <uri>namespace URIn</uri>
</transmitnamespace>
```

Note:

- There may be any number of *namespace URI* entries.

namespace filter	*keep*	include all form elements in the *namespace URI* list with the transmission; omit those not in the list
	omit	omit all form element that are in the namespaces in the *namespace URI* list from the transmission; include those not in the list
namespace URI	*string*	identifies a namespace whose elements will be filtered

Available In

action, button, cell

Example

This example shows a *transmitnamespaces* setting that omits the http://www.ibm.com/xmlns/prod/XFDL/Custom namespace.

```
<transmitnamespaces>
   <filter>omit</filter>
   <uri>http://www.ibm.com/xmlns/prod/XFDL/Custom</uri>
</transmitnamespaces>
```

Usage Details

1. Default: **omit nothing (keep all namespaces)**.

2. Other filters may take precedence over the *transmitnamespaces* option. Refer to "Order of Precedence of Filters" on page 395 for more information on the order of precedence of filters.

transmitoptionrefs

Specifies individual options that are filtered for a particular transmission. Filtering option references means keeping or omitting individual options, rather than all options of a particular type (see *transmitoptions*).

Syntax

```
<transmitoptionrefs>
   <filter>transmit flag</filter>
   <optionref>option reference1</optionref>
   ...
   <optionref>option referencen</optionref>
</transmitoptionrefs>
```

Note:

- There may be any number of *option reference* entries.

transmit flag	*keep*	include options in the *option reference* list with the transmission; omit those not in the list
	omit	omit options in the *option reference* list from the transmission; include those not in the list
option reference	*string*	identifies the option to be filtered

Available In

action, button, cell

Examples

This sample shows how you would use transmitoptionrefs in conjunction with transmitoptions: although all options that are *values* are omitted, the value in the ″NameField″ on ″page1″ will be kept.

```
<transmitoptions>
   <filter>omit</filter>
   <optiontype>value</optiontype>
</transmitoptions>
<transmitoptionrefs>
   <filter>keep</filter>
   <optionref>page1.NameField.value</optionref>
</transmitoptionrefs>
```

This sample shows how you would use transmitoptionrefs in conjunction with transmititemrefs: although the item called ″MgrSignButton″ on ″page1″ is omitted, its *signer* option is kept

```
<transmititemrefs>
   <filter>omit</filter>
   <itemref>MgrSignButton</itemref>
</transmititemrefs>
```

```
<transmitoptionrefs>
   <filter>keep</filter>
   <optionref>page1.MgrSignButton.signature</optionref>
</transmitoptionrefs>
```

Usage Details

1. Default: **keep all options (omit nothing), unless the containing item is omitted**.
2. Element tag names can be specified with or without a namespace prefix. The default namespace URI is the XFDL namespace URI if no namespace prefix is given. If a namespace prefix is given, then the namespace URI to which the prefix is bound is used in determining whether or not each element matches the given filter.

   For exampe, the filter component <optionref>*page1.field1.value*</optionref> could equivalently be written <optionref>*page1.field1.xfdl:value*</optionref> if the prefix *xfdl* is mapped to the XFDL namespace URI.
3. Other filters may take precedence over the *transmitoptionrefs* option. Refer to "Order of Precedence of Filters" on page 395 for more information on the order of precedence of filters.

transmitoptions

Specifies types of options that are filtered for a particular transmission. Filtering options means keeping or omitting all options of a particular type, rather than individual items (see *transmititemrefs*).

Syntax

```
<transmitoptions>
   <filter>transmit flag</filter>
   <optiontype>option type1</optiontype>
   ...
   <optiontype>option typen</optiontype>
</transmitoptions>
```

Note:

- There may be any number of *option type* entries.

transmit flag	*keep*	include types of options in the *option type* list with the transmission; omit those not in the list
	omit	omit types of options in the *option type* list from the transmission; include those not in the list
option type	*string*	specifies the type (element tag name) of options to be filtered

Available In

action, button, cell

Example

This sample specifies that only the *active*, *mimedata*, and *value* options should be included in the form data submitted to the form processing application.

```
<transmitoptions>
   <filter>keep</filter>
   <optiontype>active</optiontype>
   <optiontype>mimedata</optiontype>
   <optiontype>value</optiontype>
</transmitoptions>
```

Usage Details

1. Default: **keep all options (omit nothing), unless the containing item is omitted**.
2. Element tag names can be specified with or without a namespace prefix. The default namespace URI is the XFDL namespace URI if no namespace prefix is given. If a namespace prefix is given, then the namespace URI to which the prefix is bound is used in determining whether or not each element matches the given filter component.

   For example, the filter component <optiontype>*triggeritem*</optiontype> could be equivalently written as <optiontype>*xfdl:triggeritem*</optiontype> if the prefix *xfdl* is mapped to the XFDL namespace URI.
3. Other filters may take precedence over the *transmitoptions* option. Refer to "Order of Precedence of Filters" on page 395 for more information on the order of precedence of filters.

transmitpagerefs

Specifies individual pages that are filtered for a particular signature. Filtering pages means keeping or omitting a page and all of its contents.

Syntax

```
<transmitpagerefs>
   <filter>transmit flag</filter>
   <pageref>page reference1</pageref>
   ...
   <pageref>page referencen</pageref>
</transmitpagerefs>
```

Note:

- There may be any number of *page reference* entries.

transmit flag	*keep*	include pages in the *page reference* list with the transmission; omit those not in the list
	omit	omit pages in the page reference list from the transmission; include those not in the list
page reference	*string*	specifies the page to be filtered

Available In

action, button, cell

Examples

This sample specifies that only page1 should be transmitted, and that all other pages should be omitted:

```
<transmitpagerefs>
   <filter>keep</filter>
   <pageref>page1</pageref>
</transmitpagerefs>
```

Usage Details

1. Default: **keep all pages**
2. Other filters may take precedence over the *transmitpagerefs* option. Refer to “Order of Precedence of Filters” on page 395 for more information on the order of precedence of filters.

triggeritem

Identifies the item that triggered a form submission. Items triggering form submissions have a *type* option setting of refresh, submit, or done.

When a user selects an item that triggers a form submission, the *triggeritem* option is added to the form globals and assigned the item reference of the selected item.

Syntax

```
<triggeritem>item reference</triggeritem>
```

item reference	*string*	identifies the trigger item

Available In

form global

Example

This sample indicates that the item triggering the request is on the page called ″Page_one″ and has is called ″submit_button″.

```
<triggeritem>Page_one.submit_button</triggeritem>
```

Usage Details

1. Actions of type **submit** or **done** set the *triggeritem* to the SID of the triggering item. Actions of type **refresh** first clear the *triggeritem* by setting it to empty (″″), then set the *triggeritem* to the SID of the triggering item.

type

Associates a task with an item that can trigger a task: *action*, *button*, or *cell*.

Syntax

```
<type>action type</type>
```

action type	*(see below)*	the task to perform

Tasks

The tasks can be any of the following:

Tasks	**Description of Tasks**	**Use with the Following Items**
cancel	Close the form; if any changes were made to the form since the last save or submit, then the user is informed that the form has changed and is allowed to choose whether the cancellation will proceed. Note that the value options of many items, as well as the contents of data items, may change in response to an enclose or remove action.	action, button, cell
display	Display an enclosed file. The web browser will choose the appropriate viewer according to the file's MIME type.	action, button cell
done	Perform a submit followed by a cancel.	action, button, cell
enclose	Allows the user to place one or more files into one or more of the datagroups defined for the form. The files will be encoded using base64 encoding format.	button, cell
extract	Allows a user to copy the contents of an enclosed file into a file on the local disk.	button, cell
link	Perform all requests specified by the url options in the current item. See section ″8.93 url″ for more details.	action, button, cell
pagedone	Move to the page specified in the url option. This closes the current page and replaces it with the new page. All fields containing error checking on the current page must be correctly filled out before it can be closed.	action, button, cell
print	Print the form on a local printer.	action, button, cell
refresh	Sets the triggeritem to ″″ and then to the full reference (including scope ID) of the item that triggered the refresh.	action, button, cell
remove	Allows the user to remove an item from a datagroup; the underlying data item will only be deleted if it belongs to no other datagroups.	button, cell
replace	Perform a link followed by a cancel.	action, button, cell

Tasks	Description of Tasks	Use with the Following Items
saveform	Saves the form to the current filename and location. If no filename has been set, prompts the user for a filename and location, then saves the file.	action, button, cell
saveas	Prompts the user for a filename and save location, then saves the file.	action, button, cell
select	With *action* items: the item's *active* option goes from off, to on, to off again. Additionally, if an *xforms:action* is contained within the *action* item, then the *xforms:action* is triggered. With *button* items containing images: stores where on the image the button was clicked in the *coordinates* option. With *cell* items: flags the cell as selected when a user chooses the cell. This means the item reference of the cell is copied to the value option of the parent list or popup.	action, button, cell
signature	Create a signature.	button
submit	Initiate the form processing applications identified in the url options of the current item.	action, button, cell

Available In

action, button, cell

Example

This sample specifies that ″BUTTON1″ saves the form to a local file.

```
<button sid="BUTTON1">
   <value>Save</value>
   <type>saveas</type>
</button>
```

Usage Details

1. Default: **select**

url

Provides the url to a target, such as a file or application. Items containing this option must have a *type* option setting of link, replace, submit, done, or pagedone.

The object identified must be one of the following:

- **File —** Used with a *type* option of link or replace. The file identified is downloaded, and either displayed or saved. Examples of such files are images, word processing documents, and XFDL forms.

- **Application —** Used with a *type* option of submit or done. The application identified is initiated. A form processing application, such as a cgi or a servlet, is an example of such an application.
- **Item —** Used with a *type* option of pagedone. The item identified, on the page identified, receives focus. The item must be on another page.
- **Form or Page Globals —** Used with a *type* option setting of pagedone. The focus moves to the first item on the page when the new form or page appears. The form globals reference is global.global. The page globals reference is <page sid>.global for another page
- **e-mail Address —** Used with a *type* option of submit, done, link, or replace. With a submit or done type, the form is attached to an e-mail message, and that message is sent to the address in the url. With a link or replace type, an e-mail message is created and sent, but the form is not attached to the message. Depending on the settings you use, the user may be able to add additional information to the e-mail.

Syntax

```
<url>the URL</url>
```

Note:

- *item reference* can be an item, form or page global reference.

the URL	*string*	identifies the target. Can be one of: • A URL with the format: scheme://host.domain[:port]/path/filename for files and applications. *Scheme* is restricted to http or https. • A URL with a *mailto* format. See "URLs for e-mail" below for further information. • #*item reference* for the next item in the form to receive focus.

URLs for e-mail

URLs that provide an e-mail address must follow this general format:

```
mailto:address?parameter=setting&parameter=setting...
```

The first parameter follows the question mark (?) symbol, while each additional parameter is added using the ampersand (&) symbol.

For example, a URL using all parameters would look like this:

```
mailto:setting?to=setting&cc=setting&bcc=setting&
subject=setting&body=setting&filename=setting
```

The following table lists the available parameters and their settings:

Parameter	Setting	Description
mailto: to= cc= bcc=	*string*	a complete e-mail address, such as john@acme.com. To include additional addresses, use the appropriate parameter twice. For example, to add two cc addresses, use the cc= parameter twice as shown: `mailto:john@acme.com?cc=bob@acme.com &cc=fred@acme.com` Note that the first address, immediately after the mailto: parameter, is the first recipient. Additional recipients are specified using the to= parameter.
subject=	*string*	this is the subject line of the e-mail. The text must conform to standard URL encoding rules, such as replacing spaces with the plus (+) symbol.
body=	*string*	this is the body of the e-mail. The text must conform to standard URL encoding rules, such as replacing spaces with the plus (+) symbol.
filename=	*string*	this is the name you want to give to the file that is attached to the e-mail message. If you do not set this parameter, a default file name will be assigned.

If you provide the mailto:, cc=, bcc=, subject=, and body= parameters, the e-mail will be sent automatically and the user will not be able to modify the message. This is true even if the parameters are set to nothing. For example, the following URL would mail the message automatically:

```
mailto:tim@acme.com?cc=&bcc=&subject=Hello&body=Hello+Tim&body=
```

If you leave out any of those parameters, the user will see the e-mail message before it is sent, and will be able to change the e-mail.

Note: The ampersand is a restricted character in XML. As such, you must either use an entity reference (&amp;) or enclose the mailto URL in a CDATA section when using the ampersand. See below for an example that uses CDATA.

Available In

action, button, cell

Example

This sample identifies a form processing application:

```
<url>http://www.host.domain/cgi-bin/recv_status</url>
```

This sample identifies a page to display:

```
<url>#PAGE2.global</url>
```

This sample creates an e-mail message that is sent automatically because it contains all of the necessary parameters. Note that the URL is enclosed in the CDATA construct because it contains ampersands (&).

```
<url><![CDATA[mailto:john@acme.com?&subject=Hello&
   body=Hello.+How+are+you?]]></url>
```

This sample creates an e-mail message that appears to the user before sending, allowing the user to change the parameters.

```
<url>mailto:john@acme.com</url>
```

Usage Details

1. Default: **none**
2. You can only list a single URL.
3. You can create a URL that includes computed values, as shown:
   ```
   <url compute="PAGE1.FIELD1.value"></url>
   ```
4. You can create a URL that includes user input as part of the URL string. Ensure that you concatenate (+.) the elements of the string. Additionally, you must contain the e-mail parameters (mailto, &subject, and so on) within quotation marks. For example:
   ```
   <url compute="'mailto:' +. to_field.value +.      &#xA;
      '&amp;subject=' +. subject_field.value +. '&amp;body=' &#xA;
      +. body_field.value></url>
   ```
5. If you have specified an HTML *transmitformat* in a form, the form sends its data as HTML when it communicates with a server. Information transmitted in HTML is URL-encoded. Therefore, for forms transmitted in HTML, you must replace all non alpha-numeric characters with a character triplet consisting of the % character followed by two hexadecimal digits. These hexadecimal digits are derived from the ASCII code for the original character. The hexadecimal digits are "0123456789ABCDEF". For example:

Character	ASCII Code	URL-encoded triplet
<space>	32	%20
\r	13	%0D

   Applications receiving form data must check the content type of the incoming data to see whether it is url-encoded.

value

Reflects the contents of an item. Visually, this can take several forms, depending on the item to which it applies. For example, the *value* option in *label* items contains the label text; the *value* option in *radio* items contains a status indicator; and the *value* option in *list* items contains the scope identifer (sid) of the most recently selected cell (if it was a select cell).

An item's contents will be stored in the form whenever a user saves the form or submits it for processing. This is true even for inactive items and items using the default *value* option setting (in this case, a *value* option containing the default setting is added to the item's definition).

Syntax

```
<value>setting</value>
```

setting	*string*	the item's contents

Available In

button, cell, check, checkgroup, combobox, field, help, label, list, popup, radio, radiogroup, slider

Example

This sample identifies the text of a label item.

```
<value>My Form Title</value>
```

Usage Details

1. Default: varies by item. Refer to the item in question for more information.
2. Rich text fields use both the *value* and the *rtf* option to store the text of the field. Refer to the *rtf* option for more information.
3. Multiple line values need to have carriage returns inserted in the code. For example:
   ```
   <value>This value spans
   two lines.</value>
   ```
4. To get the value of a cell that a user has selected from a list or popup, dereference it in the following manner:
   ```
   page_tag.list_tag.value->value
   ```
   For example:
   ```
   page1.countryPopup.value->value
   ```
   When a user selects a cell from a list, the scope identifer (sid) of the cell is stored as the value of the list. Hence the dereference syntax.

visible

Defines whether or not the item is visible on the screen and can be printed.

Syntax

```
<visible>status</visible>
```

status	*on*	item can be seen on the screen and printed
	off	item cannot be seen on the screen and will not print when the form is printed.

Available In

box, button, check, checkgroup, combobox, field, label, line, list, popup, radio, radiogroup, slider

Example

This sample shows how an item can be set to be visible at the user's request.

```
<check sid="SHOW_INSTRUCTIONS">
   <value>off</value>
   <label>Do you want to see the instructions?</label>
</check>
<label sid="INSTRUCTION_LABEL">
```

```
      <visible compute="SHOW_INSTRUCTIONS.value=='on' ? &#xA;
         'on' : 'off'"></visible>
      <value>Please complete all portions of this form.</value>
   </label>
```

Usage Details

1. Default:
   - XFDL: **on**.
   - XForms: defaults to the *relevant* property for the data element to which the containing item is bound.
2. An XFDL item in an *xforms:group*, *xforms:repeat*, or *xforms:case* will not be visible if:
   - The XFDL item containing the *xforms:group*, *xforms:repeat*, or *xforms:case* is not visible.
   - The *xforms:case* is not selected.
   - The row containing the XFDL item (created by the *xforms:repeat*) is non-relevant.

     These cases are true regardless of the *visible* option for that XFDL item, and regardless of whether the XFDL item itself contains XForms options.

webservices

Enables you to embed a WSDL document within a form. This makes the functions defined in the WSDL document available to the form as though they were XFDL functions.

Syntax

```
<webservices>
   <wsdl name="webservice name">wsdl1</wsdl>
   ...
   <wsdl name="webservice name">wsdln</wsdl>
</webservices>
```

webservice name	*string*	the name of the Web Service.
wsdl	*n/a*	the WSDL document.

Available In

form global

Example

```
   <?xml version="1.0"?>
   <XFDL xmlns="http://www.ibm.com/xmlns/prod/XFDL/7.0"
      xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
      xmlns:xforms="http://www.w3.org/2002/xforms">
      <globalpage sid="global">
         <global sid="global">
            <webservices>
               <wsdl name="name of webservice">
                  ...wsdl document...
               </wsdl>
            </webservices>
         </global>
      </globalpage>
```

Usage Details

1. To call a web service message or function, you must use both the name of the web service package and the name of the message or operation. The package name has two parts: the service name and the port type. These two parts are separated by an underscore. The syntax of this call is:

   ```
   <service_name>_<portType>.<message_name>
   ```

   For example, the following reference calls the HelloWorld operation inside the Service1 web service. It uses the Service1Soap port type:

   ```
   Service1_Service1Soap.HelloWorld()
   ```
2. Web services must not include the underscore character (_) in either service or port names, but can include it in operation names.
3. Web services must not use mandatory headers, as defined by the *soap:mustUnderstand* tag.
4. Web services are restricted to the 46 primitive data types as defined by schema. Third party extension to the primitive data types are not supported.
5. Web services may use basic or digest authentication. In either case, authentication must be performed before calling any functions in the web service. This is accomplished by calling the *setNetworkPassword* function, which is automatically created in a package with the same name as the web service. Note that when calling the *setNetworkPassword* function, you need to include the service name, but not the port type. For example:

   ```
   Service1.setNetworkPassword()
   ```
6. Web services do not support SSL.
7. The functions provided by web services support the use of XPath references.
8. When used with XForms, web services can be called through XForms submissions.

writeonly

Sets a field to be write only. This means that the user can type into the field, but cannot read what is typed. Instead, each character is replaced by a uniform symbol (such as an asterisk).

This is useful if you are creating a password field.

Syntax

```
<writeonly>setting</writeonly>
```

setting	*on*	the item is write only, which means that each character is replaced with a uniform symbol (such as an asterisk) to obscure the input.
	off	the item displays input normally.

Available In

field

Example

The following example shows a field that is set to be write only:

```
<field sid="writeOnlyField">
   <value>You cannot type into this field.</value>
   <readonly>on</readonly>
</field>
```

Usage Details

1. Default:
   - XFDL: **off**
   - XForms: the *xforms:secret* option, if present.
2. The *xforms:secret* option overrides the *writeonly* option, and forces the field to be write only.

<custom option>

Allows form designers to add application specific information to the form definition. This is useful when submitting forms to applications requiring non-XFDL information. An example of non-XFDL information might be an SQL query statement. Custom options must not be in the XFDL namespace.

Syntax

```
<option xmlns="http://www.ibm.com/xmlns/prod/XFDL/Custom">
   <!-- Arbitrary XML content -->
</option>
```

Example

This sample shows a custom option containing an SQL query.

```
<sql:query
   xmlns:sql="http://www.iso-standards.org/9075/2002/sqlxml">
   <!-- Content describing an SQL query -->
</sql:query>
```

This XML could be included in the definition of an item that triggers a form submission. Since the internal content can use XFDL computes to populate the query based on form content from the user, the server-side processing would be able to perform a proper query and have the results used to populate a response form.

Usage Details

1. The naming conventions for a custom option are as follows:
   - Custom options can have computed values by using the XFDL compute attribute, which must be qualified with a namespace prefix associated with the XFDL namespace URI.

- In order to make the XML content addressable by the XFDL compute system, the tag names of the custom option and any elements within it must conform to the XFDL syntax for a scope identifier.

Details on XForms Options

XForms options are only required when you are creating an XFDL form that contains an XForms data model. These options link the XFDL items to the data model, so that the items and model share data.

XForms options belong to the XForms namespace. However, in most respects these options are treated just like XFDL options. They are added to the syntax of the form at the same level, set particular characteristics for the containing item, and are recognized by the XFDL parser.

Despite this, XForms options do have some features that differ from XFDL options. This chapter describes these features and then details each of the XForms options separately.

XForms Namespace

XForms options exist in the XForms 1.0 namespace, which is defined as:

```
http://www.w3.org/2002/xforms
```

By convention, XFDL uses the *xforms* prefix for this namespace, which is normally declared on the <XFDL> element of the form as shown:

```
<XFDL xmlns:xforms="http://www.w3.org/2002/xforms">
```

Linking Input Items to the XForms Data Model

Items are linked to the XForms data model through the XForms options they contain. For example, a *field* item might use the *xforms:input* option to link it to a specific element in the data model.

For most input items, this link creates a relationship between the item's *value* option and the data element, so that they share information. For rich text fields, the link is between the data element and the *rtf* option, although the *plaintext* option can also be pushed to the data model by using an additional custom item.

Single Node Binding

When creating XForms options, most of the time you will also have to link that option to the XForms data model. For example, you might create a field in your form that uses the *xforms:input* option. This option links the field to a particular element or attribute in the instance data, so that the field and that element or attribute share data.

These links are created through a *single node binding*, which is expressed in one of two ways:

- As a *ref* attribute (with an optional *model* attribute).
- As a *bind* attribute.

Using the ref Attribute to Create a Single Node Binding

When you use the *ref* attribute to create a single node binding, you use an XPath reference to create a direct link between a display element in the form, such as a field, and a data element in the XForms model. The attribute is written as shown:

```
ref="XPath to data model element"
```

For example, you might have a field that includes the *xforms:input* option. This option uses the *ref* attribute to link the contents of the field to an element in the XForms model, as shown:

```
<xforms:input ref="XPath to data model element">
```

By default, XPath expressions begin at the root element of the first instance in the data model. For example, you might have the following data model:

```
<xforms:model>
   <xforms:instance xmlns="">
      <personnel>
         <name></name>
         <address></address>
         <telephone></telephone>
      <personnel>
   </xforms:instance>
<xforms:model>
```

In this case, the XPath expression would begin at the <personnel> element, since it is the root element of the default instance in the data model. Thus, to link the <xforms:input> element to the <name> element in the model, you would use the following expression:

```
<xforms:input ref="name">
```

Inheritance

single node bindings that use the *ref* attribute are evaluated relative to the nearest expressed single node binding. For example, consider the following XForms data model:

```
<xforms:model>
   <xforms:instance xmlns="">
      <root>
         <x>
            <y>5</y>
         </x>
      </root>
   </xforms:instance>
</xforms:model>
```

Within the body of your form, you may have an *xforms:group* that contains a field with an *xforms:input*. Both of these elements are bound to the data model, as shown:

```
<xforms:group ref="x">
   <field sid="Number">
      <xforms:input ref="y">
         <xforms:label>Number:</xforms:label>
      </xforms:input>
   </field>
</xforms:group>
```

In this case, the *xforms:group* is evaluated relative to the root node of the data model, which in this case is <root>. This links the group to the <x> element. The *xforms:input*, being a child of the *xforms:group* element, inherits its starting point from the *xforms:group* and is evaluated relative to the <x> element. This links the *xforms:input* to the <y> element.

Some XForms elements have optional single node bindings. In these cases, if the binding is not declared, then the children of that element inherit the same starting point as the element itself.

For example, consider the following *xforms:group*:

```
<xforms:group ref="x">
   <button sid="Submit">
      <xforms:trigger>
         <xforms:label ref="submittext"/>
      </xforms:trigger>
   </button>
</xforms:group>
```

In this case, the *xforms:trigger* element inherits a starting point of <x> from the *xforms:group*. Since the *xforms:trigger* does not declare a single node binding itself, the *xforms:label* also inherits a starting point of <x> from the *xforms:group*.

Absolute References

Absolute references are preceded with a slash, and begin with the root element of the data instance. For example, consider the following instance:

```
<xforms:model>
   <xforms:instance xmlns="">
      <root>
         <a>
            <b>5</b>
         </a>
         <c>
            <d>10</d>
         </c>
      </root>
   </xforms:instance>
</xforms:model>
```

To create an absolute reference to the <d> element, you would write:

```
/root/c/d
```

Additionally, absolute references override all inherance rules. For example, consider the following xforms:group element:

```
<xforms:group ref="a">
   <field sid="Number1">
      <xforms:input ref="b">
         <xforms:label>First Number:</xforms:label>
      </xforms:input>
   </field>
   <field sid="Number2">
      <xforms:input ref="/root/c/d">
         <xforms:label>Second Number:</xforms:label>
      </xforms:input>
   </field>
</xforms:group>
```

In this case, the reference for the first *xforms:input* is evaluated from the <a> element because it inherits this starting location from the *xforms:group*. However, the reference for the second *xforms:input* overrides that inheritance and links to the <d> element.

Multiple Models

If you are using multiple models in your form, all references will default to the first model. To refer to a different model, you must use a *model* attribute along with your *ref* attribute. The model attribute is written as shown:

```
model="model ID"
```

The model ID is determined by the *id* attribute on the *xforms:model* tag. For example, consider the following data model:

```
<xforms:models>
   <xforms:model>
      ...
   </xforms:model>
   <xforms:model id="m2">
      <xforms:instance xmlns="">
         <root>
            <x>5</x>
         </root>
      </xforms:instance>
   </xforms:model>
</xforms:models>
```

In this case, the first model is not assigned an ID, but the second model is. To create a reference to the x element in the second model, you must use both the *ref* and *model* attributes, as shown:

```
ref="x" model="m2"
```

When you switch to the non-default model in this way, the reference is evaluated from the root element of the first data instance in that model. In other words, inheritance from a previous single node binding to a different model is ignored.

Using the bind Attribute to Create a Single Node Binding

When you use the *bind* attribute to create a single node binding, you are creating an indirect link between a display element in the form, such as a field, to a data element in the XForms model. To do this, you link the element in the form to a <bind> element in the data model using that element's ID. This linking is then automatically extended to the data element that the <bind> affects.

For example, consider the following data model:

```
<xforms:model>
   <xforms:instance xmlns="">
      <root>
         <a/>
         <b/>
         <c/>
      </root>
   </xforms:instance>
   <xforms:bind id="hypotenuse" nodeset="c">
      <xforms:bind calculate="power(../a * ../a + ../b * ../b, 0.5)"/>
   </xforms:bind>
</xforms:model>
```

In this case, the <a> and <b> elements hold the length of the sides of a triangle. The <bind> element then calculates the hypotenuse and sets that value in the <c> element. You might link an *xforms:input* to this bind, as shown:

```
<field sid="Number">
   <xforms:input bind="hypotenuse">
      <xforms:label>Number:</xforms:label>
   </xforms:input>
</field>
```

In this case, the *xforms:input* is linked to the identified bind, which in turn link the *xforms:input* to the <c> element in the data instance. This means that the field and the <c> element would share data.

Nested Binds

When creating a single node binding, you cannot link to nested binds. You can only link to the outermost bind in any nested structure.

Inheritance

single node bindings that are created using the *bind* attribute do not inherit starting locations from other single node bindings (unlike single node bindings created using the *ref* attribute).

Nodeset Binding

A nodeset binding links an element in the form to a set of data elements. For example, *xforms:repeat* options always bind to a collection of rows in the data model, with each row representing a row in a table.

Nodeset bindings operate in all ways like single node bindings, except that they return a set of nodes rather than a single node. For more information about single node bindings, refer to "Single Node Binding" on page 205.

Bindings and Relevance

The relevance of display elements is determined based on their binding. For example, if a field in the form is bound to element <x> in the data model, then that field inherits its relevance from element <x>. If an item becomes non-relevant, then its *visible* and *active* options default to off.

The *xforms:group*, *xforms:switch*, and *xforms:repeat* options override the relevance of any display elements they contain.

For example, consider the following *xforms:group*:

```
<xforms:group ref="x">
   <field sid="Number1">
      <xforms:input ref="../y">
         <xforms:label>First Number:</xforms:label>
      </xforms:input>
   </field>
   <field sid="Number2">
      <xforms:input ref="../z">
         <xforms:label>Second Number:</xforms:label>
      </xforms:input>
   </field>
</xforms:group>
```

In this case, the relevance of the group as a whole is determined by the <x> element. This means that if <x> is not relevant, then neither of the fields are considered relevant regardless of that status of <y> or <z> (which may be relevant even if <x> is not because they are siblings of <x>).

Furthermore, if an *xforms:repeat* is bound to a nodeset that contains no relevant nodes, then the *visible* and *active* options of the containing table default to off.

This overriding behavior ensures that grouped items are always displayed or hidden as a group, rather than as individual items.

Metadata Sub-Options

Many of the XForms options support the inclusion of metadata sub-options. These sub-options are optional, and provide information that is passed to the user interface, including alerts and help messages. Valid sub-options include:

- Alert Setting
- Hint Setting
- Help Setting

Alert Setting

Sets an alert message that is displayed to the user if they enter invalid information. This is equivalent to the *message* setting in the *format* option. If both an *xforms:alert* and a *message* are provided for an item, then the *message* overrides the *xforms:alert*.

The alert setting follows this syntax:

```
<xforms:alert single_node_binding>alert text</xforms:alert>
```

single node binding	*string*	see “Single Node Binding” on page 205.
alert text	*string*	the alert message. If the single node binding is provided, then it overrides this message.

Hint Setting

Provides a help message that is displayed to the user if they enter help mode. This message is generally a short instruction, such as telling the user what format is valid for a specific field, and is displayed as a tooltip.

This is equivalent to the *help* option. If an item contains both an *xforms:hint* and a *help* option, then the *help* option overrides the *xforms:hint*.

The hint setting follows this syntax:

```
<xforms:hint single_node_binding>hint text</xforms:hint>
```

single node binding	*string*	see “Single Node Binding” on page 205.
hint text	*string*	the hint text. If the single node binding is provided, then it overrides this message.

Help Setting

Provides a help message that is displayed to the user if they enter help mode. This message is generally longer than an *xforms:hint*, and is intended to provide detailed help to the user.

Although there is no direct equivalent in XFDL, *xforms:help* is treated like the *help* option, and is displayed as a tooltip when the user enters help mode.

If an item contains both an *xforms:hint* and an *xforms:help*, then the help is appended to the hint. Futhermore, if an item contains both an *xforms:help* and a *help* option, then the *help* option overrides the *xforms:help*.

The help setting follows this syntax:

```
<xforms:help single_node_binding>help text</xforms:help>
```

single node binding	*string*	see "Single Node Binding" on page 205.
help text	*string*	the help text. If the single node binding is provided, then it overrides this message.

xformsmodels

Allows you to define one or more XForms data models in your form. Once defined, you can link the data in these models directly to the presentation layer of your form.

The <xformsmodels> tag can contain any number of XForms models, which in turn can contain any number of data instances. In general, you will use only one XForms model, since multiple models do not share data or user interface bindings.

Each data instance contains an XML element. This can be any valid XML with a single root, allowing you to re-use existing data models from other applications. Optionally, you can use the **src** attribute to load XForms instance data from an external source. In this case, the result is stored in the form once it is loaded.

Syntax

```
<xformsmodels>
   <xforms:model id="name1" schema="schema URIs"
      functions="functions">
      <xsd:schema>schema1</xsd:schema>
      ...
      <xsd:schema>scheman</xsd:schema>
      <xforms:instance id="name1" xmlns="namespace"
         src="source">
         instance1
      </xforms:instance>
      ...
      <xforms:instance id="namen" xmlns="namespace"
         src="source">
         instancen
      </xforms:instance>
      <xforms:bind1 property_settings>
         <xforms:bind1a property_settings>
            ...
         </xforms:bind>
         ...
      </xforms:bind>
      ...
      <xforms:bindn property_settings/>
      <xforms:submission1 submission_attributes/>
      ...
      <xforms:submissionn submission attributes/>
   </xforms:model>
   ...
   <xforms:model id="namen" schema="schema URIs"
      functions="functions">
      ...
   </xforms:model>
</xformsmodels>
```

Note:

- *there can be any number of xforms:model elements, each containing its own model.*
- <xforms:bind> elements can nest to any depth.

name	*string*	optional. An arbitrary name that you may assign to each model and data instance. The first model/instance is default, and does not require an ID. All subsequent models and instances do. All names assigned to id attributes must be globally unique within the form.
namespace	*string*	optional. The namespace of the instance data. All references to instance data must include an appropriate namespace prefix, unless you use the empty namespace (denoted by empty quotation marks, "").

schema URIs	*string*	optional. A space separated list of URIs that point to XML schemas. These schemas are constantly enforced within the XForms model. The file: scheme of referencing URIs is supported, with the following exceptions: • Cannot reference the Program Files directory. • Cannot reference the Windows® or Windows System directories. • Cannot reference any temporary directory. Additionally, the resulting file must be contained by a folder in a sub-tree of the folder that contains the originating file. To refer to a schema file in the Viewer's schema folder, use an *xsf* prefix. For example: `xsf:filename.xsd`

functions	*string*	optional. A space separated list functions used by the model. By default, all XForms 1.0 functions are supported. This allows you to add support for the following XForms 1.1 functions:
		`power`
		`current`

schema	*schema*	optional. An XML schema that is embedded in the form. These schemas are constantly enforced within the XForms model.

src	*URI*	optional. Use this to point to an external file that contains the instance data.
		
		This supports HTTP, HTTPS, and file URIs. Relative URIs are evaluated based on the location from which the form was obtained.
		
		When the document is retrieved from the Web, relative file locations are evaluated from the following folder:
		`Documents and Settings\<user>\`
		`   Application Data\PureEdge\xforms`

instance	*XML*	optional. An arbitrary XML data instance with a single root. This can be as simple or complex as you like, as long as it is valid XML. If the containing <xforms:instance> tag has an *src* attribute, this instance data is ignored and the loaded data is used instead.
property settings	*special*	sets properties for particular elements in the XForms model. See below for more information.
submission attributes	*(see below)*	

Property Settings

The property settings allow you to set specific properties for elements in your data model. They follow this syntax:

```
<xforms:bind id="name" nodeset="nodeset" property1 ... propertyn/>
```

Note:

- *xforms:bind elements never contain data.*
- an xforms:bind element can itself contain xforms:bind elements. Furthermore, the nodeset for the inner bind is evaluated relative to each node of the outer bind's nodeset.

name	*string*	optional. An arbitrary name you assign to the bind. All names assigned to id attributes must be globally unique within the form.
nodeset	*XPath*	an XPath expression defining which node or nodes in the data model are affected. This links the properties to one or more elements in a data instance.
property	*expression*	the property to apply to the indicated nodeset. Property attributes are expressed as follows: *property_name = setting* See below for more information.

Available Properties

The following list describes the properties you can set for a nodeset:

calculate

Applies a calculation that sets the content of a node. This calculation is written as an XPath expression that is evaluated relative to a node in the bind's nodeset.

For example, if you had a purchase order form, you might use the following expression to set the value of the ″total″ node to equal the value of the ″subtotal″ node plus the value of the ″tax″ node:

```
calculate="../subtotal + ../tax"
```

Do not link a data node with a calculation to a UI element that has a value set by a an XFDL compute. When the XForms UI binding transfers data to the UI element, the XFDL compute is destroyed.

constraint

Allows you to set a constraint for a node. For example, you could specify that the value of the node must be greater than one, or that it must not equal the value of a different node.

This property is set by any XPath expression that results in a boolean value. True means the constraint has been met, false means it has not.

For example, the following expression would ensure that the value for the upperPage node was always greater than or equal to the value of the lowerPage node:

```
nodeset="pagination/upperPage"
constraint=". &gt;= ../lowerPage"
```

If a constraint is set for a data node, and a different constraint is set for a linked UI element, then then the validity of the data is determined by considering both settings. If the data is invalid for either setting, the data will be considered invalid overall.

If a node is relevant (see below) and has a failed constraint, then an XForms submission is not permitted.

readonly
sets whether the associated node is readonly. If a UI element is linked to a readonly node, then the UI element will also be readonly. However, if the UI element has the *readonly* option set, it will override the setting for the data node.

This property is set by any XPath expression that results in a boolean value. True means the node is readonly, false means the node is not.

For example, the following expression would make the associated node readonly:

```
readonly="true()"
```

relevant
Determines whether a node is relevant. Non-relevant nodes are omitted from XForms submissions. Additionally, if a UI element is linked to a non-relevant node, then that UI element is not displayed to the user. However, if the UI element has either the *active* or *visible* options set, they will override the setting for the data node.

This property is set by any XPath expression that results in a boolean value. True means the node is relevant, false means the node is not.

For example, the following expression would make the node relevant if another node named “paymentType” had a value of “credit”:

```
nodeset="creditCardNumber"
relevant="../paymentType = 'credit'"
```

required
Sets whether the node requires input. If a UI element is linked to a required node, then the UI element inherits the setting. Furthermore, if the node does not require input, but a linked UI element does, then input is still required.

This property is set by any XPath expression that results in a boolean value. True means the node is required, false means the node is not.

For example, the following expression would set the node to require input:

```
required="true()"
```

If a relevant (see above) data note is required but empty, then an XForms submission in not permitted.

type Sets the data type of the node. The valid data types are defined by XML schema.

For example, the following expression sets the node to a date type:

```
type="xsd:date"
```

If the data type set for the node conflicts with the data type set for a linked UI element, then the validity of the data is determined by considering both settings. If the data is invalid for either setting, the data will be considered invalid overall.

For example, if you set a data node to be type int and you set a linked XFDL field to be type string, then typing in ″abc″ (a valid XFDL string) would still result in an error, since it does not match the int type.

If a relevant (see above) data node’s content does not match its type, then an XForms submission is not permitted.

Available Data Types

XML Schema defines a group of data types that can be used with the <xforms:model> tag. This includes the following commonly used types:
- xsd:boolean
- xsd:date
- xsd:double
- xsd:integer

For more information about the available data types, refer to the XML Schema specification as published by the W3C.

Submission Attributes

Submissions are defined by a set of attributes on an <xforms:submission> tag. This allows you to control the submission, setting where it goes, how it is formatted, and so on. The <xforms:submission> tag must be placed inside the <xforms:model> tag, but outside of the data instances contained in the model.

The <xforms:submission> tag follows this format:

```
<xforms:submission id="name" single_node_binding
   method="method" mediatype="media" action="URL
   includenamespaceprefixes="prefixes"
   replace="replace" instance="instanceID"
   xfdl:actionref="dynamicURL"/>
```

name	*string*	an arbitrary name you assign to the submission. All names assigned to id attributes must be globally unique within the form.
single node binding	*string*	sets the root node for the submission. This node and all of its children are submitted. The node must be part of the same model in which the *xforms:submission* appears. This binding is optional. If the submission does not contain a single node binding, then the default instance of the default data model is sent. For more information about single node bindings, refer to “Single Node Binding” on page 205.
method	*string*	determines the submission method used. Set to one of the following: • **put** — serializes the data in the XForms model as XML. No response is expected (this is generally used with a file URL). • **post** — serializes the data in the XForms model as XML. • **get** — serializes the data in the XForms model using URL encoding (x-www-form-urlencoded).

media	*MIME type*	*optional*. Overrides the default MIME type for the submission. If this is not set, the submission defaults to *application/xml*.
URL	*URL*	the URL to which the data is submitted. This must be a complete URL, and may use any of the following schemes: *http*, *https*, or *file*.
prefixes	*string*	a space separated list of namespace prefixes. The definitions for these namespaces are included with the root node when the data is submitted. If no prefixes are listed, all prefixes inherited by the root node are included. To include only the default namespace, use #default.
replace	*string*	determines how the return data is handled. Set to one of the following: • **all** — replace the entire form with the returned data. • **instance** — replaces the instance specified by the *instance* attribute. • **none** — ignore the returned data.
instanceID	*string*	*optional*. The ID of the instance to replace. If not provided, defaults to the instance that contained the submission data.
dynamicURL	*URL*	*optional*. the element in the model which will provide the URL to which the data is submitted. The URL in this element must be complete, and may use any of the following schemes: *http*, *https*, or *file*. If used, the URL found in the specified element is default. If no URL is found, then the URL provided by the **action** attribute is used.

Available In

form global

Example

The following example creates a small XForms data model that contains the name, age, and birthdate of a person. The model includes binds that set the age to be an integer value and set the birthdate to be a date value, as well as an submission that submits the entire instance.

```
<?xml version="1.0"?>
<XFDL xmlns="http://www.ibm.com/xmlns/prod/XFDL/7.0"
   xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
   xmlns:xforms="http://www.w3.org/2002/xforms">
   <globalpage sid="global">
      <global sid="global">
         <xformsmodels xmlns="">
            <xforms:model>
               <xforms:instance>
                  <testmodel>
                     <name></name>
                     <age></age>
                     <birthdate></birthdate>
                  </testmodel>
               </xforms:instance>
               <xforms:bind nodeset="age" type="xsd:integer"/>
               <xforms:bind nodeset="birthdate" type="xsd:date"/>
               <xforms:submission id="submitTest" method="post"
                  action="http://www.testserver.com/cgi-bin/testscript"
                  includenamespaceprefixes=""/>
            </xforms:model>
         </xformsmodels>
      </global>
   </globalpage>
```

Usage Details

1. When using an XForms model, you must link the model to the presentation layer of the form. You do this by adding other XForms options to the form, and then using single node and nodeset bindings to create these links.
2. If you are including an XForms model in your form, you must declare the XForms namespace. In general, you should declare this on the XFDL node, as shown:
   ```
   <XFDL xmlns="http://www.ibm.com/xmlns/prod/XFDL/7.0"
      xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
      xmlns:xforms="http://www.w3.org/2002/xforms">
   ```
3. When writing the XPath expression for a constraint, you must express the less than and greater than symbols as character references (&lt; and &gt; respectively) or they will be confused with the opening and closing symbols for the tag.
4. Each data model can have any number of *xforms:submissions*. This is useful if you need to submit different data at given times, or if you need to submit the same data to different servers.
5. There are two ways to trigger a submission:
   - Create a submission button using the *xforms:submit* option. For more information, refer to "xforms:submit" on page 236.
   - Create a submission button using an *xforms:trigger* and one or more *xforms:send* actions. For more information, refer to ″Details on XForms Actions″.
6. The <xforms:model> portion of the *xformsmodels* option can trigger XForms actions using the *xforms-model-construct*, *xforms-model-construct-done*, *xforms-model-destruct*, and *xforms-ready* events. For more information, refer to "Details on XForms Event Handlers" on page 261.
7. The <xforms:submission> portion of the *xformsmodels* option can trigger XForms actions using the *xforms-submit*, *xforms-submit-done*, and *xforms-submit-error* events. For more information, refer to "Details on XForms Event Handlers" on page 261.
8. Action handlers for *xforms-submit-done* and *xforms-submit-error* are supported only for *replace=instance* and *replace=none* type submissions.
9. For SOAP submissions, use a method of **post** and the following MIME type in the *mediatype* attribute:
   ```
   application/soap+xml; action=Some Action;[charset=x]
   ```
   Where you provide the name of the action and optionally supply a character set.

   If the root element of the submitted data is in the SOAP 1.1 namespace (http://schemas.xmlsoap.org/soap/envelope/) then the content-type used is text/xml rather than the given mediatype, the *charset* parameter is preserved, and the SOAP Action header is added with a value of *Some Action*.
10. If an element in the XForms data model is both empty and invalid, then any item on the form that is bound to that element is set to be mandatory.
11. You can create an XForms version of Smartfill by creating submissions that save and load instance data to a file on the user's computer. To create a "save user data" submission, you must include a submission id, an action that

points to the xml file that will contain the data, a “put” method to place the data in the file, a reference to the instance containing the data, and an instance replace of none. For example:

```
<xforms:submission id="saveName" action="file:savedata1.xml" method="put"
   ref="name" replace="none"></xforms:submission>
```

To create a ″load user data″ submission, you must include a submission id, an action that points to the xml file that contains the user data, a “get” method to load the data into the data model, a reference to the instance that will contain the user data, and replace the entire instance. Furthermore, to ensure that the new data is displayed in the form, you must create an *xforms:setvalue* action. For example:

```
<xforms:submission id="loadName" action="file:savedata1.xml" instance="name"
   method="get" replace="instance">
   <xforms:setvalue ref="name" value="instance('name')"
      ev:event="xforms-submit-done"/>
</xforms:submission>
```

xforms:group

This option groups the items that are contained within a *pane* item.

Syntax

```
<xforms:group single_node_binding>
   ...items in group...
</xforms:group>
```

single node binding	*string*	optional. See “Single Node Binding” on page 205.

Available In

pane

Example

The following example shows a pane item with a group of fields that store the user’s address:

```
<pane sid="address">
   <xforms:group ref="customerData/address">
      <field sid="Street">
         <xforms:input ref="street">
            <xforms:label>Street:</xforms:label>
         </xforms:input>
      </field>
      <field sid="City">
         <xforms:input ref="city">
            <xforms:label>City:</xforms:label>
         </xforms:input>
      </field>
      <field sid="Country">
         <xforms:input ref="country">
            <xforms:label>Country:</xforms:label>
         </xforms:input>
      </field>
   </xforms:group>
</pane>
```

Usage Details

1. If the group has a single node binding, and it resolves to an empty nodeset or a non-relevant node, then the *xforms:group* provides a default of false to the pane item's *visible* option. However, if it resolve to a relevant node, then the default visibility is true.

xforms:input

This option links a field, combobox or check box to an element in the data model so that they share data. However, the xforms:input only support a single line of data. For example, if you added an *xforms:input* option to a field in your form, you could use that option to link the field to a name element in your data model. Once linked, any changes made to the data in one would be reflected by the other.

This option is only available if you are using an XForms data model.

Syntax

```
<xforms:input single_node_binding>
   <xforms:label>label text</xforms:label>
   Alert Setting
   Hint Setting
   Help Setting
</xforms:input>
```

single node binding	*string*	see "Single Node Binding" on page 205.
label text	*string*	sets the text for the item's built-in label, as well as the accessibility message for the item. Although the *xforms:label* tag must appear, you can use an empty string if you do not want to set the label. If the item also has a *label* or *acclabel* option, they will override this setting.
Alert, Hint, Help Setting	*metadata*	see "Metadata Sub-Options" on page 210.

Available In

check, combobox, field, custom

Example

The following code shows an XForms model that contains a name, age, and birthdate element:

```
<xformsmodels>
   <xforms:model>
      <xforms:instance id="test">
         <testmodel>
            <name></name>
            <age></age>
            <birthdate></birthdate>
         </testmodel>
      </xforms:instance>
   </xforms:model>
</xformsmodels>
```

Using that data model, the following code links a field to the name element in the data model, so that they share data:

```
<field sid="nameField">
   <xforms:input ref="name">
      <xforms:label>Name:</xforms:label>
   </xforms:input>
</field>
```

Once you have a basic field, you can add a help message to it. In the following example, the <xforms:hint> element is used to provide some simple help for the user:

```
<field sid="nameField">
   <xforms:input ref="name">
      <xforms:label>Name:</xforms:label>
      <xforms:hint>Enter your full name.</xforms:hint>
   </xforms:input>
</field>
```

You can also add an alert, in case the user enters the wrong type of data. For instance, the following example creates a field that links to the age node in the data model above and has a data type of integer. Additionally, there is an alert that will appear if the user tries to enter the wrong data type:

```
<field sid="ageField">
   <xforms:input ref="age">
      <xforms:label>Age:</xforms:label>
      <xforms:alert>You must enter an integer value.</xforms:alert>
   <xforms:input>
   <format>
      <datatype>integer</datatype>
   </format>
</field>
```

To get the user's birthdate, you can use a combobox instead of a field. In this case, we will create a combobox with no cells, and set the *value* option using the *date* function. This causes the combobox to display a date picker widget when opened.

```
<combobox sid="birthdate">
   <xforms:input ref="birthdate">
      <xforms:label>Birthdate:</xforms:label>
   </xforms:input>
   <format>
      <datatype>date</datatype>
   </format>
   <group>date</group>
</combobox>
```

Usage Details

1. This option limits a field to a single line of input. To create a field that accepts more input, use the *xforms:textarea* option instead.
2. When using *xforms:input* with a *check* box item, the **on** or **off** values of the check box are translated into a boolean **true** or **false** (xsd:boolean) when copied to the data model.
3. When using *xforms:input* with a *combobox* item, the combobox must have an XFDL data type of **date**.

4. Any item with an *xforms:input* option can trigger XForms actions using the *xforms-value-changed* event. For more information, refer to "Details on XForms Event Handlers" on page 261.
5. When you add an *xforms:input* to a custom item, it works just as though it was placed in a valid XFDL item. This means that a value option is created and is linked to the data model.
6. Pressign the ENTER key in a single line field will commit the value that has been typed to the form. This means it will be copied to the XForms model.

xforms:output

Links a button or a label to information in the XForms data model. This allows you to display images or text from the data model on the face of a button or label.

If you use this to link to an image in the data model, that image must be base64 encoded.

Syntax

```
<xforms:output single_node_binding value="XPath" mediatype="MIME type">
   <xforms:label>label text</xforms:label>
   Alert Setting
   Hint Setting
   Help Setting
</xforms:output>
```

single node binding	*string*	*optional*. See "Single Node Binding" on page 205.
XPath	*XPath*	*optional*. An XPath reference that sets the value for the output. This is useful for performing simple calculations on one or more data elements. For example, you might add two data elements together. The XPath reference is evaluated relative to the root node of the default instance. If a single node binding is provided, this attribute is ignored.
mediatype	*MIME type*	if the XPath parameter refers to image data, you must include this parameter to specify the MIME type of the image. Note that only image MIME types are valid.
label text	*string*	sets the text for the value option of the button or label, as well as the accessibility message for the item. Although the *xforms:label* tag must appear, you can use an empty string if you do not want to set the label. If the item also has an *acclabel* option, it will override this setting.
Alert, Hint, Help Setting	*metadata*	see "Metadata Sub-Options" on page 210.

Available In

label

(Note that *xforms:output* cannot be the immediate child of a button; however, it can be a descendent through the *xforms:trigger*, *xforms:submit*, or *xforms:upload* options.)

Example

The following example shows a label that uses the *xforms:output* option to link it to a total that is calculated in the data model. The label will then use this data as its value, and display it to the user.

```
<label sid="totalCost">
   <xforms:output ref="total"/>
</label>
```

Usage Details

1. Images in the data model must be base64 encoded before they can be linked with the *xforms:output* option.
2. The *xforms:label* is optional in an *xforms:output*. If provided, the value of the *xforms:label* is prepended to the text from the single node binding of the *xforms:output* (if it has one). The concatenated value is then displayed by the XFDL *label* (unless the label displays an image).
3. Right justification affects only the text provided by the *xforms:output*. Any text provided by an *xforms:label* is left justified.
4. The text provided by *xforms:label* does not affect the XFDL value option. Only data obtained by the signe node binding to the *xforms:output* is placed in the XFDL value. This means that the format option settings are applied to the *xforms:output* text, but not the *xforms:label* text. This allows you to create a leading label for other values, such as currency types.
5. Although *xforms:output* cannot be the immediate child of an XFDL button item, it can appear as a child that is several times removed through the *xforms:trigger*, *xforms:submit*, or *xforms:upload* options.
6. The *value* attribute can reference data in a non-default model if you wrap the *xforms:output* in an *xforms:group* and bind the group to the data model you want to reference. For example, the following sample shows how to add the values of 2 nodes in the non-default model:

```
<pane sid="ModelChooser">
   <xforms:group model="x" ref="/data">
      <label sid="Sum"
         <xforms:output value="a + b"/>
      </label>
   </xforms:group>
</pane>
```

xforms:range

Sets the range of values a user can select with a *slider* item.

Syntax

```
<xforms:range single node binding start="start" end="end"
   step="step">
   <xforms:label>label text</xforms:label>
</xforms:range>
```

single node binding	*string*	see “Single Node Binding” on page 205.

start	*integer or float*	the starting value for the range.
end	*integer or float*	the end value for the range.
step	*integer or float*	how much each increment in the range increases the total. For example, a step of one counts by ones (1, 2, 3...) while a step of two counts by twos (1, 3, 5...).

Available In

slider

Example

The following example shows and slider that allows the user to select any number between 1 and 10:

```
<slider sid="rating">
   <xforms:range ref="rating" start="1" end="2" step="1">
      <xforms:label>Rate this form on a scale of 1 to 10</xforms:label>
   </xforms:range>
</slider>
```

Usage Details

1. Any item with an *xforms:range* option can trigger XForms actions using the *xforms-value-changed* event. For more information, refer to “Details on XForms Event Handlers” on page 261.

xforms:repeat

Creates a template row of items for a table. These items are then duplicated for each row the user adds to the table, and for each row of data that exists in the XForms data model.

Syntax

```
<xforms:repeat id="name" nodeset_binding
   startindex="index">
   ...XFDL items...
</xforms:repeat>
```

name	*string*	an arbitrary name that you assign to the table. All names assigned to id attributes must be globally unique within the form.
nodeset binding	*string*	See “Nodeset Binding” on page 209.
index	*integer*	This determines which row of the repeat receives the focus initially. Default: **1**.
XFDL Items	*XFDL*	the XFDL items that should appear in each row of the table. This can include nested grouping items, such as *table* and *pane*, as well as non-XForms items, such as *line*.

Available In

table

Example

The following example shows a table that uses an *xforms:repeat* to create a row of data that you might find in a purchase order. This row contains: a popup that lets the user select which item to purchase, a field that lets them enter a quantity for the item, and a label that displays the cost of the item.

```
<table sid="itemsTable">
   <xforms:repeat nodeset="order/row">
      <popup sid="Product">
         <xforms:select1 appearance="minimal" ref="product">
            <xforms:label>Choose product</xforms:label>
            <xforms:item>
               <xforms:label>Widget</xforms:label>
               <xforms:value>widget</xforms:value>
            </xforms:item>
            <xforms:item>
               <xforms:label>Gadget</xforms:label>
               <xforms:value>gadget</xforms:value>
            </xforms:item>
         </xforms:select1>
      </popup>
      <field sid="Qty">
         <xforms:input ref="qty">
            <xforms:label>Qty:</xforms:label>
         </xforms:input>
      </field>
      <label sid="LineTotal">
         <xforms:output ref="lineTotal"/>
      </label>
   </xforms:repeat>
</table>
```

Usage Details

1. To add or remove rows from a table, you must use the *xforms:insert* and *xforms:delete* actions respectively. For more information about these actions, refer to "Details on XForms Actions" on page 245.
2. An *xforms:repeat* maintains an internal index that indicates which row has the focus. When the focus is sent to the table, it automatically goes to the indexed row. The index is one-based, so that the first row has an index of 1, the second row an index of 2, and so on. You can change the index by using the *xforms:setindex* action.
3. When using computes with a repeat, the following rules apply:
   - Computes written within a row may not reference elements in a different row.
   - Computes written outside a row may not reference elements within a row.
4. Due to a limitation in XForms 1.0, *xforms:repeat* cannot contain an *xforms:switch*.
5. If the nodeset binding of the *xforms:repeat* is empty or contains non-relevant nodes, then the *xforms:repeat* provides a default of **false** to the table's *visible* option.

xforms:secret

Links a field to an element in the XForms data model, and makes the field write only.

This option is only available if you are using an XForms data model. If you are not using an XForms model, use the *writeonly* option to create write only fields.

Syntax

```
<xforms:secret single_node_binding>
   <xforms:label>label text</xforms:label>
   Alert Setting
   Hint Setting
   Help Setting
</xforms:secret>
```

single node binding	*string*	see "Single Node Binding" on page 205.
label text	*string*	sets the text for the field's built-in label, as well as the accessibility message for the field. Leave this setting empty for no label. If the field also has either a *label* option or an *acclabel* option, those settings will override the *xforms:label*.
Alert, Hint, Help Setting	*metadata*	see "Metadata Sub-Options" on page 210.

Available In

field

Example

The following code shows an XForms model that contains a password element:

```
<xformsmodels>
   <xforms:model>
      <xforms:instance id="test">
         <testmodel>
            <name></name>
            <age></age>
            <birthdate></birthdate>
            <password></password>
         </testmodel>
      </xforms:instance>
   </xforms:model>
</xformsmodels>
```

Using that data model, the following code creates a write only field that links to the password element:

```
<field sid="passwordField">
   <xforms:secret ref="password">
      <xforms:label>Password:</xforms:label>
   </xforms:secret>
</field>
```

Usage Details

1. The *xforms:secret* must contain an <xforms:label> tag. Furthermore, any text in the <xforms:label> tag will override the *label* option. If you do not want a label, leave the <xforms:label> empty.
2. This setting overrides the *writeonly* option.
3. If you link a secret field to a readonly data element, the user will not be able to see or change the information in the field.
4. Any item with an *xforms:secret* option can trigger XForms actions using the *xforms-value-changed* event. For more information, refer to "Details on XForms Event Handlers" on page 261.

---

xforms:select

Sets the choices that are displayed by a *checkgroup* or *list* when the user can select one or more of the choices. When the form is processed, an individual *check* or *cell* item is automatically generated to represent each choice.

Syntax

The *xforms:select* option has two different syntaxes, depending on whether the choices are included in the option itself, or whether the choices are included in the data model and linked by the option.

If you want to include the choices in the option itself, use the following syntax:

```
<xforms:select single_node_binding appearance="style">
   <xforms:label>label text</xforms:label>
   <xforms:item1>
      <xforms:label>label for choice</xforms:label>
      <xforms:value>value for choice</xforms:value>
      <xforms:extension>XFDL Options</xforms:extension>
   </xforms:item1>
   ...
   <xforms:itemn>
      ...
   <xforms:itemn>
   Alert Setting
   Hint Setting
   Help Setting
</xforms:select>
```

Note:

- *there can be any number of xforms:item elements.*

single node binding	*string*	see "Single Node Binding" on page 205.
style	*string*	sets to one of the following values: • full for a *checkgroup* • compact for a *list* item Default: **compact**.

label text	*string*	sets the text for a label that is displayed at the top of the *checkgroup*. Leave this blank to display no label. If the item also has a *label* option, it will override this setting.
label for choice	*string*	sets the text that is displayed for the choice.
value for choice	*string*	sets the value that is stored if the user selects this choice.
XFDL Options	*XFDL options*	adds specific XFDL options to the item represented by the choice. For example, you might want to add a *type* option to the choices in a list, so that the cells that are generated by those choices trigger actions.
Hint, Help Setting	*metadata*	see "Metadata Sub-Options" on page 210.

If you want to include the choices in the data model, use the following syntax:

```
<xforms:select single_node_binding appearance="full">
   <xforms:label>label text</xforms:label>
   <xforms:itemset nodeset="XPath to choices">
      <xforms:label ref="XPath to label text"/>
      <xforms:value ref="."></xforms:value>
      <xforms:extension>XFDL Options</xforms:extension>
   </xforms:itemset>
   Hint Setting
   Help Setting
   Alert Setting
</xforms:select>
```

single node binding	*string*	see "Single Node Binding" on page 205.
label text	*string*	sets the text for a label that is displayed at the top of the *checkgroup*. Leave this blank to display no label.

XPath to choices	*string*	an XPath reference to the elements in the data model that provide the choices. This defines the set of item's that the *checkgroup* or *list* displays as choices. This reference is relative to the *XPath to element* reference.
		
		For example, your data model may contain the following elements that represent the choices for your item:
		`<choice show="US Dollars">USD</choice>`
		`<choice show="CDN Dollars">CDN</choice>`
		`<choice show="Euros">Euro</choice>`
		
		In this case, you would reference those choices as:
		`choice`

XPath to label text	*string*	an XPath reference to the label text for each choice. This text is included in your data model as attributes on the data elements that contain your choices. This reference is relative to the *XPath to choices* reference.
		
		For example, your data model may contain the following elements that represent the choices for your *checkgroup*:
		`<choice show="US Dollars">USD</choice>`
		`<choice show="CDN Dollars">CDN</choice>`
		`<choice show="Euros">Euro</choice>`
		
		In this case, your would reference the show attributes that contain the text that describes that element, as shown:
		`@show`

XFDL Options	*XFDL options*	adds specific XFDL options to all items represented by the itemset. For example, if you are creating a *checkgroup*, you might want to set the *itemlocation* for all items in the set, so that they are spaced horizontally rather than vertically.
Hint, Help Setting	*metadata*	see "Metadata Sub-Options" on page 210.

Available In

checkgroup, list

Example

The following code creates a *checkgroup* with three choices: US Dollars, CDN Dollars, and Euro. The choices themselves are defined within the *xforms:select* option.

```
<checkgroup sid="currency">
   <xforms:select ref="currency" appearance="full">
      <xforms:label
         >Select the currencies you accept:</xforms:label>
      <xforms:item>
         <xforms:label>US Dollars</xforms:label>
         <xforms:value>USD</xforms:value>
      </xforms:item>
      <xforms:item>
         <xforms:label>CDN Dollars</xforms:label>
         <xforms:value>CDN</xforms:value>
      </xforms:item>
      <xforms:item>
         <xforms:label>Euro</xforms:label>
         <xforms:value>Euro</xforms:value>
      </xforms:item>
   </xforms:select>
</checkgroup>
```

Alternatively, you could create the choices in your data model as follows:

```
<xforms:instance xmlns="" id="currency">
   <data>
      <currency/>
      <choice show="US Dollars">USD</choice>
      <choice show="CDN Dollars">CDN</choice>
      <choice show="Euros">Euro</choice>
   </data>
</xforms:instance>
```

In this case, you would use the *xforms:select* option to link to those choices, as illustrated by the following checkgroup:

```
<checkgroup sid="currency">
   <xforms:select ref="currency" appearance="full">
      <xforms:label
         >Select the currencies you accept:</xforms:label>
      <xforms:itemset nodeset="instance('currency')/choice">
         <xforms:label ref="@show"></xforms:label>
         <xforms:value ref="."></xforms:value>
      </xforms:itemset>
   </xforms:select>
</checkgroup>
```

Usage Details

1. The single node binding for the *xforms:select* must refer to the same model as the single node bindings for the choices within the *xforms:select*.
2. To create a *checkgroup* or *list* item from which the user can select only one choice, use the *xforms:select1* option.
3. The choices available in a *list* are equivalent to cells of type select. If you want a choice to perform a different action use the *xforms:extension* to set a different type. For example, the following *xforms:item* is set to type link:

   ```
   <xforms:item>
      <xforms:label>US Dollars</xforms:label>
      <xforms:value>USD</xforms:value>
      <xforms:extension>
         <type>link</type>
         <url>http://www.ibm.myserver.com/mypage.htm</url>
      </xforms:extension>
   </xforms:item>
   ```
4. If your *xforms:select* contains both an *xforms:itemset* and one or more *xforms:item* elements, the *xforms:itemset* is used and the individual *xforms:item* elements are ignored.
5. The single node binding in the *xforms:select* option creates a link between the *value* option for the containing item and the bound element in the data model, so that they share data. When the user makes a selection, the *xforms:value* of that selection is stored in the XFDL *value* of the item containing the *xforms:select* and in the data node bound to the *xforms:select*.
6. If the user makes multiple selections, those choices are stored as a space delimited list. The selected items are listed by their build order in the form. Because this list is space delimited, the choices themselves cannot contain spaces.
7. To store the value of each selection in its own data element, use the *xforms:select* and *xforms:deselect* events. For more information, refer to "Details on XForms Actions" on page 245.
8. The item set (determined by the *xforms:item* elements or the *xforms:itemset* element) may be empty if the bound nodes contain no data, or if the bound nodes are not relevant. In either case, the containing item is displayed without any choices. For example, a *checkgroup* would be displayed without any checks in it.
9. *Checkgroup* and *radiogroup* items are arranged vertically by default (that is, each choice appears immediately below the previous choice). To arrange *checkgroup* or *radiogroup* items in another manner, use the *xforms:extension* to add an *itemlocation* to each item in the group. For example, you might set the items to appear one after another horizontally with the following *itemlocation*:

```
<itemlocation>
   <after compute="itemprevious"/>
</itemlocation>
```

10. To set the choices in a *list* to perform particular actions, such as a save or submit, use the *xforms:extension* to add a *type* option to each <xforms:item> element. For example, you might create the following items in your list:

```
<xforms:item>
   <xforms:label>Save</xforms:label>
   <xforms:value>Save</xforms:value>
   <xforms:extension>
      <type>saveas</type>
   </xforms:extension>
</xforms:item>
<xforms:item>
   <xforms:label>Submit</xforms:label>
   <xforms:value>Submit</xforms:value>
   <xforms:extension>
      <type>submit</type>
   </xforms:extension>
</xforms:item>
```

11. The *value* option overrides the *xforms:value* of the listed choices. This is useful when working with *list* items. A *list* displays the *xforms:label* of a choice until it is selected, at which point it displays the *xforms:value* of a choice. This means that you can set the *value* option to override the default *xforms:value*, which will change what is displayed when a choice is selected.

    For example, you might have a data model that sets a full text label for the choices, but uses an abbreviated value. The following instance shows this, using the show attribute to set the *xforms:label* and the contents of each element to set the *xforms:value*:

```
<xforms:instance id="currency">
   <data>
      <choice show="US Dollars">USD</choice>
      <choice show="CDN Dollars">CDN</choice>
      <choice show="Euros">Euro</choice>
   </data>
</xforms:instance>
```

    In this case, you might want to display the full text label even after a choice is selected. You can do this by using *xforms:extension* to add a *value* to each item in the set. You can then set the *value* to compute its contents to equal the *label*, which is populated with the full text description. The following sample shows this:

```
<checkgroup sid="currency">
   <xforms:select ref="currency" appearance="full">
      <xforms:label
         >Select the currencies you accept:</xforms:label>
      <xforms:itemset nodeset="instance('currency')/choice">
         <xforms:label ref="@show"></xforms:label>
         <xforms:value ref="."></xforms:value>
         <xforms:extension>
            <value compute="label"/>
         </xforms:extension>
      </xforms:itemset>
   </xforms:select>
</checkgroup>
```

12. Any item with an *xforms:select* option can trigger XForms actions using the *xforms-value-changed*, *xforms-select*, or *xforms-deselect* events. For more information, refer to “Details on XForms Event Handlers” on page 261.

xforms:select1

Sets the choices that are displayed by a *checkgroup, radiogroup, list, popup,* or *combobox.* The *xforms:select1* option limits users to selecting one of the choices.

Syntax

The *xforms:select1* option has two different syntaxes, depending on whether the choices are included in the option itself, or whether the choices are included in the data model and linked by the option.

If you want to include the choices in the option itself, use the following syntax:

```
<xforms:select1 single_node_binding appearance="style"
   selection="type">
   <xforms:label>label text</xforms:label>
   <xforms:item1>
      <xforms:label>label for choice</xforms:label>
      <xforms:value>value for choice</xforms:value>
   </xforms:item1>
   ...
   <xforms:itemn>
      ...
   <xforms:itemn>
   Alert Setting
   Hint Setting
   Help Setting
</xforms:select>
```

Note:

- *there can be any number of xforms:item elements.*

single node binding	*string*	see "Single Node Binding" on page 205.
style	*string*	set to one of the following values: • **full** — if you are setting choices for a *radiogroup* of *checkgroup*. • **compact** — if you are setting choices for a *list*. • **minimal** — if you are setting choices for a *popup* or *combobox*. Default: **minimal**.
type	*string*	set one of the following values: • **open** — set to open if the item is a *combobox*. • **closed** — set to closed if the item is not a *combobox*. This parameter defaults to closed, and is not required if the item is not a *combobox*. Default: **closed**.
label text	*string*	sets the text for a label that is displayed at the top of the item. Leave this blank to display no label. If the item also has a *label* option, it will override this setting.

label for choice	*string*	sets the text that is displayed for the choice.
value for choice	*string*	sets the value that is stored if the user selects this choice.
Alert, Hint, Help Setting	*metadata*	see “Metadata Sub-Options” on page 210.

If you want to include the choices in the data model, use the following syntax:

```
<xforms:select1 single_node_binding appearance="type"
   selection="input">
   <xforms:label>label text</xforms:label>
   <xforms:itemset nodeset="XPath to choices">
      <xforms:label ref="XPath to label text"
         ></xforms:label>
      <xforms:value ref="."></xforms:value>
   </xforms:itemset>
   Alert Setting
   Hint Setting
   Help Setting
</xforms:select>
```

single node binding	*string*	see “Single Node Binding” on page 205.
type	*string*	set to one of the following values: • **full** — if you are setting choices for a *radiogroup* of *checkgroup*. • **compact** — if you are setting choices for a *list*. • **minimal** — if you are setting choices for a *popup* or *combobox*.
input	*string*	set one of the following values: • **open** — set to open if the item is a *combobox*. • **closed** — set to closed if the item is not a *combobox*. This parameter defaults to closed, and is not required if the item is not a *combobox*.
label text	*string*	sets the text for a label that is displayed at the top of the item. Leave this blank to display no label.

XPath to choices	*string*	an XPath reference to the elements in the data model that provide the choices. This defines the set of item s that the item displays as choices. This reference is relative to the *XPath to element* reference.
		
		For example, your data model may contain the following elements that represent the choices for your item:
		`<choice show="US Dollars">USD</choice>`
		`<choice show="CDN Dollars">CDN</choice>`
		`<choice show="Euros">Euro</choice>`
		
		In this case, your would reference those choices as:
		`choice`

XPath to label text	*string*	an XPath reference to the label text for each choice. This text is included in your data model as attributes on the data elements that contain your choices. This reference is relative to the *XPath to choices* reference.
		
		For example, your data model may contain the following elements that represent the choices for your *checkgroup*:
		`<choice show="US Dollars">USD</choice>`
		`<choice show="CDN Dollars">CDN</choice>`
		`<choice show="Euros">Euro</choice>`
		
		In this case, your would reference the show attributes that contain the text that describes that element, as shown:
		`@show`

Alert, Hint, Help Setting	*metadata*	see "Metadata Sub-Options" on page 210.

Available In

checkgroup, combobox, list, popup, radiogroup

Example

The following code creates a *popup* with three choices: US Dollars, CDN Dollars, and Euro. The choices themselves are defined by the *xforms:select1* option.

```
<popup sid="currencyType">
   <xforms:select1 ref="selectedCurrency" appearance="minimal">
      <xforms:label>Select the payment currency:</xforms:label>
      <xforms:item>
         <xforms:label>US Dollars</xforms:label>
         <xforms:value>USD</xforms:value>
      </xforms:item>
      <xforms:item>
         <xforms:label>CDN Dollars</xforms:label>
         <xforms:value>CDN</xforms:value>
      </xforms:item>
      <xforms:item>
         <xforms:label>Euro</xforms:label>
         <xforms:value>Euro</xforms:value>
      </xforms:item>
   </xforms:select1>
</popup>
```

Alternatively, you could create the choices in your data model as follows:

```
<xforms:instance id="currency" xmlns="">
   <data>
  <choice show="US Dollars">USD</choice>
      <choice show="CDN Dollars">CDN</choice>
      <choice show="Euros">Euro</choice>
   </data>
</xforms:instance>
```

In this case, you would use the *xforms:select* option to link to those choices, as illustrated by the following checkgroup:

```
<popup sid="currencyType">
   <xforms:select1 ref="selectedCurrency" appearance="minimal">
      <xforms:label>Select your preferred currency for payment:</
         xforms:label>
      <xforms:itemset nodeset="instance('currency')/choice">
         <xforms:label ref="@show"></xforms:label>
         <xforms:value ref="."></xforms:value>
      </xforms:itemset>
   </xforms:select1>
</popup>
```

Usage Details

1. The single node binding for the *xforms:select1* must refer to the same model as the single node bindings for the choices within the *xforms:select1*.
2. To create a *checkgroup* or *list* item from which the user can select any number of choices, use the *xforms:select* option.
3. The choices available in a *list* are equivalent to cells of type **select**. If you want a choice to perform a different action use the *xforms:extension* to set a different type. For example, the following *xforms:item* is set to type **link**:
   ```
   <xforms:item>
      <xforms:label>US Dollars</xforms:label>
      <xforms:value>USD</xforms:value>
      <xforms:extension>
         <type>link</type>
         <url>http://www.ibm.myserver.com/mypage.htm</url>
      </xforms:extension>
   </xforms:item>
   ```
4. If your *xforms:select1* contains both an *xforms:itemset* and one or more *xforms:item* elements, the *xforms:itemset* is used and the individual *xforms:item* elements are ignored.
5. The single node binding in the *xforms:select1* option creates a link between the *value* option for the containing item and the bound element in the data model, so that they share data. When the user makes a selection, the *xforms:value* of that selection is stored in the XFDL *value* of the item containing the *xforms:select1* and in the data node bound to the *xforms:select1*.
6. The item set (determined by the *xforms:item* elements or the *xforms:itemset* element) may be empty if the bound nodes contain no data, or if the bound nodes are not relevant. In either case, the containing item is displayed without any choices. For example, a *checkgroup* would be displayed without any checks in it.
7. *Checkgroup* and *radiogroup* items are arranged vertically by default (that is, each choice appears immediately below the previous choice). To arrange *checkgroup* or *radiogroup* items in another manner, use the *xforms:extension* to add an *itemlocation* to each item in the group. For example, you might set the items to appear one after another horizontally with the following *itemlocation*:
   ```
   <itemlocation>
      <after compute="itemprevious"/>
   </itemlocation>
   ```
8. To set the choices in a *combobox*, *list*, or *popup* to perform particular actions, such as a save or submit, you can use the *xforms:extension* to add a *type* option to each <xforms:item> element. For example, you might create the following items in:
   ```
   <xforms:item>
      <xforms:label>Save</xforms:label>
      <xforms:value>Save</xforms:value>
      <xforms:extension>
         <type>saveas</type>
      </xforms:extension>
   </xforms:item>
   <xforms:item>
      <xforms:label>Submit</xforms:label>
      <xforms:value>Submit</xforms:value>
   ```

```
      <xforms:extension>
         <type>submit</type>
      </xforms:extension>
   </xforms:item>
```

9. The *value* option overrides the *xforms:value* of the listed choices. This is useful when working with *list* items. A *list* displays the *xforms:label* of a choice until it is selected, at which point it displays the *xforms:value* of a choice. This means that you can set the *value* option to override the default *xforms:value*, which will change what is displayed when a choice is selected.

   For example, you might have a data model that sets a full text label for the choices, but uses an abbreviated value. The following instance shows this, using the show attribute to set the *xforms:label* and the contents of each element to set the *xforms:value*:

   ```
   <xforms:instance id="currency">
      <data>
         <choice show="US Dollars">USD</choice>
         <choice show="CDN Dollars">CDN</choice>
         <choice show="Euros">Euro</choice>
      </data>
   </xforms:instance>
   ```

   In this case, you might want to display the full text label even after a choice is selected. You can do this by using *xforms:extension* to add a *value* to each item in the set. You can then set the *value* to compute its contents to equal the *label*, which is populated with the full text description. The following sample shows this:

   ```
   <checkgroup sid="currency">
      <xforms:select ref="currency" appearance="full">
         <xforms:label>Select the currencies you accept:</xforms:label>
         <xforms:itemset nodeset="instance('currency')/choice">
            <xforms:label ref="@show"></xforms:label>
            <xforms:value ref="."></xforms:value>
            <xforms:extension>
               <value compute="label"/>
            </xforms:extension>
         </xforms:itemset>
      </xforms:select>
   </checkgroup>
   ```

10. Any item with an *xforms:select1* option can trigger XForms actions using the *xforms-value-changed*, *xforms-select*, or *xforms-deselect* events. For more information, refer to “Details on XForms Event Handlers” on page 261.

xforms:submit

Sets a *button* or *action* item to perform and XForms submission. The rules for the submission are defined in the XForms data model. The *xforms:submit* links a button to a particular set of rules, which are then carried out when the button is clicked.

For more information about XForms submissions, see “xformsmodels” on page 211.

Syntax

```
<xforms:submit submission="ID">
   <xforms:label>label text</xforms:label>
   Alert Setting
   Hint Setting
   Help Setting
</xforms:submit>
```

ID	*string*	the ID of the *xforms:submission* you want to use. This is declared in the data model.
label text	*string*	sets the text that the button containing the submit displays, as well as the default accessibility message for that button. If the button also has *value* or *acclabel* options, they will override this setting.
Alert, Hint, Help Setting	*metadata*	see "Metadata Sub-Options" on page 210.

Available In

action, button

Example

The following example shows a button that is linked to the *completeSubmit* submission rules in the XForms data model. This button also relies on the xforms:label to set the text and accessibility message for the button.

```
<button sid="completeSubmit">
   <xforms:submit submission="completeSubmit">
      <xforms:label>Submit All Data</xforms:label>
   </xforms:submit>
</button>
```

Usage Details

1. You can also trigger submissions using the *xforms:send* action. For more information, refer to "xforms:send" on page 255.

xforms:switch

Allows you to divide a portion of the form into sets of items, and then control which set is shown to the user. For example, you may have a form page with Basic and Advanced settings, and may only want to show one type of settings to the user at any given time.

The switch option uses the *xforms:case* element to group the items into sets. Each set can contain any number of XFDL items, which are written normally as children of the *xforms:case*.

To change which set of items is displayed, you must use the *xforms:toggle* action.

Syntax

```
<switch id="name" single_node_binding xfdl:state="state">
   <xforms:case1 id="name" selected="boolean"
      ...XFDL items...
   </xforms:case1>
   ...
   <xforms:casen id="name" selected="boolean"
      ...XFDL items...
   </xforms:casen>
</switch>
```

Note:

- *there can be any number of xforms:case elements, each containing its own set of items.*

name	*string*	an arbitrary name that you assign to the switch or case. All names assigned to id attributes must be globally unique within the form.
single node binding	*string*	optional. See "Single Node Binding" on page 205.
state	*XPath*	optional. An XPath expression that is evaluated relative to the single node binding (if given) or the context node of the switch. If this resolves to a node that contains the ID of an <xforms:case>, then that case is active and shown to the user. Otherwise, the xforms:case with a *selected* attribute of true is shown. The selected case and the state node are synchronized (that is, changing one changes the other).
boolean	*boolean*	true indicates the case is active (that is, the one shown to the user); false indicates the case is not active. Note that only one case can be active within a single switch.
XFDL items	*XFDL*	the items that are displayed to the user if the case is active.

Available In

pane

Example

This example shows a switch that contains two cases: one if the user is single, and another if the user is married. The first case simply contains a label that informs the user that no additional information is necessary. The second case contains a number of fields that request some information about the user's spouse. To begin with, the items for the single case are displayed on the form, since it is selected (that is, selection = true).

```
<pane sid="marriageStatus">
   <xforms:switch>
      <xforms:case id="single" selected="true">
         <label sid="singleLabel">
            <xforms:output ref="spouse/none"/>
         </label>
      </xforms:case>
      <xforms:case id="married" selected="false">
         <field sid="spouseName">
            <xforms:input ref="spouse/name">
               <xforms:label>Enter your spouse's name:</xforms:label>
            </xforms:input>
         </field>
         <field sid="spouseAge">
            <xforms:input ref="spouse/age">
               <xforms:label>Enter your spouse's age:</xforms:label>
            </xforms:input>
         </field>
      </xforms:case>
   </xforms:switch>
</pane>
```

In addition, the following code shows a button that uses the *xforms:toggle* action to change the switch to show the married case. Note that the xforms:toggle is enclosed in an *xforms:trigger*, which is activated when the button is clicked and triggers the toggle action.

```
<button sid="setMarried">
   <xforms:trigger>
      <xforms:label></xforms:label>
      <xforms:toggle case="married" ev:event="DOMActivate"
         ></xforms:toggle>
   </xforms:trigger>
</button>
```

Usage Details

1. An *xforms:switch* can contain any number of *xforms:case* elements, which can in turn contain any number of XFDL items, including panes (with groups and switches) and tables (with repeats).
2. To change which case is selected, you must use the *xforms:toggle* action. For more information, see "xforms:toggle" on page 259.
3. An *xforms:switch* is not allowed in an *xforms:repeat* element.

xforms:trigger

This option triggers an event in response to an XFDL *action* or the user clicking an XFDL *button*. The event is expressed as an XForms action, which allow you to make various changes to the form. For example, you could use an XForms action to set which case in a switch statement is true, or you could us an XForms action to insert or delete items on the form.

Syntax

```
<xforms:trigger single_node_binding>
   <xforms:label>label text</xforms:label>
   XForms Action
   Alert Setting
   Hint Setting
   Help Setting
</xforms:trigger>
```

single node binding	*string*	optional. See “Single Node Binding” on page 205. Adding a single node binding to an *xforms:trigger* does not cause the trigger to share data with the bound node. It simply allows the trigger to inherit the relevant properties from the data model.
label text	*string*	sets the text that the button containing the trigger displays, as well as the default accessibility message for that button. If the button also has *value* or *acclabel* options, they will override this setting.
XForms Action	*(see below)*	
Alert, Hint, Help Setting	*metadata*	see “Metadata Sub-Options” on page 210.

XForms Action

The XForms action determines what type of action is actually triggered. This could be a single action, such as deleting a data node, or multiple actions, such as setting the value of multiple data nodes. For more information about using XForms actions, refer to “Details on XForms Actions” on page 245.

Available In

action, button

Example

The following example shows a button that toggles the case of a *switch* item. In this case, the toggle sets the *basic* case to be true. Note that this trigger does not include a *ref* attribute, since the button is always relevant.

```
<button sid="basicPrefs">
   <xforms:trigger>
      <xforms:label></xforms:label>
      <xforms:toggle case="basic" ev:event="DOMActivate"
         ></xforms:toggle>
   </xforms:trigger>
</button>
```

Usage Details

1. The *xforms:trigger* option triggers XForms actions through the *DOMActivate* event. For more information about actions, refer to "Details on XForms Actions" on page 245. For more information about events, refer to "Details on XForms Event Handlers" on page 261.
2. A button or action that contains the trigger to type must be of type select, which is the default if the type is omitted. Any other type will override the trigger.
3. An *xforms:trigger* does not set the *triggeritem* option for the form. If this is required, you must add a compute that will set the option when the button or action is activated.

xforms:textarea

This option links a multi-line field to an element in the data model so that they share data. For example, if you added an *xforms:textarea* option to a field in your form, you could use that option to link the field to an element in your data model. Once linked, any changes made to the data in one would be reflected by the other.

This option is only available if you are using an XForms data model.

Syntax

```
<xforms:textarea single_node_binding>
   <xforms:label>label text</xforms:label>
   Alert Setting
   Hint Setting
   Help Setting
</xforms:textarea>
```

single node binding	*string*	see "Single Node Binding" on page 205.
label text	*string*	sets the text for the field's built-in label, as well as the accessibility message for the field. Leave this setting empty for no label. If the item also has a *label* or *acclabel* option, they will override this setting.
Alert, Hint, Help Setting	*metadata*	see "Metadata Sub-Options" on page 210.

Available In

custom, field

Example

The following code shows an XForms model that contains a to, from, date, and note element:

```
<xformsmodels>
   <xforms:model xmlns="">
      <xforms:instance id="memo">
         <memorandum>
```

```
            <to></to>
            <from></from>
            <date></date>
            <note></note>
         </memorandum>
      </xforms:instance>
   </xforms:model>
</xformsmodels>
```

Using that data model, the following code links a field to the note element in the data model, so that they share data:

```
<field sid="noteField">
   <xforms:textarea ref="note">
      <xforms:label>Note:</xforms:label>
   </xforms:textarea>
   <value></value>
</field>
```

Usage Details

1. This option is for multi-line fields. To create a field with only a single line of input, use the *xforms:input* option instead.
2. If an XFDL field has a *texttype* setting of text/rtf, then the UI binding connects to the *rtf* option rather than the *value* option for that field.
3. To copy the plain text from an rich text field to the XForms data model, you must use a *custom* item with an *xforms:textarea*. Within that item, create a *custom* option that has a compute. This compute must copy the contents of the *value* option from the rich text field to the *value* option for the custom item. For example:

   ```
   <field sid="commentField">
      <xforms:textarea ref="comment/richText">
         <xforms:label>Comment:</xforms:label>
      </xforms:textarea>
      <texttype>text/rtf</texttype>
      <value>contains the plain text</value>
   </field>
   <custom:holder sid="plainTextHolder">
      <xforms:textarea ref="comment/plainText">
         <xforms:label/>
      <xforms:textarea>
      <custom:copytext compute="toggle(commentField.value) == 1 ?
         set(value, commentField.value) : ''"/>
      <value>is set to contain the plain text</value>
   </custom:holder>
   ```
4. Any item with an *xforms:textarea* option can trigger XForms actions using the *xforms-value-changed* event. For more information, refer to "Details on XForms Event Handlers" on page 261.
5. When you add an *xforms:textarea* to a custom item, it works just as though it was placed in a valid XFDL item. This means that a value option is created and is linked to the data model.

xforms:upload

Sets a button to attach a file to the form. The file is loaded directly into the XForms data model as base64 data. If the file is an image, you can use an *xforms:output* to display the file to the user.

This is equivalent to a button of type enclose, but does not allow multiple enclosures.

Syntax

```
<xforms:upload single_node_binding mediatype="MIME type">
   <xforms:label>label text</xforms:label>
   <xforms:mediatype single_node_binding></xforms:mediatype>
   <xforms:filename single_node_binding"></xforms:filename>
   Alert Setting
   Hint Setting
   Help Setting
</xforms:upload>
```

single node binding	*string*	See "Single Node Binding" on page 205.
MIME type	*MIME type*	filters the file types that the user can upload. This is a space delimited list of MIME types that are allowed. To limit uploads to those image formats supported by XFDL, set this to: `image/*`
label text	*string*	sets the text that the button containing the upload displays, as well as the default accessibility message for that button. If the button also has *value* or *acclabel* options, they will override this setting.
Alert, Hint, Help Setting	*metadata*	see "Metadata Sub-Options" on page 210.

Available In

button

Example

The following example show a button that uploads an employee assessment into the data model. This upload also copies the MIME type of the file and the filename to elements in the data model.

```
<button sid="loadAssessment">
   <xforms:upload ref="assessment/text"
      mediatype="text/plain">
      <xforms:label>Enclose Assessment</xforms:label>
      <xforms:mediatype ref="../mediatype"></xforms:mediatype>
      <xforms:filename ref="../filename"></xforms:filename>
   </xforms:upload>
</button>
```

Usage Details

1. An *xforms:upload* can only upload a single file. If you want to add multiple files to a form, you must create multiple upload buttons (or use custom XFDL constructs).
2. The only way to display an uploaded file is through the *xforms:output* option. However, this option is limited to displaying text and image files.

Details on XForms Actions

XForms actions are similar to the XFDL actions you can create using the *action* item. XForms actions allow you to initiate a number of processes, including submitting a form, setting a value in a form, inserting a row in a repeat table, and so on.

Unlike XFDL *action* items, XForms actions are triggered by events in the form. For example, you might create an action that occurs when the user clicks a button, when a particular value in the form has changed, or when a submission has returned an error.

This section provides general information about actions, and then details each action type in turn.

Syntax

Actions are written in two ways, depending on whether you want to use a group of action or a single action.

When using a group of actions, you place them in an <xforms:action> tag. This groups the actions together, so that the actions all respond to the same trigger event. Once triggered, each action is processed in turn. The following syntax applies:

```
<xforms:action event xfdl:if="condition">
   <action1 action_settings>
   ...
   <actionn action_settings>
</xforms:action>
```

event	*string*	this sets the event that will trigger the actions. For more information about events, refer to "Details on XForms Event Handlers" on page 261.
condition	*XPath*	*optional*. An XPath expression that evaluates to either true or false. If false, the actions within the *xforms:action* tag are not processed. This expression is evaluated relative to the context set by its nearest ancestor with a single node or nodeset binding. This allows for conditional logic within actions.
action	*string*	the type of action you want to use.
action_setting	*string*	one or more attributes that set any values required for the action.

Optionally, you can use only a single action. In this case, no <xforms:action> tag is required, as the triggering event is included on the action's tag. This is written as shown:

```
<action event xfdl:if="condition" action_settings>
```

action	*string*	the type of action you want to use.
event	*string*	this sets the event that will trigger the action. For more information about events, refer to “Details on XForms Event Handlers” on page 261.
condition	*XPath*	*optional*. An XPath expression that evaluates to either true or false. If false, the action is not processed. This expression is evaluated relative to the context set by its nearest ancestor with a single node or nodeset binding. This allows for conditional logic within actions.
action_setting	*string*	one or more attributes that set any values required for the action.

Note that all actions are in the XForms namespace, and are preceded by the *xforms:* prefix. Futhermore, all action settings are written as attributes.

Actions and XForms Functions

Actions that include an XPath reference may also use XForms functions to resolve that reference. For more information about XForms functions, refer to “Details on XForms Function Calls” on page 349.

Placing Actions in a Form

The placement of actions depends on the triggering event. For example, if an action is triggered by a button click, the action must be placed within the <xforms:trigger> element of the *button* item. Similarly, if an action is triggered by a change in the XForms model, it must be included as a child of <xforms:model> tag.

For example:

```
<button>
   <xforms:trigger>
   </xforms:trigger>
<button>
```

For more information about the events available and where they are placed, refer to “Details on XForms Event Handlers” on page 261.

xforms:delete

Deletes a row of elements from a table. The elements are first deleted from the XForms model, then the table’s repeat deletes the visible items that were linked to those data elements.

Syntax

```
   <xforms:delete event nodeset_binding at="index"/>
OR
   <xforms:action event>
      <xforms:delete nodeset="XPath" at="index"/>
   </xforms:action>
```

event	*string*	the XForms event that triggers the action.
nodeset binding	*special*	see "Single Node Binding" on page 205.
index	*string*	an index number that determines which row to delete. Indexing is one-based, meaning the first row is row 1, the second row is row 2, and so on.

Example

This example assumes you are working with a purchase order form that includes the following data instance:

```
<xforms:instance xmlns="">
   <po>
      <order>
         <row>
            <product>widget</product>
            <quantity>2</quantity>
            <unitCost>2.00</unitCost>
            <lineTotal>4.00</lineTotal>
         </row>
         ...
         <row>
            <product></product>
            <quantity>0</quantity>
            <unitCost>0</unitCost>
            <lineTotal>0</lineTotal>
         </row>
      </order>
      <subtotal>4.00</subtotal>
      <tax>1.12</tax>
      <totalCost>4.48</totalCost>
   </po>
</xforms:instance>
```

This data instance includes multiple <row> elements that represent the rows in a table. The last row is a "template" row. Template rows are not visible to the user. It simply contains the basic information for a table row, so that when the user wants to add a row to the table, the information needed to create that row is available. All other rows are visible to the user and contain data that the user has entered.

The *xforms:repeat* that creates this table must also exclude the template row, as shown:

```
<table sid="orderTable">
    <xforms:repeat nodeset="order/row[position()!=last()]"
       id="orderTable" startindex="1">
```

In this case, the nodeset includes all rows that are not in the last position in the table (which would be the template row). This ensures that the template row is never included in the table itself, and is never shown to the user.

To delete a row from this table, you need to create a control (such as a button) that the user can click to trigger an *xforms:delete* action. You also need to add some other actions to this control to account for special cases. The following button illustrates this:

```
<button sid="deleteRow">
   <xforms:trigger>
      <xforms:label>Delete Row</xforms:label>
      <xforms:action ev:event="DOMActivate">
         <xforms:delete nodeset="order/row[last()>1]"
            at="index('orderTable')"/>
         <xforms:insert nodeset="order/row[last()=1]" at="1"
            position="before"/>
         <xforms:setfocus control="orderTable"/>
      </xforms:action>
   </xforms:trigger>
</button>
```

In this button, the first action is the *xforms:delete*. This delete identifies the nodeset for the row. It uses the *last* function to ensure that there is more than one row left in the table (if there is more than one row, the last row will have an index greater than one). If there was only one row left, that would mean you were deleting the template row. In this case, the delete will not execute because the last row is not greater than one. The delete then uses the *index* function to determine which row of the repeat (called orderTable) the cursor is on, then deletes that row.

The next action is an *xforms:insert*. This action uses the *last* function to detect whether there is only one row left (in which case the last row is row 1). If there is only one row left, then the table is effectively empty, since the last row is always the invisible template row. In this case, the insert function adds a new, blank row to the table. This prevents the user from deleting all rows in the table, and thereby making the table disappear.

Finally, we use the *xforms:setfocus* action to reset the focus to the table. This is necessary because when the user clicks the "Delete Row" button, the focus shifts to the button, so we put it back to the table.

Usage Details

1. When deleting a row, the index for the *xforms:repeat* does not change. Once the row is deleted, the rows following are all renumbered (their index is reduced by one). This effectively places the focus on the row that followed the deleted row.
2. When processing the data model on the back end, make sure that you also exclude the template row. Otherwise, you will include a row of blank data in your results.

xforms:insert

Allows you to add a row of elements to a table. This function copies the last row of elements in the data model, then inserts the copy in the desired location in the data model. Once the copy is inserted in the data model, the table's repeat creates corresponding items that are displayed to the user.

Syntax

```
   <xforms:insert event nodeset_binding at="index"
      position="position"/>
OR
   <xforms:action event>
      <xforms:insert nodeset="XPath" at="index"
         position="position"/>
   </xforms:action>
```

event	*string*	the XForms event that triggers the action.
nodeset binding	*special*	see "Nodeset Binding" on page 209.
index	*string*	an index number that sets the insertion point. The copy is placed either before or after this row in the table. Indexing is one-based, meaning the first row is row 1, the second row is row 2, and so on.
position	*string*	determines whether the copy is placed before or after the insertion point (determined by the *at* setting). Valid settings are before and after.

Example

This example assumes you are working with a purchase order form that includes the following data instance:

```
   <xforms:instance xmlns="">
      <po>
         <order>
            <row>
               <product>widget</product>
               <quantity>2</quantity>
               <unitCost>2.00</unitCost>
               <lineTotal>4.00<lineTotal>
            </row>
            ...
            <row>
               <product></product>
               <quantity>0</quantity>
               <unitCost>0</unitCost>
               <lineTotal>0</lineTotal>
            </row>
         </order>
         <subtotal>4.00</subtotal>
         <tax>1.12</tax>
         <totalCost>4.48</totalCost>
      </po>
   </xforms:instance>
   <xforms:insert ev:event="xforms-model-construct-done"
      nodeset="order/row[last()=1]" at="1" position="before"/>
```

This data instance includes a single <row> element. This is the "template" row for the table. As such, it is is never used to store data. The purpose of this row is to provide a template that is copied when adding new rows to the form.

Since the data model begins with only one row, and that row is invisible to the user, we also add an *xforms:insert* to the data model. This insert is triggered when the form is first opened, once the XForms model is completely built. It uses the last function to determine whether there is only one row in the table. If there is, it add a new, blank row. This ensures that the table always begins with one blank row that the user can see.

The *xforms:repeat* that creates this table must also exclude the template row, as shown:

```
      <table sid="orderTable">
          <xforms:repeat nodeset="order/row[position()!=last()]"
             id="orderTable" startindex="1">
```

In this case, the nodeset includes all rows that are not in the last position in the table (which would be the template row). This ensures that the template row is never included in the table itself, and is never shown to the user.

To add more rows to this table, you must create a control (such as a button) that the user can click to trigger another *xforms:insert* action. You also need to add some other actions to this control account for special cases. The following button illustrates this:

```
   <button sid="addRow">
      <xforms:trigger>
         <xforms:label>Add Row</xforms:label>
         <xforms:action ev:event="DOMActivate">
            <xforms:insert nodeset="order/row" at="index('orderTable')"
               position="after"/>
            <xforms:setfocus control="orderTable"/>
         </xforms:action>
      </xforms:trigger>
   </button>
```

In this button, the first action is the *xforms:insert*. This insert uses the *index* function to determine which row of the repeat (called orderTable) the cursor is on, then adds a new row after the row with the focus.

The next action is an *xforms:setfocus*, which resets the focus to the table. This is necessary because when the user clicks the ″Add Row″ button, the focus shifts to the button, so we put it back to the table.

Usage Details

1. When adding a row, the index for the *xforms:repeat* is automatically updated to point to the row that was just added.
2. When processing the data model on the back end, make sure that you also exclude the template row. Otherwise, you will include a row of blank data in your results.

xforms:message

Sets a message that is displayed to the user in a small dialog box.

Syntax

```
   <xforms:message event level="modal">message</xforms:message>
OR
   <xforms:action event>
      <xforms:message level="modal">message</xforms:message>
   </xforms:action>
```

event	*string*	The XForms event that triggers the action.
level	modal	Determines the appearance of the message. This attribute must always be **modal**.
message	*string*	The text of the message.

Example

The following code shows an XForms model that contains the *xforms:message* action. When the model is first initiated, the xforms-ready event triggers the message action, which opens and dialog that says, "Data Model Ready".

```
<xforms:model>
   <xforms:instance id="data" xmlns="">
      <data>
         <field1>25</field1>
         <field2>0</field2>
         <field3></field3>
      </data>
   </xforms:instance>
   <xforms:message level="modal" ev:event="xforms-ready"
      >Data Model Ready</xforms:message>
</xforms:model>
```

xforms:rebuild

Causes the form viewing application to rebuild any internal data structures that are used to track computational dependencies within a particular model.

In general, the XForms processor automatically runs this action when required. As such, this action is included mostly for completeness.

Syntax

```
   <xforms:rebuild event model="model"/>
OR
   <xforms:action event>
      <xforms:rebuild model="model"/>
   </xforms:action>
```

event	*string*	the XForms event that triggers the action.
model	*string*	the ID of the model to rebuild. If the model attribute is omitted, then the default model is used.

Example

The following button rebuilds model X when clicked:

```
<button sid="rebuildX">
   <xforms:trigger>
      <xforms:label>Rebuild</xforms:label>
      <xforms:rebuild ev:event="DOMActivate" model="X"/>
   </xforms:trigger>
</button>
```

Usage Details

1. This feature is most likely to be used on scaled-down xforms processors, in which the implicit rebuild-recalculate-revalidate-refresh sequence is not implemented because of limited resources. In this case, explicit requests for these actions may force an exchange with a server.
2. If an action sequence includes a *setvalue* action that affects a node which is used in an XPath predicate in the nodeset of an XForms bind, then you can call *rebuild* to cause the XForms bind to be re-evaluated.

xforms:recalculate

Causes the forms viewing application to recalculate any instance data that is affected by computations and is not up-to-date. This affects all data instances in the designated model.

In general, the XForms processor automatically runs this action when required. As such, this action is included mostly for completeness.

Syntax

```
   <xforms:recalculate event model="model"/>
OR
   <xforms:action event>
      <xforms:recalculate model="model"/>
   </xforms:action>
```

event	*string*	the XForms event that triggers the action.
model	*string*	the ID of the model to recalculate. If the model attribute is omitted, then the default model is used.

Example

The following button recalculates model X when clicked:

```
<button sid="recalculateX">
   <xforms:trigger>
      <xforms:label>Recalculate</xforms:label>
      <xforms:recalculate ev:event="DOMActivate" model="X"/>
   </xforms:trigger>
</button>
```

Usage Details

1. This feature is most likely to be used on scaled-down xforms processors, in which the implicit rebuild-recalculate-revalidate-refresh sequence is not

implemented because of limited resources. In this case, explicit requests for these actions may force an exchange with a server.
2. Normally, recalculation occurs at the end of an action sequence. However, the form author may need to force an earlier recalculation if there is a *setvalue* action whose XPath references depend on the recalculated results of prior *setvalue* actions.

xforms:refresh

Causes the forms viewing application to update all user interface elements linked to a particular model, so that they match the underlying data in the XForms model.

In general, the XForms processor automatically runs this action when required. As such, this action is included simply for completeness.

Syntax

```
   <xforms:refresh event model="model"/>
OR
   <xforms:action event>
      <xforms:refresh model="model"/>
   </xforms:action>
```

event	*string*	the XForms event that triggers the action.
model	*string*	the ID of the model to refresh. If the model attribute is omitted, then the default model is used.

Example

The following button refreshes the model X when clicked:

```
<button sid="refreshX">
   <xforms:trigger>
      <xforms:label>Refresh</xforms:label>
      <xforms:refresh ev:event="DOMActivate" model="X"/>
   </xforms:trigger>
</button>
```

Usage Details

1. This feature is most likely to be used on scaled-down xforms processors, in which the implicit rebuild-recalculate-revalidate-refresh sequence is not implemented because of limited resources. In this case, explicit requests for these actions may force an exchange with a server.
2. If you are performing a number of consecutive submissions, refresh may be useful for updating the form to show progress, especially after submissions that return data to the form.

xforms:reset

Returns a particular XForms model to the state it was in when the form was opened. This allows the user the reset the contents of the form to their ″starting point″, which can increase usability of the form.

Syntax

```
   <xforms:reset event model="model"/>
OR
   <xforms:action event>
      <xforms:reset model="model"/>
   </xforms:action>
```

event	*string*	the XForms event that triggers the action.
model	*string*	the ID of the model to reset. If the model attribute is omitted, then the default model is used.

Example

The following model contains data for a change of address, as well as a submission that sends the information for processing. Once the data is submitted, an *xforms:reset* action is triggered. This action resets the data in the form, allowing the user to type in the next address change without have to close the form and open it again. This allows for rapid entry of several changes. Note that this requires the submission button to be of type submit rather than done.

```
<xforms:model>
   <xforms:instance xmlns="">
      <root>
         <customerInfo>
            <id/>
            <newAddress>
               <street/>
               <city/>
               <state/>
               <zip/>
            </newAddress>
         </customerInfo>
      </root>
   </xforms:instance>
   <xforms:submission id="submit" method="post"
      action="http://www.ibm.poserver.com/cgi-bin/updateAddress"
      includenamespaceprefixes="">
      <xforms:reset ev:event="xforms-submit-done"/>
   </xforms:submission>
</xforms:model>
```

xforms:revalidate

Causes the forms viewing application to validate all instance data in a particular model. This ensures that all validation checks have been performed.

In general, the XForms processor automatically runs this action when required. As such, this action is included simply for completeness.

Syntax

```
   <xforms:revalidate event model="model"/>
OR
   <xforms:action event>
      <xforms:revalidate model="model"/>
   </xforms:action>
```

event	*string*	the XForms event that triggers the action.
model	*string*	the ID of the model to revalidate. If the model attribute is omitted, then the default model is used.

Example

The following button revalidates model X when clicked:

```
   <button sid="revalidateX">
      <xforms:trigger>
         <xforms:revalidate ev:event="DOMActivate" model="X"/>
         <xforms:label>Revalidate</xforms:label>
      </xforms:trigger>
   </button>
```

Usage Details

1. This feature is most likely to be used on scaled-down xforms processors, in which the implicit rebuild-recalculate-revalidate-refresh sequence is not implemented because of limited resources. In this case, explicit requests for these actions may force an exchange with a server.

xforms:send

Triggers an XForms submission. The submission must already be defined in the XForms model.

Syntax

```
   <xforms:send event submission="submission"/>
OR
   <xforms:action event>
      <xforms:send submission="submission"/>
   </xforms:action>
```

event	*string*	the XForms event that triggers the action.
submission	*string*	the ID of the submission you want to trigger. You can include multiple submission attributes.

Example

The following example shows two <xforms:submission> elements. When triggered, the submission attempts to post the data for a purchase order to a cgi script for processing. If an error is encountered, the xforms:send action is triggered. This starts a second submission, called "error" which submits to a different server (possibly a fall-back server).

```
<xforms:submission id="S" method="post" includenamespaceprefixes=""
   action="http://www.ibm.poserver.com/cgi-bin/po">
   <xforms:send ev:event="xforms-submit-error"
      submission="error"/>
</xforms:submission>
<xforms:submission id="error" method="post" includenamespaceprefixes=""
   action="http://www.ibm.errorserver.com/cgi-bin/error">
</xforms:submission>
```

Usage Details

1. You can also use the *xforms:submit* option to initiate a send action.

xforms:setfocus

Sets the focus to a particular presentation element in the form.

Syntax

```
   <xforms:setfocus event control="XPath"/>
OR
   <xforms:action event>
      <xforms:setfocus control="XPath"/>
   </xforms:action>
```

event	*string*	the XForms event that triggers the action.
XPath	*XPath reference*	an XPath reference to an element in the data model. The presentation element that is linked to this data element will receive the focus.

Example

The following examples shows a button that is used to add new rows to a table:

```
<button sid="addRow">
   <xforms:trigger>
      <xforms:label>Add Row</xforms:label>
      <xforms:action ev:event="DOMActivate">
         <xforms:insert nodeset="order/row" at="index('orderTable')"
            position="after"/>
         <xforms:setfocus control="orderTable"/>
      </xforms:action>
   </xforms:trigger>
</button>
```

When the button is clicked by the user, the focus moves to the button itself. However, once the new row appears, it's preferable to send the focus back to the table.

To accomplish this, the *xforms:setfocus* is included as the last action for the button, and moves the focus back to the *xforms:repeat* that controls the table. The focus is then automatically placed on the row that was just added.

Usage Details

1. You can use *xforms:setfocus* to refer to specific items in a repeat template. In this case, the row index of the repeat determines which item gets the focus.

   For example, consider a case in which each row of repeat X contains a field and a popup, and the repeat begins with an index of 2. You set the focus to the popup item. To locate this item, we first go to row 2 of the repeat, then find the popup within that row.
2. Setting the focus to an element on a different page of the form will change the page that is displayed.

xforms:setindex

Sets the index for the *xforms:repeat* element in a table. This determines which row in the table receives the focus.

Rows use one-based indexing. This means that the first row has an index of 1, the second and index of 2, and so on.

Syntax

```
   <xforms:setindex event repeat="ID" index="index"/>
OR
   <xforms:action event>
      <xforms:setindex repeat="ID" index="index"/>
   </xforms:action>
```

event	*string*	the XForms event that triggers the action.
id	*XPath reference*	the ID of the <xforms:repeat> element for which you want to set the index.
index	*XPath*	the number to set the index to.

Example

The following button deletes a row from a table:

```
<button sid="deleteRow">
   <xforms:trigger>
      <xforms:label>Delete Row</xforms:label>
      <xforms:action ev:event="DOMActivate">
         <xforms:delete nodeset="order/row"
            at="index('orderTable')"/>
         <xforms:setindex index="index" repeat="orderTable/>
         <xforms:setfocus control="orderTable"/>
      </xforms:action>
   </xforms:trigger>
</button>
```

The third action performed by this button is an *xforms:setindex*. In this case, *xforms:setindex* ensures that the index is always focused on the last row of the table and unless the row to be deleted is selected by the user.

Usage Details

1. You can use *xforms:setindex* to set the index of a repeat that is nested in another repeat. In this case, the row index of the outer repeat determines which inner repeat is set.

   For example, consider a case in which repeat X contains repeat Y. Repeat X has an index of 2, and repeat Y has an index of 5. You set repeat Y to an index of 3. To locate the correct repeat, we first go to row two of repeat X, then find repeat Y within that row.

xforms:setvalue

Sets the value for a specified element in the data model.

Syntax

```
   <xforms:setvalue event single_node_binding
      value="value XPath">value</xforms:setvalue>
OR
   <xforms:action event>
      <xforms:setvalue ref="XPath" model="model"
         value="value XPath"/>value</xforms:setvalue>
   </xforms:action>
```

event	*string*	the XForms event that triggers the action.
single node binding	*special*	see “Single Node Binding” on page 205.
value XPath	*XPath reference*	optional. If included, the result of the XPath reference is used to set the value. This overrides the *value* setting.
value	*string*	optional. If included, this string is used to set the value. If the *value XPath* setting is present, it overrides this setting.

Example

The following data instance contains two address blocks: one for the home address and one for the mailing address:

```
<xforms:instance xmlns="" id="registration">
   <address>
      <home>
         <street/>
         <city/>
         <state/>
         <zip/>
      </home>
      <mailing>
         <street/>
         <city/>
         <state/>
         <zip/>
         </mailing>
   </address>
</xforms:instance>
```

In this case, if the addresses are the same, it might be useful to include a button in the form that will copy all of the data from the home address to the mailing address, as shown:

```
<button sid="copyAddress">
   <xforms:trigger>
      <xforms:action ev:event="DOMActivate">
         <xforms:setvalue ref="address/mailing/street"
            value="../../home/street"/>
         <xforms:setvalue ref="address/mailing/city"
            value="../../home/street"/>
         <xforms:setvalue ref="address/mailing/state"
            value="../../home/state"/>
         <xforms:setvalue ref="address/mailing/zip"
            value="../../home/zip"/>
      </xforms:action>
   </xforms:trigger>
</button>
```

Usage Details

1. The *xforms:setvalue* action does not work on values that are set to be readonly in the XForms model; however, it does work on values that are set to be readonly through the *readonly* option.

xforms:toggle

Selects one of the cases in an *xforms:switch* and makes it active. When one case is selected, all other cases in the switch are deselected.

Syntax

```
   <xforms:toggle event case="case"/>
OR
   <xforms:action event>
      <xforms:toggle case="case"/>
   </xforms:action>
```

event	*string*	the XForms event that triggers the action.
case	*string*	the ID of the case to select.

Example

The following button assumes that you have an *xforms:switch* with two cases: single and married. When clicked, the button sets the switch to the married case:

```
<button sid="setMarried">
   <xforms:trigger>
      <xforms:label>Married</xforms:label>
      <xforms:toggle case="married" ev:event="DOMActivate"
         ></xforms:toggle>
   </xforms:trigger>
</button>
```

Usage Details

1. XFDL allows an *xforms:switch* option to appear inside an *xforms:repeat* option. In this case, the row index of the repeat determines which switch is affected by the *xforms:toggle* action.

For example, consider a case in which repeat X contains a switch, and repeat X begins with an index of 2. You toggle the case of the switch. To locate the correct switch, we first locate row 2 of the repeat, then locate the switch within that row.

Details on XForms Event Handlers

XForms event handlers track events in the form, such as a button click or the selection of a particular choice. When these events occur, they are registered by the XForms system. This allows you to create actions that are triggered by these events. For example, you might create an action that is triggered when a particular button is clicked, or when a particular choice in a list is selected.

This section provides general information about event handlers, and then details each event in turn.

Syntax

XForms event handlers exist in the following namspace:

```
http://www.w3.org/2001/xml-events
```

By convention, XFDL uses the *ev* prefix for this namespace, which is normally declared on the <XFDL> element of the form as shown:

```
<XFDL xmlns:ev="http://www.w3.org/2001/xml-events">
```

Event handlers themselves are declared as an attribute on XForms actions, and are written as shown:

```
ev:event="event"
```

When creating an event handler, you can add the attribute to either the general <xforms:action> tag or, for a single action event handler, directly to any specific XForms action tag.

When using the <xforms:action> tag, the individual actions do not require event handlers, as shown:

```
<xforms:action ev:event="event">
   <action1 action_settings>
   ...
   <actionn action_settings>
</xforms:action>
```

In this case, the event triggers all of the actions contained in the <xforms:action> tag, which are then processed in the order listed.

When using a single action, the event is added to the action tag as shown:

```
<action ev:event="event" action_settings>
```

In this case, the occurrence of the event triggers only the single action containing the ev:event.

Placing Events in a Form

The ev:event attribute is always placed within an action tag to make it an event handler, but the type of event you use dictates where in the form that action may be placed. For example, actions that rely on a button press must be placed within the *xforms:trigger* option in that button.

In general, the event handler for an event (the action containing an ev:event for a given event) must appear as a child element of the XForms element that receives the event.

For more information about where specific events can be placed, refer to the detailed description for that event.

DOMActivate

Detects the activation of the presentation element that contains this event. For example, clicking a button registers the DOMActivate event for that button.

Syntax

```
ev:event="DOMActivate"
```

Available In

action item, button item

Example

The following button assumes that you have an *xforms:switch* with two cases: single and married. When clicked, the *DOMActivate* event triggers an *xforms:toggle* action, which sets the switch to the married case:

```
<button sid="setMarried">
   <xforms:trigger>
      <xforms:label></xforms:label>
      <xforms:toggle case="married" ev:event="DOMActivate"/>
   </xforms:trigger>
</button>
```

Usage Details

1. See the *xforms:insert* action for an example of an event handler that performs more than one XForms action.

xforms-deselect

Occurs when a choice in an *xforms:select*, *xforms:select1*, or *xforms:switch* option that was previously selected becomes deselected.

Syntax

```
ev:event="xforms-deselect"
```

Available In

<xforms:case> element, <xforms:item> element

Example

The following list allows the user to choose one or more peripherals that they want included when purchasign a computer. The list contains three choices (mouse, keyboard, and USB memory stick) that are represented by three <xforms:item> tags. Each item also contains some *xforms:setvalue* actions. When the user selects an accessory, an *xforms-select* event occurs for that choice. This triggers the the first setvalue action in that item, which sets an element in the data model to "Yes". When the user deselects and accessory, that choice registers an *xforms-deselect* even and triggers the second setvalue action in that item, which resets the element in the data model to blank.

```
<list sid="accessories">
   <xforms:select ref="po/accessories" appearance="compact">
      <xforms:label>Select the accessory:</xforms:label>
      <xforms:item>
         <xforms:label>Mouse</xforms:label>
         <xforms:value>Mouse</xforms:value>
         <xforms:setvalue ev:event="xforms-select"
            ref="po/accessories/mouse" value="Yes"/>
         <xforms:setvalue ev:event="xforms-deselect"
            ref="po/accessories/mouse" value=""/>
      </xforms:item>
      <xforms:item>
         <xforms:label>Keyboard</xforms:label>
         <xforms:value>Keyboard</xforms:value>
         <xforms:setvalue ev:event="xforms-select"
            ref="po/accessories/keyboard" value="Yes"/>
         <xforms:setvalue ev:event="xforms-deselect"
            ref="po/accessories/keyboard" value=""/>
      </xforms:item>
      <xforms:item>
         <xforms:label>USB Memory Stick</xforms:label>
         <xforms:value>USB</xforms:value>
         <xforms:setvalue ev:event="xforms-select"
            ref="po/accessories/USB" value="Yes"/>
         <xforms:setvalue ev:event="xforms-select"
            ref="po/accessories/USB" value=""/>
      </xforms:item>
   </xforms:select>
</list>
```

Usage Details

1. The *xform-deselect* event only occurs in *xforms:select* and *xforms:select1* options that use the <xforms:item> element. If the option uses the <xforms:itemset> element, use the *xforms-value-changed* event instead.
2. The *xforms-deselect* event does not occur within a *combobox* item because the selection is open.

xforms-disabled

Occurs when an data node changes its state from relevant to non-relevant, or when a data node that is not relevant changes value and remains non-relevant.

This event is triggered on the XForms control bound to that node.

Syntax

```
ev:event="xforms-disabled"
```

Available In

xforms:input, xforms:output, xforms:range, xforms:secret, xforms:select, xforms:select1, xforms:submit, xforms:textarea, xforms:trigger, xforms:upload

Example

This example shows a label item that displays the total for a purchase order. If the total is less than 100, then the data node becomes non-relevant and the *xforms-disabled* event is triggered. In this case, the event triggers an *xforms:message* action that explains that the total is too low.

The following bind sets the minimum value of the po/total node in the data model to be 100:

```
<xforms:bind nodeset="po/total" relevant=". &gt; 100"/>
```

The following code defines the label that displays the total:

```
<label sid="Total">
   <xforms:output ref="po/total">
      <xforms:message ev:event="xforms-disabled" level="modal"
         >Values less than $100 should be paid from petty cash.</xforms:message>
   </xforms:output>
</label>
```

Usage Details

1. The order in which user interface events are processed is indeterminate. This means you cannot rely on them processing in a particular order.

xforms-enabled

Occurs when a data node that non-relevant (relevant = false) becomes relevant, or when a node that is relevant changes value and remains relevant.

This event is triggered on the XForms control bound to that node.

Syntax

```
ev:event="xforms-enabled"
```

Available In

xforms:input, xforms:output, xforms:range, xforms:secret, xforms:select, xforms:select1, xforms:submit, xforms:textarea, xforms:trigger, xforms:upload

Example

This example shows a label item that displays the total for a purchase order. If the total is less than 100, then the data node becomes non-relevant. If the value is then changed to be greater than 100, the node becomes relevant again and the *xforms-enabled* event is triggered. In this case, the event triggers an *xforms:message* action that explains that the value is now acceptable.

The following bind sets the minimum value of the po/total node in the data model to be 100:

```
<xforms:bind nodeset="po/total" relevant=". &lt; 100"/>
```

The following code defines the label that displays the total:

```
<label sid="Total">
   <xforms:output ref="po/total">
      <xforms:message ev:event="xforms-disabled"
         level="modal">The value is acceptable.</xforms:message>
   </xforms:output>
</label>
```

Usage Details

1. The order in which user interface events are processed is indeterminate. This means you cannot rely on them processing in a particular order.

xforms-invalid

Occurs when a data node changes its state from invalid to valid, or when data that is invalid changes value and remains invalid. Validity is determined based on whether the data matches the data type and constraints specified in the model, as well as the schema validity. Note that the state of ″required but empty″ is valid.

This event is triggered on the XForms control bound to that node.

Syntax

```
ev:event="xforms-invalid"
```

Available In

xforms:input, xforms:output, xforms:range, xforms:secret, xforms:select, xforms:select1, xforms:submit, xforms:textarea, xforms:trigger, xforms:upload

Example

This example shows a label item that displays the total for a purchase order. If the total exceeds 10,000, then the data node becomes invalid and the *xforms-invalid* event is triggered. In this case, the event triggers an *xforms:message* action that explains that the total is too high.

The following bind sets the maximum value of the po/total node in the data model to be 10,000:

```
<xforms:bind nodeset="po/total" constraint=". &lt; 10000"/>
```

The following code defines the label that displays the total:

```
<label sid="Total">
   <xforms:output ref="po/total">
      <xforms:message ev:event="xforms-invalid"
         level="modal">Total exceeds maximum allowed valued.</xforms:message>
   </xforms:output>
</label>
```

Usage Details

1. The order in which user interface events are processed is indeterminate. This means you cannot rely on them processing in a particular order.

xforms-model-construct

Occurs when the forms viewing application first opens a form and begins to construct the XForms model.

Syntax

```
ev:event="xforms-model-construct"
```

Available In

<xforms:model> element

Example

The following model contains simple data for a test form. When the form is first opened, an *xforms-model-construct* event occurs as the forms viewer begins to construct the model. This triggers the *xforms:message* action, which opens a dialog that says, ″Beginning model construction.″

```
<xforms:model>
   <xforms:instance xmlns="">
      <testmodel>
         <a/>
         <b/>
         <c/>
      </testmodel>
   </xforms:instance>
   <xforms:message level="modal" ev:event="xforms-model-construct"
      >Beginning model construction.</xforms:message>
</xforms:model>
```

Usage Details

1. Because the data structures for XForms instances are not created until after action handlers for this event are run, few XForms actions other than message will work. This event is mainly for debugging.

xforms-model-construct-done

Occurs when the forms viewing application has finished constructing the XForms model. This occurs just after the form is opened, and always completes before the user can interact with the form.

Syntax

```
ev:event="xforms-model-construct-done"
```

Available In

<xforms:model> element

Example

The following model contains data for a purchase order form. The data begins with a single template row which is set to be non-relevant by a bind. This means the row is not visible in the form. When the form is first opened, the forms viewer will construct the XForms data model, creating a table with a single invisible row in it. When it has completed building the model, an *xforms-model-construct-done* event occurs. This triggers the *xforms:insert* action, which duplicates the template row so that there is a visible row that the user can work with.

```
<xforms:model>
   <xforms:instance id="po" xmlns="">
      <po>
         <order>
            <row>
               <product/>
               <unitCost>0</unitCost>
               <qty></qty>
               <lineTotal></lineTotal>
            </row>
         </order>
         <subtotal>0</subtotal>
         <tax>0</tax>
         <total>0</total>
      </po>
   </xforms:instance>
   <xforms:bind nodeset="order/row[last()]" relevant="false()"/>
   <xforms:insert ev:event="xforms-model-construct-done"
      nodeset="order/row[last()=1]" at="1" position="before"/>
</xforms:model>
```

Usage Details

1. The *xforms-model-construct-done* event is appropriate for initializing data because the user interface has not yet been processed (that is, repeats have not been expanded, and form controls have not been recognized). However, because the user interface layer is not yet available, actions that operate on the UI layer are not appropriate in event handlers for this event. For example, toggle, setfocus and setindex will not work. To initialize the user interface, use the *xforms-ready* event instead.

xforms-model-destruct

Occurs when the XForms model is removed from memory. This generally happens when the form is closed or submitted.

Syntax

```
ev:event="xforms-model-destruct"
```

Available In

<xforms:model> element

Example

The following model contains simple data for a test form. When the is closed, the forms viewer will destroy the XForms data model and an *xforms-model-destruct-done* event will occur. This triggers the *xforms:message* action, which opens a dialog that says, ″Model destroyed.″

```
<xforms:model>
   <xforms:instance xmlns="">
      <testmodel>
         <a/>
         <b/>
         <c/>
      </testmodel>
   </xforms:instance>
   <xforms:message level="modal" ev:event="xforms-model-destruct"
      >Model destroyed.</xforms:message>
</xforms:model>
```

xforms-optional

Occurs when a data node that is required (required = true) changes to being optional, or when a data node that is required changes value and remains required.

This event is triggered on the XForms control bound to that node.

Syntax

```
ev:event="xforms-optional"
```

Available In

xforms:input, xforms:output, xforms:range, xforms:secret, xforms:select, xforms:select1, xforms:submit, xforms:textarea, xforms:trigger, xforms:upload

Example

This example shows a field item that accepts the first name of the user's spouse. This field is required if they select the ″married″ radio button in the form. However, if they then select the ″single″ radio button, the field becomes optional and the *xforms-optional* event is triggered. In this case, the event triggers the *xforms:message* action, which tells the user that the spousal information is no longer required.

The following bind sets makes the spouse's name required if the married radio button is selected:

```
<xforms:bind nodeset="personalInfo/spouseFirstName"
   required="../marital_status = 'married'"/>
```

The following code defines the label that displays the total:

```
<field sid="spouseFirstName">
   <xforms:input ref="personalInfo/spouseFirstName">
      <xforms:label>Spouse's First Name:</xforms:label>
      <xforms:message ev:event="xforms-optional"
         level="modal">Spousal data no longer required.</xforms:message>
   </xforms:input>
</field>
```

Usage Details

1. The order in which user interface events are processed is indeterminate. This means you cannot rely on them processing in a particular order.

xforms-readonly

Occurs when a data node is read-write (readonly = false) becomes readonly, or when a node that is readonly changes value and reamains readonly.

This event is triggered on the XForms control bound to that node.

Syntax

```
ev:event="xforms-readonly"
```

Available In

xforms:input, xforms:output, xforms:range, xforms:secret, xforms:select, xforms:select1, xforms:submit, xforms:textarea, xforms:trigger, xforms:upload

Example

This example shows a field item that accepts the first name of the user's spouse. When the user selects the ″married″ radio button, this field becomes read-write. If the user then select ″single″, the field becomes readonly and the *xforms-readonly* event is triggered. In this case, the event triggers the *xforms:message* action, which tells the user that the spousal information is not required.

The following bind sets makes the spouse's name readonly if the ″single″ radio button is selected:

```
<xforms:bind nodeset="personalInfo/spouseFirstName" readonly="../single = 'on'"/>
```

The following code defines the label that collects the spouse's first name:

```
<field sid="spouseFirstName">
   <xforms:input ref="personalInfo/spouseFirstName">
      <xforms:label>Spouse's First Name:</xforms:label>
      <xforms:message ev:event="xforms-required"
         level="modal">Spousal information is not required.</xforms:message>
   </xforms:input>
</field>
```

Usage Details

1. The order in which user interface events are processed is indeterminate. This means you cannot rely on them processing in a particular order.

xforms-readwrite

Occurs when a data node that is readwrite (readwrite = true) becomes read-write, or when a node that is read-write changes value and remains read-write.

This event is triggered on the XForms control bound to that node.

Syntax

```
ev:event="xforms-readwrite"
```

Available In

xforms:input, xforms:output, xforms:range, xforms:secret, xforms:select, xforms:select1, xforms:submit, xforms:textarea, xforms:trigger, xforms:upload

Example

This example shows a field item that accepts the first name of the user's spouse. When the user selects the ″married″, the ″single″ radio button goes off and the field becomes readwrite, triggering an *xforms-readwrite* event. In this case, the event triggers the *xforms:message* action, which tells the user that the spousal information is required.

The following bind sets makes the spouse's last name readonly if the ″single″ radio button is selected:

```
<xforms:bind nodeset="personalInfo/spouseFirstName" readonly="../single = 'on'"/>
```

The following code defines the label that displays the total:

```
<field sid="spouseFirstName">
   <xforms:input ref="personalInfo/spouseFirstName">
      <xforms:label>Spouse's First Name:</xforms:label>
      <xforms:message ev:event="xforms-readwrite" level="modal"
         >You must provide all listed spousal information.</xforms:message>
   </xforms:output>
</label>
```

Usage Details

1. The order in which user interface events are processed is indeterminate. This means you cannot rely on them processing in a particular order.

xforms-ready

Occurs when the forms viewing application has finished the initial set up of all XForms constructs and is ready for user interaction.

Syntax

```
ev:event="xforms-ready"
```

Available In

<xforms:model> element

Example

The following model contains simple data for a test form. When the form is first opened, the forms viewer will construct the XForms data model. When is has completed building the model, an *xforms-ready* event occurs. This triggers the *xforms:setfocus* action, which sets the focus to the last element in the data model.

```
<xforms:model>
   <xforms:instance xmlns="">
      <testmodel>
         <a/>
         <b/>
         <c/>
      </testmodel>
   </xforms:instance>
   <xforms:setfocus ev:event="xforms-ready" control="c"/>
</xforms:model>
```

Usage Details

1. You can also use this event to trigger data initialization (as you can with *xforms-model-construct-done*), but *xforms-ready* is the primary event for triggering actions that initialize the user interface, such as setfocus, setindex, and toggle.

xforms-required

Occurs when an data node that is optional (required = false) becomes required, or when a data node that is required changes value and remains required.

This event is triggered on the XForms control bound to that node.

Syntax

```
ev:event="xforms-required"
```

Available In

xforms:input, xforms:output, xforms:range, xforms:secret, xforms:select, xforms:select1, xforms:submit, xforms:textarea, xforms:trigger, xforms:upload

Example

This example shows a field item that accepts the first name of the user's spouse. When the user selects the "married" radio button, this field becomes required and the *xforms-required* event is triggered. In this case, the event triggers the *xforms:message* action, which tells the user that the spousal information is required.

The following bind sets makes the spouse's last name required if the married radio button is selected:

```
<xforms:bind nodeset="personalInfo/spouseFirstName"
   required="../marital_status = 'married'"/>
```

The following code defines the label that displays the total:

```
<field sid="spouseFirstName">
   <xforms:input ref="personalInfo/spouseFirstName">
      <xforms:label>Spouse's First Name:</xforms:label>
      <xforms:message ev:event="xforms-required" level="modal"
         >You must provide all listed spousal information.</xforms:message>
   </xforms:input>
</field>
```

Usage Details

1. The order in which user interface events are processed is indeterminate. This means you cannot rely on them processing in a particular order.

xforms-select

Occurs when a choice is selected from an *xforms:select*, *xforms:select1*, or *xforms:switch* option. This event only occurs on the item that is selected.

Syntax

```
ev:event="xforms-select"
```

Available In

<xforms:case> element, <xforms:item> element

Example

The following list allows the user to choose one or more peripherals that they want included when purchasign a computer. The list contains three choices (mouse, keyboard, and USB memory stick) that are represented by three <xforms:item> tags. Each item also contains some *xforms:setvalue* actions. When the user selects an accessory, an *xforms-select* event occurs for that choice. This triggers the the first setvalue action in that item, which sets an element in the data model to ″Yes″. When the user deselects and accessory, that choice registers an *xforms-deselect* even and triggers the second setvalue action in that item, which resets the element in the data model to blank.

```
   <list sid="accessories">
      <xforms:select ref="po/accessories" appearance="compact">
         <xforms:label>Select the accessory:</xforms:label>
         <xforms:item>
            <xforms:label>Mouse</xforms:label>
            <xforms:value>Mouse</xforms:value>
            <xforms:setvalue ev:event="xforms-select"
               ref="po/accessories/mouse" value="Yes"/>
            <xforms:setvalue ev:event="xforms-deselect"
               ref="po/accessories/mouse" value=""/>
         </xforms:item>
         <xforms:item>
            <xforms:label>Keyboard</xforms:label>
            <xforms:value>Keyboard</xforms:value>
            <xforms:setvalue ev:event="xforms-select"
               ref="po/accessories/keyboard" value="Yes"/>
            <xforms:setvalue ev:event="xforms-deselect"
               ref="po/accessories/keyboard" value=""/>
         </xforms:item>
         <xforms:item>
            <xforms:label>USB Memory Stick</xforms:label>
            <xforms:value>USB</xforms:value>
            <xforms:setvalue ev:event="xforms-select"
               ref="po/accessories/USB" value="Yes"/>
            <xforms:setvalue ev:event="xforms-select"
               ref="po/accessories/USB" value=""/>
         </xforms:item>
      </xforms:select>
   </list>
```

Usage Details

1. The *xform-select* event only occurs in *xforms:select* and *xforms:select1* options that use the <xforms:item> element. If the option uses the <xforms:itemset> element, use the *xforms-value-changed* event instead.

xforms-submit

Occurs when an XForms submission begins.

Syntax

```
ev:event="xforms-submit"
```

Available In

<xforms:submission> element

Example

The following example illustrates how you can remove empty rows from a table before submitting a form.

First, you must set up your <xforms:submission> as shown below. When this submission begins, an *xforms-submit* event occurs. This triggers the first *xforms:setvalue* action, which sets an element in the data model to ″true″ to indicate that a submission is in progress. If the submission returns an error, an *xforms-submit-error* event occurs. This triggers the second *xforms:setvalue* action, which sets the same element in the data model to ″false″ to indicate that a submission is no longer occurring.

```
<xforms:submission id="S" method="post" includenamespaceprefixes=""
   action="http://www.ibm.poserver.com/cgi-bin/po">
   <xforms:setvalue ev:event="xforms-submit"
      ref="instance('temps')/submitting" value="'true'"/>
   <xforms:setvalue ev:event="xforms-submit-error"
      ref="instance('temps')/submitting" value="'false'"/>
</xforms:submission>
```

Next, you must create an <xforms:bind> that affects the relevancy of the rows in the form, as shown:

```
<xforms:bind nodeset="order/row[not(last())]"
   relevant="boolean-from-string( if( qty > 0 or
   instance('temps')/submitting='false', 'true', 'false'))"/>
```

This bind applies the following logic to each row in the table (except for the last row, which is assumed to be a template row): if the quantity is greater than zero or the form is not submitting, then the row is relevant; otherwise, the row is not relevant. This effectively makes all empty rows non-relevant when the form is being submitted (keeping in mind that template rows are already marked as non-relevant).

xforms-submit-done

Occurs when an XForms submission has successfully completed.

Syntax

```
ev:event="xforms-submit-done"
```

Available In

<xforms:submission> element

Example

The following model includes two submissions: the first submission sends the form to a shipping database while the second submission sends the form to a billing database. The form is set up so that when the user clicks the submit button, only the first submission is triggered. Once that submission has successfully completed, an *xforms-submit-done* event occurs. This triggers the *xforms:send* action in the first submission, which in turn triggers the second submission. This prevents billing from occurring if the order was not properly registered with shipping.

```
<xforms:model>
   <xforms:instance xmlns="">
      <po>
         <order>
            <row>
               <product/>
               <unitCost>0</unitCost>
               <qty></qty>
               <lineTotal></lineTotal>
            </row>
         </order>
      </po>
   </xforms:instance>
   <xforms:submission id="submitShipping" method="post"
      action="http://www.ibm.poserver.com/cgi-bin/shipping"
      includenamespaceprefixes="">
      <xforms:send ev:event="xforms-submit-done"
         submission="submitBilling"/>
   </xforms:submission>
   <xforms:submission id="submitBilling" method="post"
      action="http://www.ibm.poserver.com/cgi-bin/billing"
      includenamespaceprefixes=""/>
</xforms:model>
```

Usage Details

1. The xforms-submit-done event does not work with *replace all* submissions (refer to the *xforms:send* action for more information).

xforms-submit-error

Occurs when an XForms submission returns an error.

Syntax

```
ev:event="xforms-submit-error"
```

Available In

<xforms:submission> element

Example

The following example illustrates how you can remove empty rows from a table before submitting a form.

First, you must set up your <xforms:submission> as shown below. When this submission begins, an *xforms-submit* event occurs. This triggers the first *xforms:setvalue* action, which sets an element in the data model to ″true″ to indicate that a submission is in progress. If the submission returns an error, an *xforms-submit-error* event occurs. This triggers the second *xforms:setvalue* action, which sets the same element in the data model to ″false″ to indicate that a submission is no longer occurring.

```
<xforms:submission id="S" method="post" includenamespaceprefixes=""
      action="http://www.ibm.poserver.com/cgi-bin/po">
      <xforms:setvalue ev:event="xforms-submit"
         ref="instance('temps')/submitting" value="'true'"/>
      <xforms:setvalue ev:event="xforms-submit-error"
         ref="instance('temps')/submitting" value="'false'"/>
</xforms:submission>
```

Next, you must create an <xforms:bind> that affects the relevancy of the rows in the form, as shown:

```
<xforms:bind nodeset="order/row[not(last())]"
      relevant="boolean-from-string( if( qty > 0 or
      instance('temps')/submitting='false', 'true', 'false'))"/>
```

This bind applies the following logic to each row in the table (except for the last row, which is assumed to be a template row): if the quantity is greater than zero or the form is not submitting, then the row is relevant; otherwise, the row is not relevant. This effectively makes all empty rows non-relevant when the form is being submitted (keeping in mind that template rows are already marked as non-relevant).

xforms-valid

Occurs when a data node changes its state from invalid to valid, or when a valid node changes its value and remains valid. Validity is determined based on whether the data matches the data type and constraints specified in the model, as well as the schema validity. Note that the state of ″required but empty″ is valid.

This event is triggered on the XForms control bound to that node.

Syntax

```
   ev:event="xforms-valid"
```

Available In

xforms:input, xforms:output, xforms:range, xforms:secret, xforms:select, xforms:select1, xforms:submit, xforms:textarea, xforms:trigger, xforms:upload

Example

This example shows a label item that displays the total for a purchase order. If the total exceeds 10,000, then the data node becomes invalid. If the total is then changed so that is it less than 10,000 then it becomes valid again, and the *xforms-valid* event is triggered. In this case, the event triggers an *xforms:message* action that explains that the value is now acceptable.

The following bind sets the maximum value of the po/total node in the data model to be 10,000:

```
<xforms:bind nodeset="po/total" constraint=". &lt; 10000"/>
```

The following code defines the label that displays the total:

```
<label sid="Total">
   <xforms:output ref="po/total">
      <xforms:message ev:event="xforms-valid"
         level="modal">Total is valid.</xforms:message>
   </xforms:output>
</label>
```

Usage Details

1. The order in which user interface events are processed is indeterminate. This means you cannot rely on them processing in a particular order.

xforms-value-changed

Occurs when a value is changed in an XForms option.

Syntax

```
ev:event="xforms-value-changed"
```

Available In

xforms:input option, xforms:output option, xforms:range option, xforms:secret option, xforms:select option, xforms:select1 option, xforms:textarea option

Example

The following example assumes that you are working with a purchase order form that contains two data instances. The first data instance contains information about the products that can be purchases, as shown:

```
<xforms:instance id="products" xmlns="">
   <products>
      <product name="Widget" code="W1" unitcost="9.99"/>
      <product name="Gadget" code="G1" unitcost="5.49"/>
      <product name="Trinket" code="T1" unitcost="11.25"/>
      <product name="Gromet" code="G2" unitcost="7.77"/>
   </products>
</xforms:instance>
```

The second data instance contains the data elements that the user fills out to order the products:

```
<xforms:instance id="po" xmlns="">
   <po>
      <order>
         <row>
            <product/>
            <unitCost>0</unitCost>
            <qty></qty>
            <lineTotal></lineTotal>
         </row>
      </order>
   <subtotal>0</subtotal>
   <tax>0</tax>
   <total>0</total>
</po>
```

This instance is linked to a *table* item in the form, which creates a table with four columns: product name, unit cost, quantity, and line total. The product name is chosen from a *popup* item that contains the *xforms:select1* shown below. When the user selects something from the popup, an *xforms-value-changed* event occurs. In this case, this event triggers two actions: first, the second column is automatically populated with the unit cost for that item (by getting that cost from the po instance); second, the focus is moved to the third column, since the second column has already been completed.

```
<xforms:select1 ref="product" appearance="minimal">
   <xforms:label>Choose product</xforms:label>
      <xforms:itemset nodeset="instance('products')/product">
         <xforms:label ref="@name"/>
         <xforms:value ref="@code"/>
         <xforms:extension>
            <value compute="label"/>
         </xforms:extension>
      </xforms:itemset>
      <xforms:setvalue ref="../unitCost" ev:event="xforms-value-changed"
         value="instance('products')/product[@code=instance('po')
         /order/row[index('orderTable')]/product]/@unitcost"/>
</xforms:select1>
```

Usage Details

1. This event only occurs on *xforms:select* and *xforms:select1* options that include an <xforms:itemset> element. If the option includes a list of <xforms:item> elements instead, use the *xforms-select* and *xforms-deselect* events.

Details on Function Calls

XFDL is an assertion-based language, which means a ″truth engine″ maintains statements in the code as true. The functions described in this section of the specification allow an XFDL form to perform procedural operations that would normally require complicated computations to achieve.

Function calls run code that may be external to the XFDL form definition. Below are the BNF rules for functions.

[46]
```
FunctionCall := (LibName '.')? FunctionName '(' (Compute
   (',' Compute)*)? ')'
```
[47]
```
LibName ::= sid
```
[48]
```
FunctionName ::= sid
```

The LibName allows functions to be grouped into separate namespaces, but the predefined functions in this specification do not require a LibName. (The LibName assigned to these predefined functions is *system*.) Any user-defined namespace must contain an underscore in its name.

Examples

Calling a predefined function (in the system namespace):

```
<custom:status xfdl:compute="toggle(field1.value, 'high', 'low')">
</custom:status>
```

or

```
<custom:status xfdl:compute="system.toggle(field1.value, &#xA;
   'high', 'low')"></custom:status>
```

Calling a user-defined function (in a custom namespace)

```
<value compute="hr_funcs.holiday(field1.value, field2.value)"></value>
```

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 subsituted for the reference, resulting in the following:

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

The function will then be computed.

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')
```

Usage Details on Using Functions

Position in Strings

The position of the first character in a string is at position zero. For example:

```
This is a string
```

The capital T in the string above is at position zero.

String Functions

countLines

Counts the number of lines that a string would take up over a given width, and returns the number of lines. The count assumes that the font is a monospaced font, and that the line will be wrapped at the ends of words, and not in the middle of words.

This function is useful if it is necessary to dynamically size items into which a string will be inserted. For example, to insert an entry from a database into a field on a form, dynamically size the height of the field so that all of the text is visible.

Note: The width must be a character-based width and not a pixel-based width.

Syntax

```
countLines(string, width)
```

string	*literal string or option reference*	the string to base the measurement on
width	*positive int*	the width, in monospaced characters, to base the measurement on

Returns

The number of lines, or "" (empty) if an error occurs.

Example

In this example, the field's height will be set by the number that *countLines* returns. The calculation is based on a dynamically-generated value, and the field's set width (50).

```
<field sid="commentField">
   <label>Comments</label>
   <itemlocation>
      <below>deptField</below>
   </itemlocation>
   <size>
      <width>50</width>
      <height compute="countLines(value, '50')"></height>
   </size>
</field>
```

countWords

Counts the number of words in a specified string.

Syntax

```
countWords(string)
```

string	*literal string or option reference*	the original string (enclose literal strings in double quotation marks, do not enclose option references in quotation marks)

Returns

The number of words in the original string, or nothing if an error occurs.

Example

In this example, *countWords* will return the value "5".

```
<field sid="Field1">
   <label>Test countWords()</label>
   <format>
```

```
      <datatype>string</datatype>
   </format>
   <value compute="countWords('Hello my name is Simon.')"></value>
</field>
```

pad

Pads or truncates an ASCII string to a specified length as explained:

- **Padding —** If the string is shorter than the specified length in the *pad* function then the string is padded with spaces. The pad_orientation parameter determines where the original string is oriented within the characters that make up the new **string**.
- **Truncating —** If the string is longer than the specified length in the *pad* function, than the string is truncated to the new length and any excess characters are lost. The pad_orientation parameter specifies what part of the original string is saved.

Syntax

```
pad(string, length, pad_orientation)
```

string	*literal string or option reference*	the original string to pad or truncate (enclose literal strings in double quotation marks, do not enclose option references in quotation marks)
length	*literal string or option reference*	length of the new string (enclose literal strings in double quotation marks, do not enclose option references in quotation marks)
pad_orientation	*literal string or option reference*	the position of the original string in the new padded or truncated string. (Enclose literal strings in double quotation marks, do not enclose option references in quotation marks.) This is an optional parameter. The valid choices are left, center, or right. The default value is left if the parameter is invalid or is not supplied.

Returns

The string padded or truncated to the specified length, or nothing if an error occurs.

Example

```
<field sid="Field1">
   <label>Test pad(): Center pad</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="pad('Hello','11','center')"></value>
</field>
```

```
<field sid="Field2">
   <label>Test pad(): Right pad</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="pad('Hello','10','right')"></value>
</field>
<field sid="Field3">
   <label>Test pad(): Right Truncate</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="pad('Hello','2','left')"></value>
</field>
```

In Field1 of the previous example, the *pad* function has the pad orientation center. Pad inserts 3 spaces on either side of the string ″Hello″ to create a new string which is ″11″ characters long and looks like the following:

```
"   Hello   "
```

In Field2 of the previous example, the *pad* function has the pad orientation right. Pad inserts 5 spaces at the beginning of the string ″Hello″ to create a new string which is ″10″ characters long and looks like the following:

```
"     Hello"
```

In Field3 of the previous example, the *pad* function has the pad orientation left and will truncate 4 characters from the end of the string ″Hello″ to create the new string ″He″ which is two characters long.

replace

Takes a string and replaces a substring in it (marked by start and end) with a new string. Returns the resulting string.

If start is less than 0 then the substring will begin on the first character of string. If end is greater than or equal to the length of string then the substring will end on the last character of string. If the new string is not long enough (that is, it does not reach position end), replacement will end with the last character of newString. If the new string is too long (that is, it extends past position end), replacement will end on position end.

An error occurs if start is greater than end, if either of start and end is not a valid integer, or if string is empty.

Syntax

```
replace(string, start, end, newString)
```

string	*literal string or option reference*	the original string (enclose literal strings in double quotation marks, do not enclose option references in quotation marks)

start	*int*	position of character at the start of the substring (the first character in *string* is zero)
end	*int*	position of character at the end of the substring (the first character in *string* is zero)
newString	*literal string or option reference*	the replacement string (enclose literal strings in double quotation marks, do not enclose option references in quotation marks)

Returns

The modified string, or ″″ (empty) if an error occurs.

Example

In this example, the result of *replace* is ″Go east, young man!″.

```
<field sid="replaceField">
   <label>Test replace()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value>Go west, young man!</value>
   <custom:change xfdl:compute="replace(value, '3', '6', "east")">
   </custom:change>
</field>
```

strlen

Returns the length of string.

Syntax

```
strlen(string)
```

string	*literal string or option reference*	the string (enclose literal strings in double quotation marks, do not enclose option references in quotation marks)

Returns

A string containing the length.

Example

In this example, the result of *strlen* is ″28″.

```
<field sid="stringLengthField">
   <label>The length of this label is:</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="strlen(label)"></value>
</field>
```

strmatch

Determines if the wildcard string *wild* matches the non-wildcard string *real* and returns the boolean result.

Syntax

```
strmatch(wild, real)
```

wild	*literal string or option reference*	the wildcard string to match (enclose literal strings in double quotation marks, do not enclose option references in quotation marks). Any of the following wild card characters can be used: **?** — represents any one (1) character ***** — represents any number of characters
real	*literal string or option reference*	the non-wildcard match string (enclose literal strings in double quotation marks, do not enclose option references in quotation marks)

Returns

A string containing ″1″ if a match occurs, ″0″ if no match occurs.

Example

In this example, the result of *strmatch* is ″1″.

```
   <field sid="testStrmatch">
      <label>Test strmatch()</label>
      <format>
         <datatype>string</datatype>
         <constraints>
            <mandatory>on</mandatory>
         </constraints>
      </format>
      <value>To be or not to be, etc.</value>
      <custom:change xfdl:compute="strmatch('?o be* ?o be*', value)"
         ></custom:change>
   </field>
```

strpbrk

Returns the position of the first character in *string1* that matches any of the characters in *string2*. Note that the count is zero based.

Syntax

```
strpbrk(string1, string2)
```

string1	*literal string or option reference*	the string (enclose literal strings in double quotation marks, do not enclose option references in quotation marks)
string2	*literal string or option reference*	the string of characters (enclose literal strings in double quotation marks, do not enclose option references in quotation marks)

Returns

A string containing the position, or ″-1″ if no matching characters are found.

Example

The result of *strpbrk*, displayed in ″FIELD2″ in the example below, is ″9″.

```
<field sid="testStrpbrk">
   <label>testField</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value>To be or not to be, etc.</value>
   <custom:change xfdl:compute="strpbrk(value, 'lLmMnNOpP')"></custom:change>
</field>
<field sid="FIELD2">
   <label>result field</label>
   <value compute="testStrpbrk.custom:change"></value>
</field>
```

strrstr

Returns the position of the first character of the last occurrence of *string2* in *string1*.

Syntax

```
strrstr(string1, string2)
```

string1	*literal string or option reference*	the string (enclose literal strings in double quotation marks, do not enclose option references in quotation marks)
string2	*literal string or option reference*	the substring (enclose literal strings in double quotation marks, do not enclose option references in quotation marks)

Returns

A string containing the position, or ″-1″ if no substring is found.

Example

The result of *strrstr*, displayed in ″FIELD2″ in the example below, is ″16″.

```
<field sid="testStrrstr">
   <label>testField</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value>To be or not to be, etc.</value>
   <custom:change xfdl:compute="strrstr(value, 'be')"
      ></custom:change>
</field>
<field sid="FIELD2">
   <label>result field</label>
   <value compute="testStrrstr.custom:change"></value>
</field>
```

strstr

Returns the position of the first character of the first occurrence of *string2* in *string1*.

Syntax

```
strstr(string1, string2)
```

string1	*literal string or option reference*	the string (enclose literal strings in double quotation marks, do not enclose option references in quotation marks)
string2	*literal string or option reference*	the substring (enclose literal strings in double quotation marks, do not enclose option references in quotation marks)

Returns

A string containing the position, or ″-1″ if no occurrence is found.

Example

The result of *strstr*, displayed in ″FIELD2″ in the example below, is ″3″.

```
<field sid="testStrstr">
   <label>testField</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
<value>To be or not to be, etc.</value>
   <custom:change xfdl:compute="strstr(value, 'be')"
      ></custom:change>
</field>
```

```
<field sid="FIELD2">
   <label>result field</label>
   <value compute="testStrstr.custom:change"></value>
</field>
```

substr

Returns the substring of *string* from the position indicated in *start* through the position indicated in *end*. If *start* is less than zero then the substring will begin on the first character of string. If *end* is greater than or equal to the length of string then the substring will end on the last character of string.

An error occurs if *start* is greater than *end*, if either of *start* and *end* is not a valid integer, or if string is empty.

Syntax

```
substr(string, start, end)
```

string	*literal string or option reference*	the string (enclose literal strings in double quotation marks, do not enclose option references in quotation marks)
start	*int*	position of character at the start of the substring (the first character in *string* is zero)
end	*int*	position of character at the end of the substring (the first character in *string* is zero)

Returns

The substring, or "" (empty) if an error occurs.

Example

The result of *substr*, displayed in "FIELD2" in the example below, is "Watso".

```
<field sid="surnameField">
   <label>Surname</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value>Watson</value>
   <custom:change xfdl:compute="substr(value, '0', '4')"
      ></custom:change>
</field>
<field sid="FIELD2">
   <label>result field</label>
   <value compute="surnameField.custom:change"></value>
</field>
```

tolower

Returns the lower case of *string*.

Syntax

```
tolower(string)
```

string	*literal string or option reference*	the original string (enclose literal strings in double quotation marks, do not enclose option references in quotation marks)

Returns

The lower case string.

Example

The result of *tolower*, shown in ″displayField″ in the example below, is ″hello!″.

```
   <field sid="tolowerField">
      <label>Test tolower()</label>
      <format>
         <datatype>string</datatype>
         <constraints>
            <mandatory>on</mandatory>
         </constraints>
      </format>
      <value>Hello!</value>
      <custom:change xfdl:compute="tolower(value)"></custom:change>
   </field>
   <field sid="displayField">
      <value compute="tolowerField.custom:change"></value>
  </field>
```

toupper

Returns the upper case of *string*.

Syntax

```
toupper(string)
```

string	*literal string or option reference*	the original string (enclose literal strings in double quotation marks, do not enclose option references in quotation marks)

Returns

The upper case string.

Example

The result of *toupper*, shown in ″displayField″ in the example below, is ″HELLO!″.

```
   <field sid="toupperField">
      <label>Test toupper()</label>
      <format>
         <datatype>string</datatype>
         <constraints>
            <mandatory>on</mandatory>
         </constraints>
      </format>
      <value>Hello!</value>
      <custom:change xfdl:compute="toupper(value)"></custom:change>
   </field>
   <field sid="displayField">
      <value compute="toupperField.custom:change"></value>
   </field>
```

trim

Returns a copy of *string* with all leading and trailing white space (blanks, tabs, newlines, carriage returns) removed.

Syntax

```
trim(string)
```

string	*literal string or option reference*	the original string (enclose literal strings in double quotation marks, do not enclose option references in quotation marks)

Returns

The string with leading and trailing whitespace removed.

Example

In this example, the result of *trim* is ″Test trim()″.

```
<field sid="trimField">
   <label>     Test trim()   </label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="trim(label)"></value>
</field>
```

URLDecode

Returns a URL-decoded version of *string*.

Syntax

```
URLDecode(string)
```

string	*literal string or option reference*	the original string (enclose literal strings in double quotation marks, do not enclose option references in quotation marks)

Returns

The URL-decoded string.

Example

In this example, the result of *URLDecode* is "This is a line".

```
<field sid="URLDecodeField">
   <label>Test URLDecode()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="URLDecode('This%20is%20a%20line%0D')"></value>
</field>
```

URLEncode

Returns a URL-encoded version of *string*.

Syntax

```
URLEncode(string)
```

string	*literal string or option reference*	the original string (enclose literal strings in double quotation marks, do not enclose option references in quotation marks)

Returns

The URL-encoded string.

Example

In this example, the result of *URLEncode* is "This+is+a+line%0A".

```
<field sid="URLEncodeField">
   <value compute="URLEncode('This is a line\n')"></value>
</field>
```

Math Functions

abs

Returns the absolute value of the number represented in *number*.

An error occurs if *number* is not a valid number.

Syntax

```
abs(number)
```

number	*decimal number*	a number

Returns

A string containing the absolute of the number, or "" if an error occurs.

Example

In this example, the result of *abs* is "2341.23".

```
<field sid="absTest">
   <label>Test abs()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="abs('-2341.23')"></value>
</field>
```

acos

Returns the arc cosine of a number stored in *number*.

An error occurs if *number* is not a valid number or has absolute value greater than 1.

Syntax

```
acos(number)
```

number	*decimal number*	a number

Returns

A string containing the arc cosine, or "" if an error occurs.

Example

In this example, the result of *acos* is "1.047198".

```
<field sid="arccosineField">
   <label>Test acos()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="acos('0.5')"></value>
</field>
```

annuity

Returns the present value *annuity factor* for an ordinary annuity, at a periodic interest rate indicated by *rate* over a number of periods specified in *periods*. (*Present value* is the lump sum to invest at rate in order to produce a set payment over periods. An *ordinary annuity* provides the payment at the end of each period specified in periods.)

This function might be used to figure out either:
- P, the present value (lump sum to invest).
- R, the periodic payment amount that will be received.

For reference:
- P = R * annuity_factor
- R = P / annuity_factor

An error occurs if *periods* is not a valid integer, or if rate is 0.

Syntax

```
annuity(rate, periods)
```

rate	*decimal number*	the rate of interest in decimal form compounded each period
periods	*integer*	the number of periods

Returns

A string containing the present value annuity factor, or "" if an error occurs.

Example

In this example, annuity returns "5.786373" and, if the desired payment entered into "paymentField" were $1, then the value of "presentValueInv" would be $5.78. (That is, a person would have to invest $5.78 at 5% for seven payments.)

```
   <field sid="presentValueInv">
      <label>The present value to invest is:</label>
      <format>
         <datatype>string</datatype>
         <constraints>
            <mandatory>on</mandatory>
         </constraints>
      </format>
      <value compute="paymentField.value * &#xA;
         annuity('.05', '7')"></value>
   </field>
```

asin

Returns the arc sine of a number stored in *number*.

An error occurs if *number* is not a valid number or has an absolute value greater than 1.

Syntax

```
asin(number)
```

number	*decimal number*	a number

Returns

A string containing the arc sine, or "" if an error occurs.

Example

In this example, the result of *asin* is "0.523599".

```
<field sid="arcsinField">
   <label>Test asin()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="asin('0.5')"></value>
</field>
```

atan

Returns the arc tangent of a number stored in *number*.

An error occurs if *number* is not a valid number.

Syntax

```
atan(number)
```

number	*decimal number*	a number

Returns

A string containing the arc tangent, or "" if an error occurs.

Example

In this example, the result of *atan* is "0.463648".

```
<field sid="arctangentField">
   <label>Test atan()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="atan('0.5')"></value>
</field>
```

ceiling

Returns the ceiling of a number.

An error occurs if *number* is not a valid number.

Syntax

```
ceiling(number)
```

number	*decimal number*	a number

Returns

A string containing the ceiling of the number, or "" if an error occurs.

Example

In this example, the result of *ceiling* is "-19".

```
<field sid="ceilingTest">
   <label>Test ceiling()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="ceiling('-19.6')"></value>
</field>
```

compound

Returns the compound interest factor at a rate indicated by *rate* over a number of periods specified in *periods*.

This might be used to calculate the total amount of a loan, by multiplying an original principle by the result of *compound*. See below for an example.

An error occurs if *periods* is not a valid integer.

Syntax

```
compound(rate, periods)
```

rate	*decimal number*	the rate of interest in decimal form compounded each period
periods	*integer*	the number of periods

Return

A string containing the compound interest factor, or "" if an error occurs.

Example

In this example, the result of *compound* is "1.948717". The value of the field is 1.948717 x the amount in the "principleField".

```
<field sid="totalAmountField">
   <label>Total Amount of Loan</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="principleField.value * &#xA;
      compound('.1', '7')"></value>
</field>
```

cos

Returns the cosine of an angle stored in *angle* and expressed in radians.

An error occurs if *angle* does not contain a valid angle.

Syntax

```
cos(angle)
```

angle	*decimal number*	the angle in radians

Returns

A string containing the cosine, or "" if an error occurs.

Example

In this example, the result of *cos* is "-0.416147".

```
<field sid="cosineField">
   <label>Test cos()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
```

```
            <mandatory>on</mandatory>
         </constraints>
      </format>
      <value compute="cos('2')"></value>
   </field>
```

deg2rad

Returns the number of radians in an angle expressed in degrees stored in *angle*.

An error occurs if *angle* does not contain a valid angle.

Syntax

```
deg2rad(angle)
```

angle	*decimal number*	the angle in degrees

Returns

A string containing the number of radians, or "" if an error occurs.

Example

In this example, the result of *deg2rad* is "2.00000".

```
   <field sid="deg2radField">
      <label>Test deg2rad()</label>
      <format>
         <datatype>string</datatype>
         <constraints>
            <mandatory>on</mandatory>
         </constraints>
      </format>
      <value compute="deg2rad('114.591559')"></value>
   </field>
```

exp

Returns the exponentiation of the number represented in *number* (i.e., e^{number}).

An error occurs if *number* is not a valid number.

Syntax

```
exp(number)
```

number	*decimal number*	a number

Returns

A string containing the exponentiation of the number, or "" if an error occurs.

Example

In this example, the result of *exp* is ″20.855369″.

```
<field sid="expTestField">
   <label>Test exp()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="exp('3')"></value>
</field>
```

fact

Returns the factorial value of the integer represented in *integer*.

An error occurs if *integer* is negative.

Syntax

```
fact(number)
```

integer	*integer*	a non-negative integer

Returns

A string containing the factorial of the integer, or ″″ if an error occurs.

Example

In this example, the result of *fact* is ″40320″.

```
<field sid="factTestField">
   <label>Test fact()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="fact('8')"></value>
</field>
```

floor

Returns the floor of the number represented in *number*.

An error occurs if *number* is not a valid number.

Syntax

```
floor(number)
```

number	*decimal number*	a number

Returns

A string containing the floor of the number, or "" if an error occurs.

Example

In this example, the result of floor is "-20".

```
<field sid="floorTestField">
   <label>Test floor()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="floor('-19.6')"></value>
</field>
```

ln

Returns the natural logarithm of the number represented in *number*.

An error occurs if *number* is not a decimal number greater than zero.

Syntax

```
ln(number)
```

number	*decimal number*	a number

Returns

A string containing the natural log of the number, or "" if an error occurs.

Example

In this example, the result of *ln* is "0".

```
<field sid="lnTestField">
   <label>Test ln()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="ln('1')"></value>
</field>
```

log

Returns the logarithm of the number represented in *number* to the base indicated by *base*.

An error occurs if either of *number* or *base* is not a valid number. The *number* must be equal to, or greater than 1.

Syntax

```
log(number, base)
```

number	*decimal number*	a number
base	*decimal number*	optional. A number representing the base for which the logarithm will be computed. If no base is supplied, a base of 10 is used.

Returns

A string containing the log of the number to the base, or "" if an error occurs.

Example

In this example, the result of *log* is "2".

```
<field sid="logTestField">
   <label>Test log()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="log('100', '10')"></value>
</field>
```

mod

Returns the modulus of the number represented in *number* using the divisor indicated by *divisor*.

An error occurs if either of *number* or *divisor* is not a valid number, or *divisor* is 0.

Syntax

```
mod(number, divisor)
```

number	*decimal number*	a number
divisor	*decimal number*	a number representing the divisor for which the modulus will be computed

Returns

A string containing the modulus, or "" if an error occurs.

Example

In this example, the result of *mod* is ″-0.200000″.

```
   <field sid="modTestField">
      <label>Test mod()</label>
      <format>
         <datatype>string</datatype>
         <constraints>
            <mandatory>on</mandatory>
         </constraints>
      </format>
      <value compute="mod('-3.5', '.3')"></value>
   </field>
```

pi

Returns the value of PI to 13 decimal places.

Syntax

```
pi()
```

Returns

A string containing the value of p.

Example

In this example, the result of *pi* is ″3.14159265359″ (precision is software-dependent).

```
<field sid="piTestField">
      <label>Test pi()</label>
      <format>
         <datatype>string</datatype>
         <constraints>
            <mandatory>on</mandatory>
         </constraints>
      </format>
      <value compute="pi()"></value>
</field>
```

power

Returns the number represented in *number* raised to the power indicated by *power*.

An error occurs if either of *number* or *power* is not a valid number.

Syntax

```
   power(number, power)
```

number	*decimal number*	a number
power	*decimal number*	a number representing the power by which the number will be raised

Returns

A string containing the number raised to the power, or "" if an error occurs.

Example

In this example, the result of *power* is "100.00000".

```
<field sid="powerTestField">
   <label>Test power()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="power('0.1', '-2')"></value>
</field>
```

rad2deg

Returns the number of degrees in an angle expressed in radians stored in *angle*.

An error occurs if *angle* does not contain a valid angle.

Syntax

```
rad2deg(angle)
```

angle	*decimal number*	the angle in radians

Returns

A string containing the number of degrees, or "" if an error occurs.

Example

In this example, the result of *rad2deg* is "114.591559".

```
<field sid="rad2degField">
   <label>Test rad2deg()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="rad2deg('2')"></value>
</field>
```

rand

Returns a random integer from the range of integers indicated by *lowerlimit* and *upperlimit*. (The range includes *lowerlimit* and *upperlimit*).

An error occurs if either of *lowerlimit* or *upperlimit* is not a valid integer, or *upperlimit* is less than *lowerlimit*.

Syntax

```
rand(lowerlimit, upperlimit)
```

lowerlimit	*integer*	the lower limit of the random number's range
upperlimit	*integer*	the upper limit of the random number's range

Returns

A string containing the random integer, or "" if an error occurs.

Example

In this example, the result of *rand* is an integer in the range [45,90].

```
<field sid="randTestField">
   <label>Test rand()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="rand('45', '90')"></value>
</field>
```

round

Returns the number represented in *number* rounded to the nearest decimal position indicated by *place* (e.g., 100, 10, 1, 0.1, ...). All numbers rounded to right of an *x/y* graph, so the result of rounding negative numbers goes the opposite way you would expect. See below for an example.

An error occurs if *number* is not a valid number or *place* is not a power of 10.

Syntax

```
round(number, place)
```

number	*decimal number*	a number
place	*decimal number*	a number representing the decimal place where number is to be rounded

Returns

A string containing the rounded number, or "" if an error occurs.

Examples

In this example, the result of *round* is ″323.2400″.

```
<field sid="roundTestField">
   <label>Test round()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="round('323.235', '.01')"></value>
</field>
```

In this example, the result of *round* is ″-323.2300″.

```
<field sid="roundTestField">
   <label>Test round()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="round('-323.235', '.01')"></value>
</field>
```

sin

Returns the sine of an angle stored in *angle* and expressed in radians.

An error occurs if *angle* does not contain a valid angle.

Syntax

```
sin(angle)
```

angle	*decimal number*	the angle in radians

Returns

A string containing the sine, or ″″ if an error occurs.

Example

In this example, the result of *sin* is ″0.909297″.

```
<field sid="sineField">
   <label>Test sin()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="sin('2')"></value>
</field>
```

sqrt

Returns the square root of the number represented in *number*.

An error occurs if *number* is a negative number.

Syntax

```
sqrt(number)
```

number	*decimal number*	a non-negative number

Returns

A string containing the square root, or "" if an error occurs.

Example

In this example, the result of *sqrt* is "4.415880".

```
<field sid="sqrtTestField">
   <label>Test sqrt()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="sqrt('19.5')"></value>
</field>
```

tan

Returns the tangent of an angle expressed in radians stored in *angle*.

An error occurs if *angle* does not contain a valid angle (for example, p/2, 3p/2, 5p/2, and so on).

Syntax

```
tan(angle)
```

angle	*decimal number*	the angle in radians

Returns

A string containing the tangent, or "" if an error occurs.

Example

In this example, the result of *tan* is "-2.185040".

```
<field sid="tanField">
   <label>Test tan()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="tan('2')"></value>
</field>
```

Utility Functions

applicationName

Returns the name of the application that is processing the form. This name must be set in the application so that it is available to the function call.

Syntax

```
applicationName()
```

Returns

A string containing the application name. For example, Workplace Forms products will return the following application names:

- **Workplace Forms Viewer —** Viewer
- **Workplace Forms Webform Server —** Webform Server

Example

In this example, if the application were being displayed by the Viewer, the result of *applicationName* would be ″Viewer″.

```
<field sid="appNameField">
   <label>Test applicationName()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="applicationName()"></value>
</field>
```

applicationVersion

Returns the version of the currently running application in the format ″MM.mm.TT″, where MM is the Major version number, mm is the minor version number, TT is the maintenance number.

Syntax

```
applicationVersion()
```

Returns

A string containing the application version.

Example

In this example, if running in an application of version 3.2.4, the result of *applicationVersion* would be "03.02.04".

```
<field sid="appVersionField">
   <label>Test applicationVersion()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="applicationVersion()"></value>
</field>
```

applicationVersionNum

Returns the decimal form of the version of the currently running application. This number is obtained from the hexadecimal format 0xMMmmTTPP, where MM is the Major version number, mm is the minor version number, TT is the maintenance number, and PP is the patch number. At this point, individual patches are not recognized in version numbers and so will always be 0.

Syntax

```
applicationVersionNum()
```

Returns

A string containing the application version number.

Example

In this example, if running in an application at version v3.2.4, the result of *applicationVersionNum* would be "50463744", which is the decimal representation of 0x03020400.

```
<field sid="appVersionNumField">
   <label>Test applicationVersionNum()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="applicationVersionNum()"></value>
</field>
```

checkValidFormats

This function checks the format of *all* items in the form and returns the number of items whose format is invalid. You can also set the function to create a list of the invalid items. Note that you cannot use this function to check the validity of XForms nodes, nor can you pass the results of this check to an item containing XForms.

To check the validity of a single item, use the *isValidFormat* function (see "isValidFormat" on page 330).

Syntax

```
checkValidFormats(reference, optionName)
```

reference	*reference string*	optional. A reference to an option will contain the list of invalid items. This option must be in the same item that the function is called from.
		
		For example, if you called the function from a *label* item you might create the following custom option:
		`<label sid="numberInvalid">`
		`   <custom:list/>`
		
		In this case, you would use the following reference:
		`custom:list`
		
		The function will create the option if it does not already exist. The function then populates the option with a list of references to the invalid items. For example, if two fields were invalid you would get a list like this:
		`<custom:list>`
		`   <custom:invalidref>Page1.Field1`
		`   </custom:invalidref>`
		`   <custom:invalidref>Page1.Field2`
		`   </custom:invalidref>`
		`</custom:list>`
		
		If this parameter is not specified in the call, the function still validates all form items but does not create a list of references to invalid items.

optionName	*string*	optional. Sets the tag that is used to store each item in the list of invalid items. This is useful if you need to set a particular namespace for the list. For example, if you store your list in a *custom:list* option, you might set the following tag: `custom:invalidref` This ensures that the list is stored in the same namespace as the containing element. Default: **xfdl:ae**

Returns

Returns the number of items that failed the validity check. If there are no invalid items, the function returns 0.

Example

The following sample form shows how you can use the *checkValidFormats* function to specify custom behaviors if a form with invalid values is submitted:

```
<?xml version="1.0"?>
<XFDL xmlns="http://www.ibm.com/xmlns/prod/XFDL/7.0"
   xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
   xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom">
   <globalpage sid="global">
      <global sid="global"></global>
   </globalpage>
   <page sid="PAGE1">
      <global sid="global"></global>
      <field sid="FIELD1">
         <format>
            <datatype>date</datatype>
            <constraints>
               <mandatory>on</mandatory>
            </constraints>
         </format>
      </field>
      <field sid="FIELD2">
         <value>4.00</value>
         <format>
            <datatype>dollar</datatype>
            <constraints>
               <mandatory>on</mandatory>
            </constraints>
         </format>
      </field>
      <button sid="BUTTON1">
         <value>Submit</value>
         <type>done</type>
         <url>http://localhost/cgi-bin/test.pl</url>
         <custom:checkIfValid compute=
            "toggle(BUTTON1.activated, 'off', 'on') == '1' ? &#xA;
            (checkValidFormats('custom:brokenOptions', &#xA;
            'custom:invalidref') != '0' ? &#xA;
            set('BUTTON1.activated', 'off') + &#xA;
            viewer.messageBox('Unable to send because at ' &#xA;
            +. 'least ' +. custom:brokenOptions[0] +. &#xA;
            'item is invalid.') : '') : ''"></custom:checkIfValid>
      </button>
   </page>
</XFDL>
```

The *format* option for "FIELD1" specifies that this item must contain a date and cannot be blank. If a user submits the form without correctly completing this field, the *checkValidFormats* function prevents the submission. Instead, the function creates a new option called "brokenOptions":

```
<custom:brokenOptions>
   <custom:invalidref>PAGE1.FIELD1</custom:invalidref>
</custom:brokenOptions>
```

The remaining code uses the information in brokenOptions to display a context-sensitive message to the user.

countChildren

Counts the number of children that belong to a form element.

Note that this count will include all global elements (such as global pages and global items) if they are children of the node. Additionally, it will include elements that are not written out with form but are included when the form is in memory, such as *activated*, *pageprevious*, *pagenext*, *itemprevious*, *itemnext*, and so on.

Syntax

```
countChildren(reference, referenceType, scheme)
```

reference	*reference string*	a reference to the element containing the children.
referenceType	*string*	*optional*. Identifies the type of reference used in the *reference* parameter. Valid settings are: form, page, item, option, and array. An array is anything below the option level in the form.
scheme	*string*	*optional*. The scheme used to write the reference. Defaults to *XFDL*. Requires the *referenceType* parameter.

Returns

An integer representing the number of children or "" if an error occurs.

Example

In this example, the **countChildren** function calculates the number of items on the page. Note that the compute subtracts one from this total to account for the global item.

```
<field sid="totalItems">
   <value compute="countChildren('Page1', 'page') - 1"></value>
</field>
```

countDatagroupItems

Returns the number of items in a particular datagroup.

Syntax

```
countDatagroupItems(datagroup)
```

datagroup	*string*	the name of the datagroup. This can include a page reference, such as *Page1.myGroup*. If it does not, the function searches for the group on the page that contains the function.

Returns

The number of items in the datagroup or "" if an error occurs.

Example

In this example, the field displays the number of items in a datagroup called *Data1*.

```
<field sid="totalCount">
   <label>The Green group contains this many items:</label>
   <value compute="countDatagroupItems('Data1')"></value>
</field>
```

countGroupedItems

Returns any of the following:

- The total number of items in a group.
- The total number of items in a group that have a particular option. For example, the number of items with a *bgcolor* setting in a group.
- The total number of items in a group that have a particular option setting. For example, the number of items with a *bgcolor* set to *blue* in a group.

Syntax

```
countGroupedItems(group, option, literal, groupContext, groupContextType,
referenceType, scheme)
```

group	*string*	the name of the group you want to get an item from. This can include a page reference, such as *Page1.myGroup*.
option	*string*	*optional*. If supplied, the function will only count items that contain this option.
literal	*string*	*optional*. If supplied, the function will only count items that have the specified option set to this value. Must be used with the *option* parameter.
groupContext	*string*	*optional*. The starting point to use to locate the group if the group name is not fully qualified. For example, if the group was on the first page, you would use *Page1*.
groupContextType	*string*	*optional*. The level of the group context parameter, such as *page*, *item*, or *option*. Currently only *page* is valid.
referenceType	*string*	*optional*. Sets the scope of the reference that is returned. The reference begins at the level below this. For example, to get a reference that begins at the page level, set this parameter to *form*. Valid settings are *form* and *page*. Defaults to *form*.
scheme	*string*	*optional*. The referencing scheme used. Defaults to *XFDL*.

Returns

The number of items that match the search criteria or "" if an error occurs.

Example

In this example, the field displays the number of items in a group called *Green*.

```
<field sid="totalCount">
   <label>The Green group contains this many items:</label>
   <value compute="countGroupedItems('Page1.Green')"></value>
</field>
```

decimal

Returns the decimal representation of the number represented by *number* with base indicated by *base*.

An error occurs if *number* is not a valid number, if *base* is not a valid positive integer base, or *number* cannot be resolved under the specified *base*.

Syntax

```
decimal(number, base)
```

number	*number*	a number
base	*positive integer*	an integer that is the base of the provided number

Returns

A string containing the decimal representation of the number, or "" if an error occurs.

Example

In this example, the result of *decimal* is "-74".

```
<field sid="decimalTestField">
   <label>Test decimal()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="decimal('-4A', '16')"></value>
</field>
```

destroy

Destroys (or deletes) specified elements from a form, including boxes, buttons, checks, comboboxes, fields, labels, lines, lists, pages, popups, radios, and spacers.

When a compute within the *destroy* function causes a change to the form, the form checks for other computes referenced by the *destroy* function and evaluates them immediately.

Syntax

```
destroy(reference, type)
```

reference	*reference string*	a reference to the element you want to destroy
type	*string*	the type of element being destroyed; types can be page, item, or option

Returns

Nothing if the operation was successful, or an error message if the reference cannot be destroyed.

Example

This example creates a button that, when clicked by the user, deletes ″FIELD1″ from ″PAGE2″ of the form.

```
<button sid="deleteField_BUTTON">
   <value>Delete</value>
   <custom:destroy xfdl:compute=
      "toggle(activated, 'off', 'on') == '1' ? &#xA;
      destroy('PAGE2.FIELD1', 'item') : ''"></custom:destroy>
</button>
```

Usage Details

1. If you destroy an entire page, the bindings to the XForms data model are automatically removed. However, this update does not occur if you only destroy an item or option within the page. This means that destroying items or options that are bound to the data model may cause your form to behave erratically.

duplicate

Makes a copy of a specified form element, places the copy in a designated location, and assigns it a new name. Computes in duplicate elements are evaluated immediately.

When creating elements for duplication, you will need to use relative tags (see “Function Call Syntax” on page 425). Also, see the *generateUniqueName* and *getReference* functions.

Syntax

```
duplicate(reference, type, newReference, newType, location, newName)
```

reference	*reference string*	a reference to the element you want to duplicate. For example, if you want to duplicate ″FIELD1″ on ″PAGE1″, the reference would be *PAGE1.FIELD1*.

type	*string*	the type of element being duplicated. Types can be page, item or option. For example, if you want to duplicate an element whose reference was *PAGE1.FIELD1*, the type would be item.
newReference	*reference string*	a reference that identifies where to put the new element. The new element is created as either a child or sibling of this element. For example, to create an element as a sibling of *FIELD1*, you would use: *PAGE1.FIELD1*.
newType	*string*	the type of the newReference. Types can be page, item, or option. For example, if the newReference is *PAGE2*, the newType would be page; if the newReference is *FIELD8*, the newType would be item.
location	*string*	a description of where to put the new element in relation to the newReference supplied in newType. Valid settings are: • **append_child** — adds the new element as the last child of the newType. • **after_sibling** — adds the new element as a sibling of the newType, placing it immediately after that node in the form structure. • **before_sibling** — adds the new element as a sibling of the newType, placing it immediately before that node in the form structure.
newName	*string*	a new name for the duplicate element

Returns

A duplicate element that contains the settings of the original element.

Example

In this example, the option custom:duplicate calls the *duplicate* function to make a copy of ″Field1″ on ″Page1″, which is identified as an *item* type. The call then places the duplicate on ″Page2″, which is identified as a *page* type, and places it immediately after the last item on ″Page2″ (append_child).

```
<button sid="duplicateFieldButton">
   <custom:duplicate xfdl:compute="toggle(activated, 'off', 'on') == '1' ? &#xA;
      duplicate('Page1.Field1', 'item', 'Page2', 'page', &#xA;
      'append_child', new_Name) : ''"></custom:duplicate>
</button>
```

Usage Details

1. If you duplicate an entire page, the elements in the new page will automatically bind to the XForms data model. However, this binding will not occur if you only duplicate an item or option within the page. This means that duplicating items or options that are bound to the data model may cause your forms to behave erratically.

forLoop

This function creates loop that you can use to run a compute a number of times.

The forLoop uses an option in your form (such as a custom option) as an index that stores the current count of the loop. As the loop counts, it sets this option to the current count. For example, if the for loop counted from 1 to 3, it would first set the option to 1, then to 2, then to 3. Each time the loop increments, it will perform a particular compute.

A form with a for loop will begin counting that loop as soon as the form opens, unless the loop itself relies on a triggering event, such as a *keypress* event or a *toggle* function.

Creating a Loop that Counts Once

You can create a loop that counts once by setting the initial count and the final count to be equal. For example, a loop that counts from 1 to 1 will count once.

Syntax

```
forLoop(indexReference, initialIndex, maxIndex, compute)
```

indexReference	*reference string*	a reference to the option that will store the current count of the for loop. If this option does not exist, the function creates it with an empty value.
initialIndex	*int*	the starting value of the for loop. This value is inclusive, meaning that the for loop will begin by counting this value.
maxIndex	*int*	the ending value of the for loop. This value in inclusive, meaning that the for loop will end by counting this value.
compute	*string*	the compute that you want to run each time the loop counts.

Returns

1 on success or 0 (zero) on failure.

Example

The following button contains computes that takes input from a field, calculates the factorial of that input, then sets another field with the result:

```
<button sid="calculateButton">
   <value>Click to calculate</value>
   <custom:toggle1 xfdl:compute="toggle( &#xA;
      calculateButton.activated, 'off', 'on') == '1' ? &#xA;
      set('resultField.value', '1') + &#xA;
      forLoop('custom:counter', '1', numberField.value, &#xA;
      set('resultField.value', resultField.value * &#xA;
      custom:counter) : ''"></custom:toggle1>
   <custom:counter></custom:counter>
</button>
```

The *custom:toggle1* option contains a compute that is triggered when a button in the form is clicked. Once triggered, the compute does three things: (1) it resets the result field to a value of 1, (2) it starts a for loop that runs from 1 to x, where x is a number typed in by the user, and (3) it runs a compute each time the count changes. This compute multiplies the value of the result field by the current count, then sets that value to the result field.

This effectively creates a loop that calculates the factorial of the number. On the first count, 1 * 1 is calculated and stored in the result field. On the second count, the value of the result field (1) * 2 is calculated and stored in the result field. On the third count, the value of the result field (2) * 3 is calculated, and stored in the result field. And so on, until the final value has been calculated and stored in the result field.

formatString

Returns a string formatted according to the rules set out in a referenced *format* option.

An error occurs if an invalid *format* is specified.

Syntax

```
formatString(reference, formatOptionReference)
```

reference	*reference string*	a reference to the string to reformat. For example, to format the value contained in ″Field3″, the reference would be *Field3.value*.
formatOptionReference	*string*	the format option used to apply to the reference. For example, to format the reference with the format options defined in ″Field1″, the string would be *Field1.format*.

Returns

The formatted string.

Example

In this example, the result of *formatString* in ″Field2″ is ″$30,095.60″.

```
<field sid="Field1">
   <label>Field 1</label>
   <format>
      <datatype>dollar</datatype>
      <presentation>
         <chowcurrency>on</showcurrency>
      </presentation>
   </format>
   <value></value>
</field>
<field sid="Field2">
   <label>Field 2</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="formatString(Field3.value, 'Field1.format')"></value>
```

```
</field>
<field sid="Field3">
   <label>Field 3</label>
   <value>30095.6</value>
</field>
```

In this example, *formatString* reformats a value as an integer and inserts it into *custom:value*.

```
<field sid="Field4">
   <value>$1.00</value>
   <format>
      <datatype>currency</datatype>
   </format>
   <custom:value xfdl:compute="formatString(value,
      'custom:format')"></custom:value>
   <custom:format>
      <datatype>integer</datatype>
   </custom:format>
</field>
```

generateUniqueName

Creates a unique name for an element, and is usually used with the *duplicate* function. The name is constructed using a prefix and an integer that are concatenated, as shown:

```
<prefix> +. <integer>
```

For example, consider a button that contains the code to copy a field (using the *duplicate* function). You could include the *generateUniqueName* function in this code. As a result, when a user clicks this button, the newly created field would have a unique name (such as ″newField_1″), which is composed of a designated prefix (″newField_″) and an automatically assigned integer (beginning at ″1″).

The function searches the form for any pages, items or options with the same name as the specified prefix. If the name already exists within the scope of the parent element, then the function will increment the integer until a unique name can be generated.

When generating unique names using this function, you will need to use the *set* function.

Syntax

```
generateUniqueName(reference, type, prefix)
```

reference	*reference string*	a reference to the parent element of the element for which you wish to generate a unique name. For example, if you want to generate a unique name for an item on ″PAGE1″, the reference would be *PAGE1*. To indicate that the parent element is the current form, leave this parameter blank. You would do this if you wanted to generate a unique name for a page.

type	*string*	the type of node identified in the reference. For example, if the reference is *PAGE1*, the type would be ″page″. Valid types include: • form • page • item • option • array
prefix	*string*	the prefix for the unique name generated. This can be any string you like. For example, if you choose ″newField_″ as the prefix, the first name generated by the function would be ″newField_1″, the second would be ″newField_2″, and so on.

Returns

A string that is composed of characters and an integer.

Example

In this example, the *generateUniqueName* function generates a unique name and returns the result newField_1.

```
<button sid="duplicateFieldButton">
   <value>duplicate Field1</value>
   <custom:name></custom:name>
   <custom:generate xfdl:compute=
      "toggle(activated, 'off', 'on') == '1' ? &#xA;
      set('custom:name', generateUniqueName('PAGE1', 'page', &#xA;
          'newField_')) : ''"></custom:generate>
</button>
```

get

Gets the value of either an XFDL option or an element in the XForms model.

Syntax

```
get(reference, referenceType, scheme)
```

reference	*reference string*	a reference to the string to get. If getting a string from an XFDL element, use XFDL referencing. For example, to retrieve the value of ″Field1″, the reference would be *Field1.value*. If getting a string from an element in the XForms data model, use XPath referencing. For example, to retrieve the value of the ZIP element, the reference might be *address/zip*.
referenceType	*string*	*optional*. Identifies the type of reference used in the *reference* parameter. To indicate a reference to an option node, use option. To indicate a reference to a node below the option level, use array. To indicate a reference to an XForms element, use empty string.

scheme	*string*	*optional*. The referencing scheme used. Use xfdl to refer to XFDL elements in the form, or xforms to refer to the XForms data model. If you need to refer to a particular data model, you must use a MIME type format instead, as shown: `application/xforms; model=`*`ID`* Set this to the ID of the model you want to work with. If you do not specify a model, the first model in the form is used. If the scheme is not provided, it defaults to xfdl. If you provide a scheme, you must also provide the *referenceType* parameter.

Returns

The value of the form option reference or "" if an error occurs.

Example

In this example, *get* retrieves the value of "Field2", which is "gold".

```
<field sid="Field1">
   <label>Field 1</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value>gold</value>
</field>
<field sid="Field2">
   <label>Test get()</label>
   <format>
      <datatype>string</datatype>
   </format>
   <value compute="get('Field1.value')"></value>
</field>
```

Usage Details

1. The *get* function does not support XPath referencing for the entire form. Use XPath references only when getting data from the XForms data model.
2. If you use the instance function to access a particular data instance, you must use the escape sequence for the quotations marks (&quot;) that appear around the parameter. For example:
   ```
   get('instance(&quot;loan&quot;)/Borrower/Name', '','xforms')
   ```
3. If you attempt to get the value of an element in the XForms data model, and that element is a parent (that is, it contains child elements), then the value of the first child text node is returned.
4. If you reference an item that changes, the change does not automatically trigger a re-evaluation of the function.

getAttr

Returns the value of an attribute on a form element.

Syntax

```
getAttr(reference, type, attrName, scheme)
```

reference	*reference string*	a reference to the element that has the attribute. For example, to get an attribute from a custom data element in *Field1*, you might use: `Page1.Field1.custom:data` All namespace prefixes are resolved relative to this node.
type	*string*	the type of reference used. This is one of *page*, *item*, *option*, or *array*.
attrName	*string*	the name of the attribute, including the appropriate namespace. For example: `prefix:attribute` Use *null* for the empty namespace, or nothing for the XFDL namespace. All other namespace prefixes are resolved relative to the *reference* node supplied.
scheme	*string*	*optional*. The referencing scheme used. Defaults to *XFDL*.

Returns

The value of the attribute or "" if an error occurs.

Example

In this example, *getAttr* retrieves the value of the *id* attribute from the <custom:data> element.

```
<field sid="nameField">
   <value>John B.</value>
   <custom:data id="12"></custom:data>
</field>
<field sid="idField">
   <value compute="getAttr('nameField.custom:data', 'option','custom:id')"/></value>
</field>
```

getDataByPath

Retrieves specific information from a signature in the form, such as the signer's name, the signer's e-mail address, and so on.

Syntax

```
getDataByPath(type, ref, path)
```

type	*string*	the type of object you are working with. This must be signature.
ref	*string*	a reference to the signature item in the form.

path	*string*	the path to the data you are retrieving from the signature. See the Usage Details below for further information.

Returns

A string containing the requested data.

Usage Details

About Data Paths

Data paths describe the location of information within a signature, just like file paths describe the location of files on a disk. You describe the path with a series of colon separated tags. Each tag represents either a piece of data, or an object that contains further pieces of data (just like directories can contain files and subdirectories).

For example, to retrieve the version of a signature, you would use the following data path:

```
Demographics
```

However, to retrieve the signer's common name, you first need to locate the signing certificate, then the subject, then finally the common name within the subject, as follows:

```
signingCert: Subject: CN
```

Some tags may contain more than one piece of information. For example, the issuer's organizational unit may contain a number of entries. You can either retrieve all of the entries as a comma separated list, or you can specify a specific entry by using a zero-indexed element number.

For example, the following path would retrieve a comma separated list:

```
signingCert: Issuer: OU
```

Adding an element number of 0 would retrieve the first organizational unit in the list, as shown:

```
signingCert: Issuer: OU: 0
```

signature Tags

The following list describes the tags available in a signature object. Note that Clickwrap and HMAC Clickwrap signatures have additional tags (detailed in *Clickwrap signature Tags* and *HMAC Clickwrap Tags*).

Engine
: The security engine used to create the signature.

signingCert
: The certificate used to create the signature. This is an object that contains further information, as detailed in *Certificate Tags*. Note that this object does not exist for Clickwrap or HMAC Clickwrap signatures.

HashAlg
: The hash algorithm used to create the signature.

CreateDate
The date on which the signature was created.

Demographics
A string describing the signature.

Clickwrap signature Tags

The following list describes additional tags available in both Clickwrap and HMAC Clickwrap signatures. Note that HMAC Clickwrap signatures have further tags (detailed in *HMAC Clickwrap Tags*).

TitleText
The text for the Windows title bar of the signature dialog box.

MainPrompt
The text for the title portion of the signature dialog box.

MainText
The text for the text portion of the signature dialog box.

Question1Text
The first question in the signature dialog box.

Answer1Text
The signer's answer.

Question2Text
The second question in the signature dialog box.

Answer2Text
The signer's answer.

Question3Text
The third question in the signature dialog box.

Answer3Text
The signer's answer.

Question4Text
The fourth question in the signature dialog box.

Answer4Text
The signer's answer.

Question5Text
The fifth question in the signature dialog box.

Answer5Text
The signer's answer.

EchoPrompt
Text that the signer must echo to create a signature.

EchoText
The signer's response to the echo text.

ButtonPrompt
The text that provides instructions for the Clickwrap signature buttons.

AcceptText
The text for the accept signature button.

RejectText
The text for the reject signature button.

Certificate Tags

The following list describes the tags available in a certificate object. Note that Clickwrap and HMAC Clickwrap signatures do not contain these tags.

Subject
: The subject's distinguished name. This is an object that contains further information, as detailed in *Distinguished Name Tags*.

Issuer
: The issuer's distinguished name. This is an object that contains further information, as detailed in *Distinguished Name Tags*.

IssuerCert
: The issuer's certificate. This is an object that contains the complete list of certificate tags.

Engine
: The security engine that generated the certificate. This is an object that contains further information, as detailed in *Security Engine Tags*.

Version
: The certificate version.

BeginDate
: The date on which the certificate became valid.

EndDate
: The date on which the certificate expires.

Serial
: The certificate's serial number.

signatureAlg
: The signature algorithm used to sign the certificate.

PublicKey
: The certificate's public key.

FriendlyName
: The certificate's friendly name.

Distinguished Name Tags

The following list describes the tags available in a distinguished name object. Note that Clickwrap and HMAC Clickwrap signatures do not contain these tags.

CN	The common name.
E	The e-mail address.
T	The title.
O	The organization.
OU	The organizational unit.
C	The country.
L	The locality.
ST	The state.
All	The entire distinguished name.

HMAC Clickwrap Tags

The following list describes the tags available in HMAC Clickwrap signature. Note that these tags are in addition to both the regular *signature Tags* and the *Clickwrap signature Tags*.

HMACsigner
: A string indicating which answers store the signer's ID.

HMACSecret
: A string indicating which answers store the signer's secret.

Notarization
: The notarizing signatures. This is one or more signature objects that contain further information, as detailed in *signature Tags*. There can be any number of notarizing signatures. Use an element number to retrieve a specific signature. For example, to get the first notarizing signature use:

```
Notarization: 0
```

If no element number is provided, the data will be retrieved from the first valid notarizing signature found. If no valid notarizing signatures are found, the function will return empty string.

Security Engine Tags

The following list describes the tags available in the security engine object:

Name The name of the security engine.

Help The help text for the security engine.

HashAlg
: A hash algorithm supported by the security engine.

Example

The following example shows a label that displays the signer's e-mail address once the form is signed. In this case, the label's *value* is set using an if/then test to determine whether the *signer* option on the signature button is set to anything other than an empty string. If it is, then the form has been signed, and *getDataByPath* is called to get the e-mail address of the signer.

```
<label sid="e-mailLabel">
   <value compute="Page1.sigButton.signer != '' ?
      getDataByPath('signature', 'Page1.usersignature',
      'signingCert: Subject: E') : ''></value>
<label>
```

getGroupedItem

Returns a reference to any of the following:
- The first item in a given group.
- The first item in a group that has a particular option. For example, the first item with a *bgcolor*.
- The first item in a group that has a particular option setting. For example, the first item with a *value* of on.

Syntax

```
getGroupedItem(group, option, literal, groupContext, groupContextType,
itemScopeType, scheme)
```

group	*string*	the name of the group you want to get an item from. This can include a page reference, such as *Page1.myGroup*.
option	*string*	*optional*. If supplied, the function will search for an item that has this option.
literal	*string*	*optional*. If supplied, the function will search for an item that has the specified option set to this value. Must be used with the *option* parameter. If the *empty* function is used as this parameter, *getGroupedItem* will search for an item that has no value set for the specified option.
groupContext	*string*	*optional*. The starting point to use to locate the group if the group name is not fully qualified. For example, if the group was on the first page, you would use *Page1*.
groupContextType	*string*	*optional*. The level of the group context parameter, such as *page*, *item*, or *option*. Currently only *page* is valid.
referenceType	*string*	*optional*. Sets the scope of the reference that is returned. The reference begins at the level below this. For example, to get a reference that begins at the page level, set this parameter to *form*. Valid settings are *form* and *page*. Defaults to *form*.
scheme	*string*	*optional*. The referencing scheme used. Defaults to *XFDL*.

Returns

A reference to the first item matching the search criteria or "" if an error occurs.

Usage Details

*getGroupedItem*supports the *empty* function as its third parameter. This allows *getGroupedItem* to return items that contain empty values in the specified option.

Example

In this example, *getGroupedItem* gets the sid of the radio button that is turned on.

```
<field sid="Field1">
   <label>The SID of the selected radio button is:</label>
   <value compute="getGroupedItem('Page1.radioGroup', 'value','on')"></value>
</field>
```

getInstanceRef

Returns a reference to a particular instance in the XML Data Model. You must know the ID of the instance.

Syntax

```
getInstanceRef(instanceID, scheme)
```

instanceID	*string*	the ID of the data instance.
scheme	*string*	*optional*. The referencing scheme used. Defaults to *XFDL*.

Returns

A string that contains a fully qualified reference to the data instance or "" if an error occurs.

Example

The following example uses to *getInstanceRef* to set the value of a label. The label will then display a reference to the *personnelInfo* instance.

```
<field sid="instanceReference">
   <value compute="getInstanceRef('personnelInfo')"></value>
</field>
```

getPosition

Returns the position index for an element within its parent.

For example, if a page contained two fields, and you called this function on the second field, it would return a value of "1", indicating that it was the second child (indexing is zero based).

Syntax

```
getPosition(reference, type, scheme)
```

reference	*reference string*	a reference to the element whose position you want to determine.
type	*string*	the type of reference used. This is one of *page*, *item*, *option*, or *array*.
scheme	*string*	*optional*. The referencing scheme used. Defaults to *XFDL*.

Returns

An integer representing the position of the element within its parent or "" if an error occurs.

Example

This example uses *getPosition* to determine the position index of the *lastNameField* on the first page. It then adds one to the index to change change it from zero based indexing to one based indexing, and concatenates that value into a string that reads: "The lastNameField is element #2 on page one."

```
<page sid="Page1">
   <field sid="firstNameField"></field>
   <field sid="lastNameField"></field>
</page>
<page sid="Page2">
   <field sid="positionField">
      <value compute="'The lastNameField is element #' +. &#xA;
         (getPosition('Page1.lastNameField', 'item') + 1) &#xA;
         +. ' on page one.'"></value>
   </field>
</page>
```

getPref

This returns the value of any setting in the Viewer Preferences form. For example, you could use this function to retrieve the user's e-mail address from the Preferences form.

Syntax

```
getPref(prefName)
```

prefName	*string*	the name of the Preferences setting to get. See the *Usage Details* below for a list of valid names.

Returns

A string containing the value of the Preferences setting or "" if an error occurs.

Usage Details

The following table lists shows the names for each setting in the Preferences form. Unless otherwise noted, each setting will have a value appropriate to the *value* option of the item type listed. For example, a field can be set to any string value, a check box can be set to *on* or *off*, and so on.

Preferences Type	**Setting**	**Name**
Basic	Network Access — Popup	networkAccess
	Try to locate browser automatically — Check Box	locateBrowser
	Path to Browser — Field	overrideDefaultPathTo Browser
	Use Default Simple MAPI Client — Check Box	useMAPI
	SMTP Server — Field	mailHost
	Return Address — Field	returne-mailAddress
	Use Enhanced Focus Indicator — Check Box	focusIndicator
	Use Operating System Colors — Check Box	useSystemColors
Input	Do Predictive Input Checking — Check Box	predictiveChecking

Preferences Type	Setting	Name
	I Prefer to Enter Dates — Radio Buttons Valid values are: *DDMMYY*, *MMDDYY*, and *YYMMDD*.	defaultDateFormat
	Stop Tab From Invalid Input Items — Check Box	forbidTaboutOnError
	Stop Tab From Empty Mandatory Items — Check Box	forbidTaboutOnMandatory
	Enable Smartfill — Check Box	smartfillEnabled
Printing	Print Radio Buttons as Check Boxes — Check Box	printRadiosAsChecks
	Print Radio Buttons Without Values — Check Box	printRadiosWithoutValues
	Print Scroll Bars on Fields — Check Box	printscroll barsOnFields
	Print single Line Fields as Lines — Check Box	printsingleLineFieldsAs Lines
	Print Border Around Form Edge — Check Box	printFormBorder
	Page Layout Defaults — Radio Buttons Valid values are: *fitToPage*, *shrinkToPage*, *tileOneDirection*, and *tileTwoDirections*.	printedPageLayout
	Print each page as separate print job — Check Box	printPageSeparately
	Print black and white (excluding images) — Check Box	printBlackAndWhite
Advanced	Show Boundary on All Form Items — Check Box	displayBoundingBoxes
	Use 'X' Style Check Boxes — Check Box Valid values are: *X* and *check*.	checkBoxStyle
	Scroll Fields on Zoom — Check Box	scrollFieldsOnZoom
	Digital Certificate Identity Filter — Field	certIdentifyFilter
	Path to Netscape Profile — Field	overrideDefaultPathTo NetscapeProfile
	Check CRL Distribution Points	checkCRLDistribution Points

Example

In this example, *getPref* is used to automatically populate a field with the user's e-mail address:

```
<field sid="e-mailAddress">
   <label>e-mail address:</label>
   <value compute="getPref('returne-mailAddress')"></value>
</field>
```

getReference

Returns a reference for the element that contains the call. The function works as a ″Where am I?″ check for a page, item, or option. It is always called from within the element for which the reference is needed, but the returned reference can be for that element or any of its parents.

For example, *getReference* could be called from within a page's label, and return a reference for that option (for example, *PAGE1.global.label*), its parent item (for example, *PAGE1.global*), or the page it is on (for example, *PAGE1*).

This function is especially useful when duplicating pages, since new pages will contain the identical options and items as their originals.

Syntax

```
getReference(element, type, level, scheme)
```

element	*string*	the element you want to identify. Possible values include the page sid, the item or option type, and the name or value of the item or option. For instance, to return the reference for a *label* option containing the call, the element parameter would be ″label″.
type	*string*	the element type, which can be page, item, option, or array. For example, the element type of a label would be option.
level	*string*	the level in the reference for the function to return. If the function is called from an option, but level is identified as *item*, the reference returned would be *page.item*.
scheme	*string*	*optional*. The referencing scheme used. Defaults to *XFDL*.

Returns

A reference to the element (page, item, or option) from which the function was called, or ″″ if an error occurs.

Example

In this example, *getReference* returns a reference to the page, which is *PAGE1*.

```
<page sid="PAGE1">
   <global sid="global">
      <label compute="getReference('label', 'option', 'page')"
         ></label>
   </global>
```

In this example, *getReference* returns a reference to the option of *label*, which in this case is *PAGE1.global.label*.

```
<page sid="PAGE1">
   <global sid="global">
      <label compute="getReference('label', 'option', 'option')"
         ></label>
   </global>
```

isValidFormat

Returns the boolean result of whether a string is valid according to the setting of the format option referred to in *formatOptionReference*. Note that you cannot use this function to check the validity of strings in items with XForms.

An error occurs if a non-existent format is specified.

Syntax

```
isValidFormat(string, formatOptionReference)
```

string	*string*	a string to be checked against the format. For example, to check 23.2 against a specific format, the string would be "23.2".
formatOptionReference	*string*	the option reference of the format, including the page sid if necessary, to check the string against. For example, to check 23.2 against a format specified in Field1, the formatOptionReference would be "Field1.format".

Returns

"1" if the string follows the format, "0" if not, or "" if an error occurs.

Example

In this example, the result of *isValidFormat* is "0" because the string to check contains a non-integer number representation and the specified format to check is of type integer.

```
<field sid="Field1">
   <label>Field 1</label>
   <format>
      <datatype>integer</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value>45</value>
</field>
<field sid="Field2">
   <label>Test isValidFormat()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="isValidFormat('23.2', 'Field1.format')"
      ></value>
</field>
```

set

Sets the value of an XFDL form option or of an element in the XForms model.

Syntax

```
set(reference, value, referenceType, scheme)
```

reference	*reference string*	a reference to the element to set.
		If setting an XFDL element, use XFDL referencing. For example, to set the value of ″Field1″, the reference would be *Field1.value*.
		If setting an elemement in the XForms data model, use XPath referencing. For example, to set the value of the ZIP element, the reference might be *address/zip*.
		a reference to the option to set. For example, to set the value of ″Field1″, the reference would be *Field1.value*; to set the value of a check box, the reference would be *CHECK2.value*. Note that this reference must be contained in quotes.
value	*string*	the reference's new value. For example, to set the reference's new value to silver, the value would be silver; to set the value of a checkmark to off, the value would be off.
referenceType	*string*	*optional*. Identifies the type of reference used in the *reference* parameter. To indicate a reference to an option node, use option. To indicate a reference to a node below the option level, use array. To indicate a reference to the XForms data model, use an empty string.
scheme	*string*	*optional*. The referencing scheme used. Use xfdl to refer to XFDL elements in the form, or xforms to refer to the XForms data model.
		If you need to refer to a particular data model, you must use a MIME type format instead, as shown: `application/xforms; model=ID`
		Set this to the ID of the model you want to work with. If you do not specify a model, the first model in the form is used.
		If the scheme is not provided, it defaults to xfdl. If you provide a scheme, you must also provide the *referenceType* parameter.

Returns

″1″ if the operation completed successfully, ″0″ if an error occurred. An error occurs if the specified form option could not be set to the specified value.

Example

In this example, the result of *set* is ″1″ and the value of Field1 is set to ″silver″.

```
<field sid="Field1">
   <label>Field 1</label>
   <value>gold</value>
```

```
</field>
<field sid="Field2">
   <label>Test set()</label>
   <value compute="set('Field1.value', 'silver')"></value>
</field>
```

In this example, if a form user selects the ″CHECK1″ check box (thereby turning its value on), the *set* function sets the value of ″CHECK2″ to off.

```
<check sid="CHECK1">
   <custom:set xfdl:compute="value == 'on' ? &#xA;
      set('CHECK2.value', 'off') : ''"></custom:set>
   <value>off</value>
</check>
<check sid="CHECK2">
   <custom:set xfdl:compute="value == 'on' ? &#xA;
      set('CHECK1.value', 'off') : ''"></custom:set>
   <value>off</value>
</check>
```

Usage Details

1. The *set* function does not support XPath referencing for the entire form. Use XPath references only when setting data in the XForms model.
2. If the option you are setting does not exist, the *set* function will create it so long as the containing page and item already exist. The *set* function will not create elements in the XForms model.
3. If you set the value of an XFDL option that has a compute, the compute is destroyed.
4. You can use the set function to create ″grouped″ check boxes. For example, if a form user selects one check box (thereby turning its value on), the *set* function can turn the value of another check box off.
5. If you use a *set* function to turn radio button on (or off), you must also use a set function to turn all of the other radio buttons in the same group off (or on).
6. When a compute within the *set* function causes a change to the form, the form checks for other computes referenced by the *set* function and evaluates them immediately.
7. If you use the *instance* function to access a particular data instance, you must use the escape sequence for the quotations marks (&quot;) that appear around the parameter. For example:
   ```
   set('instance(&quot;loan&quot;)/Borrower/Name', 'Bill Smith',
      '','xforms')
   ```
8. The *set* function does not work on values that are set to be readonly in the XForms model; however, it does work on values that are set to be readonly through the *readonly* option.
9. If you attempt to set the value of an element in the XForms data model, and that element is a parent (that is, it contains child elements), then the value of the first child text node is set.
10. For security reasons, the *set* function is not allowed to change the value of an option or suboption in an XForms-associated item (that is, an XFDL item that contains an XForms control or an XFDL item that has been created by an xforms:group, xforms:switch, xforms:repeat, xforms:select1 or xforms:select.

setAttr

Sets the value of an attribute on a form element.

Syntax

```
setAttr(reference, type, attrName, value, scheme)
```

reference	*reference string*	a reference to the element that has the attribute. For example, to set an attribute from a custom data element in *Field1*, you might use: `Page1.Field1.custom:data` All other namespace prefixes are resolved relative to this node.
type	*string*	the type of reference used. This is one of *page*, *item*, *option*, or *array*.
attrName	*string*	the name of the attribute, including the appropriate namespace. For example: `prefix:attribute` Use *null* for the empty namespace, or nothing for the XFDL namespace. All other namespace prefixes are resolved relative to the *reference* node supplied.
value	*string*	the value to assign to the attribute.
scheme	*string*	*optional*. The referencing scheme used. Defaults to *XFDL*.

Returns

″1″ if the operation completed successfully or ″0″ if an error occurred.

Example

In this example, s*etAttr* sets the value of the *id* attribute in the <custom:data> element to 12.

```
<field sid="nameField">
   <value>John B.</value>
   <custom:data id=""></custom:data>
</field>
<field sid="idField">
   <value compute="setAttr('nameField.custom:data', 'option',
      'id', '12')"></value>
</field>
```

Usage Details

1. Do not use the *setAttr* function to change the value of a single node or nodeset binding. This change will not be respected.
2. In the case of an *xforms:repeat*, you can use the *setAttr* function to change the nodeset binding. This allows you to copy a table to a new page and reset the binding of that table (for cases in which tables needs to wrap to a new page).
3. Do not use *setAttr* to modify the *action* attribute of a submission.

toggle

Monitors a specific form option and detects any changes to the value of that option. This function can detect any change in an option, or can watch for specific changes. For example, you can create a toggle that detects when an option changes from on to off, or from Fred to Jim.

An error occurs if the specified form option does not exist.

Syntax

```
toggle(reference, start, end)
```

reference	*reference string*	a reference to the option to watch. For example, to watch the value of a check box called ″noChoiceAllowed″, the reference would be *noChoiceAllowed.value.*
start	*string*	*optional*. The start condition for toggle. For example, you would set this to off to monitor when a check box is checked (the check box will toggle from off to on when it is checked by the user).
end	*string*	*optional*. The end condition for toggle. For example, you would set this to on to monitor when a check box is checked (the check box will toggle from off to on when it is checked by the user).

Returns

″1″ if the specified change occurs in the specified option, or ″0″ if another change occurs.

Example

In this example, *toggle* monitors a specific option for any change. Every time the value of ″nameField″ changes, toggle will return ″1″, and then a new time will be entered into ″timeStampField″, using the *now* function.

```
<field sid="timeStampField">
   <value compute="toggle(nameField.value) == '1' ? now() : ''"
      ></value>
   <label>Time Stamp</label>
   <readonly>on</readonly>
</field>
<field sid="nameField">
   <label>Name</label>
   <value></value>
</field>
```

In this example, *toggle* monitors a check box to determine if the check box is checked. If the value of the check box goes from off to on, the value of the label will change to ″The box has been checked.″

```
<label sid="checkStatusLabel">
   <value compute="toggle(check1.value, 'off', 'on') == '1' ? &#xA;
      'The box has been checked.' : &#xA;
      'The box has not been checked.'"></value>
</label>
<check sid="check1">
   <value>off</value>
</check>
```

xforms.getPosInSet

Returns an index that indicates the position in a set. For example, for a table item this determines which row of the table the compute is in. For a group (*checkgroup* or *radiogroup*), this determines which item in the group the compute is in.

This function is part of the xforms package of functions, and must include the ″xforms.″ prefix.

Syntax

```
xforms.getPosInSet()
```

Returns

An integer representing the position in the set. The integer is one-based. This means that the first element/row returns a value of 1, the second a value of 2, and so on.

Example

The following checkgroup uses the *xforms.getPosInSet* and *xforms.getSizeOfSet* functions to arrange the checks in two equal length columns. To achieve this, the x and y coordinates are computed for each item in the group as shown:

```
<checkgroup sid="color">
   <xforms:select ref="color" appearance="full">
      <xforms:label>Select the colors you like:</xforms:label>
      <xforms:itemset nodeset="../choice">
         <xforms:label ref="@show"></xforms:label>
         <xforms:value ref="."></xforms:value>
         <xforms:extension>
            <itemlocation>
               <x compute="floor((xforms.getPosInSet() - '1') / &#xA;
                  (ceiling(xforms.getSizeOfSet() / '2'))) * '60'"/>
               <y compute="(xforms.getPosInSet() - '1') % &#xA;
                  (ceiling(xforms.getSizeOfSet() / '2')) * '20'"/>
               </itemlocation>
            </xforms:extension>
         </xforms:itemset>
      </xforms:select>
   </checkgroup>
```

To calculate the x coordinate, the following algorithm is used:

1. Calculate the size of the set, divide this by two, then get the ceiling of that value.
   - This determines the length of the first column (which is always longer if there is an odd number of items).
2. Determine the position of the item in the set and subtract one.
   - This returns a zero-based position in the set.

3. Divide the position in the set by the length of the first row, then get the floor of this value.
   - This returns a zero if the position is less than the length of the first row, or a one if the position is equal to or greater than the length of the first row. This works because the set is zero-based, so the first five items (0-4) will return a zero since they are all less than 5.
4. Multiply by 60.
   - This returns an x coordinate of zero if the item is in the first column, or an x coordinate of 60 if the item is in the second column, effectively indenting the second column.

To calculate the y coordinate, the following algorithm is used:
1. Calculate the size of the set, divide this by two, then get the ceiling of that value.
   - This determines the length of the first column (which is always longer if there is an odd number of items).
2. Determine the position of the item in the set and subtract one.
   - This returns a zero-based position in the set.
3. Get the modulus of the position in the set divided by the length of the first row.
   - This returns zero for the first item, one for the second, two for the third, and so on. When the end of the first row is reached, the modulus begins again at zero.
4. Multiply by 20.
   - This determines the y coordinate, so the first item has a y coordinate of zero, the second a y coordinate of 20, and so on. The second row resets at zero and begins the count again.

xforms.getSizeOfSet

Returns the size of a set. For example, for a table item this determines how many rows are in the table. For a group (*checkgroup* or *radiogroup*), this determines how many items are in the group.

This function is part of the xforms package of functions, and must include the ″xforms.″ prefix.

Syntax

```
xforms.getPosInSet()
```

Returns

An integer representing the size of the set.

Example

The following checkgroup uses the *xforms.getPosInSet* and *xforms.getSizeOfSet* functions to arrange the checks in two equal length columns. To achieve this, the x and y coordinates are computed for each item in the group as shown:

```
<checkgroup sid="color">
   <xforms:select ref="color" appearance="full">
      <xforms:label>Select the colors you like:</xforms:label>
      <xforms:itemset nodeset="../choice">
         <xforms:label ref="@show"></xforms:label>
         <xforms:value ref="."></xforms:value>
         <xforms:extension>
            <itemlocation>
               <x compute="floor((xforms.getPosInSet() - '1') / &#xA;
                  (ceiling(xforms.getSizeOfSet() / '2'))) * '60'"/>
               <y compute="(xforms.getPosInSet() - '1') % &#xA;
                  (ceiling(xforms.getSizeOfSet() / '2')) * '20'"/>
               </itemlocation>
            </xforms:extension>
         </xforms:itemset>
      </xforms:select>
   </checkgroup>
```

To calculate the x coordinate, the following algorithm is used:

1. Calculate the size of the set, divide this by two, then get the ceiling of that value.
   - This determines the length of the first column (which is always longer if there is an odd number of items).
2. Determine the position of the item in the set and subtract one.
   - This returns a zero-based position in the set.
3. Divide the position in the set by the length of the first row, then get the floor of this value.
   - This returns a zero if the position is less than the length of the first row, or a one if the position is equal to or greater than the length of the first row. This works because the set is zero-based, so the first five items (0-4) will return a zero since they are all less than 5.
4. Multiply by 60.
   - This returns an x coordinate of zero if the item is in the first column, or an x coordinate of 60 if the item is in the second column, effectively indenting the second column.

To calculate the y coordinate, the following algorithm is used:

1. Calculate the size of the set, divide this by two, then get the ceiling of that value.
   - This determines the length of the first column (which is always longer if there is an odd number of items).
2. Determine the position of the item in the set and subtract one.
   - This returns a zero-based position in the set.
3. Get the modulus of the position in the set divided by the length of the first row.
   - This returns zero for the first item, one for the second, two for the third, and so on. When the end of the first row is reached, the modulus begins again at zero.
4. Multiply by 20.
   - This determines the y coordinate, so the first item has a y coordinate of zero, the second a y coordinate of 20, and so on. The second row resets at zero and begins the count again.

xforms.updateModel

This function updates the XForms model in the form. In general, the model is automatically updated by the forms viewing application when required. However, this function has been added for completeness.

This function is part of the xforms package of functions, and must include the ″xforms.″ prefix.

Syntax

```
xforms.updateModel(id)
```

id	*string*	optional. The id of the model you want to update. If no id is provided, the first model is updated.

Returns

″1″ if the operation completed successfully or ″0″ if an error occurred.

Example

The following checkgroup uses the *xforms.getPosInSet* and *xforms.getSizeOfSet* functions to arrange the checks in two equal length columns. To achieve this, the x and y coordinates are computed for each item in the group as shown:

```
<checkgroup sid="color">
```

xmlmodelUpdate

This function updates the XML data model in the form. This is useful if computes have changed the structure of the data model in some way, such as changing or adding bindings. These sorts of changes do not take effect until the *xmlmodelUpdate* function is called.

Syntax

```
xmlmodelUpdate()
```

Returns

″1″ if the operation completed successfully or ″0″ if an error occurred.

Example

The following XML data model has two instances for customer data. In this case, the second data instance is bound to the form, linking first name and last name.

```
<xmlmodel>
   <instances>
      <xforms:instance xmlns="http://www.w3.org/2003/xforms">
         <customers>
            <customerData>
               <firstName></firstName>
               <lastName></lastName>
            </customerData>
            <customerData>
               <firstName></firstName>
               <lastName></lastName>
            </customerData>
         <customers>
      <xforms:instance>
   </instances>
   <bindings>
      <bind>
         <ref>[customers][1][firstName]</ref>
         <boundOption>Page1.firstNameField.value</boundOption>
      </bind>
      <bind>
         <ref>[customers][1][lastName]</ref>
         <boundOption>Page1.lastNameField.value</boundOption>
      </bind>
   </bindings>
</xmlmodel>
```

The following button changes the data model so that the form elements are bound to the first data instance rather than the second. When the button is clicked, the *toggle* is triggered, which uses two *set* functions to change the *boundOption* elements in the data model. It also calls the *xmlmodelUpdate* function to ensure that the changes take effect immediately.

```
<button sid="updateDataModel">
   <value>Click to Update Data Model</value>
   <type>select</type>
   <custom:update compute=
      "toggle(activated, 'off', 'on') == '1' &#xA;
      ? set(global.global.xmlmodel[bindings][0][ref], &#xA;
      '[customers][0][firstName]') + &#xA;
      set(global.global.xmlmodel[bindings][1][ref], &#xA;
      '[customers][0][lastName]') + xmlmodelUpdate() &#xA;
      : ''"></custom:update>
</button>
```

xmlmodelValidate

Validates the XML Data Model against the available schemas.

A form may contain schemas as part of the XML Data Model, or may link to external schemas. In either case, the active schemas are listed in the *schema* attribute on the *xmlmodel* element. Only those schemas listed in the *schema* attribute are used to validate data.

Syntax

```
xmlmodelValidate()
```

Returns

The schema error message if the validation fails or an empty string if the validation is successful.

Example

The following example creates a Submit button in the form. When the user clicks the button, the *toggle* function triggers the *xmlmodelValidate* function, which validates the data.

```
<button sid="submitForm">
   <value>Submit</value>
   <type>done</type>
   <custom:results></custom:results>
   <custom:opt xfdl:compute="toggle(activated, 'off', 'on') &#xA;
      ==    '1' ? &#xA;
      set('custom:results', xmlmodelValidate()) &#xA;
      + (strlen(custom:results) > '0' &#xA;
      ? viewer.messageBox(custom:results) &#xA;
      + set('activated', 'off') &#xA;
      : '')    : ''"></custom:opt>
</button>
```

Time and Date Functions

date

Returns a date in ″yyyymmdd″ format. Either converts a number of seconds (from 12 am, January 1, 1970) or returns the current date if no value is provided.

Syntax

```
date(datesecs)
```

datesecs	*integer*	optional. The number of seconds from 12 am, January 1, 1970. If no value is provided, the current date is returned.

Returns

A string containing the current date, or the date specified by the a date represented by the number of seconds since 00:00:00 GMT, January 1, 1970

Example

In this example, if run on January 18th, 1998, the result of *date* is ″19980118″.

```
<field sid="dateTestField">
   <label>Test date()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="date()"></value>
</field>
```

dateToSeconds

Returns the number of seconds from the GMT date and time (represented in date and time respectively) since 00:00:00 GMT, January 1st, 1970.

When passing a *date* parameter to the function *dateToSeconds*, the date should be passed in a format used by XFDL. See the *format* option description for valid date formats.

Note: The best way to ensure that you pass a date in a valid format is to enter the date in a field, label or cell that has an XFDL format option assigned to it. See the second example below.

When passing a *time* parameter to the function *dateToSeconds*, the time may consist only of hours:minutes. You may use a 24-hour clock (for example, 23:34) or a 12-hour clock with A.M and P.M. (11:34 P.M.) designators. *Time* is an optional parameter.

An error occurs if either of *date* or *time* is not well formed.

To call the reverse of this function use the *date* function.

Syntax

```
dateToSeconds(date, time, reference)
```

date	*string*	a date in a recognized format (see “format” on page 90 for a list of formats)
time	*string*	*optional*. A time (ending in minutes) in a recognized format (for example, 23:34 or 11:34 P.M.)
reference	*reference string*	*optional*. A reference to an item that contains the format option to use when interpreting the date. If no format is provided, the function makes a best guess when interpreting the date.

Returns

A string containing the number of seconds, or "" if an error occurs.

Example

In this example, the result of *dateToSeconds* is "890329140".

```
<field sid="dtsField">
   <value compute="dateToSeconds('1998-03-19', '09:39')"></value>
</field>
```

The following example shows how to pass in a date that is set in another field.

```
<field sid="enterDateField">
   <format>
      <datatype>date</datatype>
      <presentation>
         <style>long</style>
      </presentation>
   </format>
```

```
    <value></value>
  </field>
  <field sid="dtsField">
    <value compute="dateToSeconds(enterDateField.value, '09:39')"
      ></value>
  </field>
```

In this example, the first field takes a date as input, and formats it in XFDL's long format. The second field calls the *dateToSeconds* function, and uses an option reference as a parameter (*enterDateField.value*). This reference takes the already-formatted date that the user enters, and passes it into the function.

day

Returns the numeric day of the month for the provided date in *dateSecs* or the current date if one is not provided. The provided date is a string representing the number of seconds since 00:00:00 GMT, January 1st, 1970.

An error occurs if *dateSecs* is not well formed.

Syntax

```
day(dateSecs)
```

dateSecs	*number*	optional. A date represented by the number of seconds since 00:00:00 GMT, January 1, 1970. If no value is supplied, the current date is used.

Returns

A string containing the day, or "" if an error occurs.

Example

In this example, the result of *day* is "19".

```
  <field sid="dayTestField">
    <label>Test day()</label>
    <format>
      <datatype>string</datatype>
      <constraints>
        <mandatory>on</mandatory>
      </constraints>
    </format>
    <value compute="day('890300356')"></value>
  </field>
```

dayOfWeek

Returns the numeric day of the week (Sunday=1, and so on) for the provided date in *dateSecs* or the current date if one is not provided. The provided date is a string representing the number of seconds since 00:00:00 GMT, January 1st, 1970.

An error occurs if *dateSecs* is not well-formed.

Syntax

```
dayOfWeek(dateSecs)
```

dateSecs	*number*	optional. A date represented by the number of seconds since 00:00:00 GMT, January 1st, 1970. If no value is supplied, the current date is used.

Returns

A string containing the day of the week, or "" if an error occurs.

Example

In this example, the result of *dayOfWeek* is "5".

```
<field sid="dowTestField">
   <label>Test dayOfWeek()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="dayOfWeek('890300356')"></value>
</field>
```

endOfMonth

Returns the number of seconds since 00:00:00 GMT, January 1st, 1970 to the current time on the last day of the month in the date provided in *dateSecs* or the current date if one is not provided. The provided date is a string representing the number of seconds since 00:00:00 GMT, January 1st, 1970.

An error occurs if *dateSecs* is not well-formed.

Syntax

```
endOfMonth(dateSecs)
```

dateSecs	*number*	optional. A date represented by the number of seconds since 00:00:00 GMT, January 1st, 1970. If no value is supplied, the current date is used.

Returns

A string containing the number of seconds, or "" if an error occurs.

Example

In this example, the result of *endOfMonth* is "891337156".

```
<field sid="eomTestField">
   <label>Test endOfMonth()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="endOfMonth('890300356')"></value>
</field>
```

hour

Returns the numeric hour for the provided hour in *dateSecs* or the current hour if one is not provided. If using *dateSecs*, the provided date is a string representing the number of seconds since 00:00:00 GMT, January 1st, 1970.

An error occurs if *dateSecs* is not well-formed.

Syntax

```
hour(dateSecs)
```

dateSecs	*number*	optional. A date represented by the number of seconds since 00:00:00 GMT, January 1st, 1970. If no value is supplied, the current date is used.

Returns

A string containing the hour, or "" if an error occurs.

Example

In this example, the result of *hour* is "9".

```
<field sid="hourTestField">
   <label>Test hour()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="hour('890300356')"></value>
</field>
```

minute

Returns the numeric minute for the provided date in *dateSecs* or the current date if one is not provided. The provided date is a string representing the number of seconds since 00:00:00 GMT, January 1st, 1970.

An error occurs if *dateSecs* is not well formed.

Syntax

```
minute(dateSecs)
```

dateSecs	*number*	optional. A date represented by the number of seconds since 00:00:00 GMT, January 1st, 1970. If no value is supplied, the current date is used.

Returns

A string containing the minute, or "" if an error occurs.

Example

In this example, the result of *minute* is "39".

```
<field sid="minuteTestField">
   <label>Test minute()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="minute('890300356')"></value>
</field>
```

month

Returns the numeric month of the year for the provided date in *dateSecs* or the current date if one is not provided. The provided date is a string representing the number of seconds since 00:00:00 GMT, January 1st, 1970.

An error occurs if *dateSecs* is not well formed.

Syntax

```
month(dateSecs)
```

dateSecs	*number*	optional. A date represented by the number of seconds since 00:00:00 GMT, January 1st, 1970. If no value is supplied, the current date is used.

Returns

A string containing the month, or "" if an error occurs.

Example

In this example, the result of *month* is "3".

```
<field sid="monthTestField">
   <label>Test month()</label>
   <format>
```

```
         <datatype>string</datatype>
         <constraints>
            <mandatory>on</mandatory>
         </constraints>
      </format>
      <value compute="month('890300356')"></value>
   </field>
```

now

Returns the number of seconds since 00:00:00 GMT, January 1st, 1970.

Syntax

```
now()
```

Returns

A string containing the number of seconds.

Example

In this example, if run at 09:39:16 GMT on Thursday, March 19th, 1998, the result of *now* would be ″890300356″.

```
<field sid="nowTestField">
   <label>Test now()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="now()"></value>
</field>
```

second

Returns the numeric second for the provided date in *dateSecs* or the current date if one is not provided. The provided date is a string representing the number of seconds since 00:00:00 GMT, January 1st, 1970.

An error occurs if *dateSecs* is not well formed.

Syntax

```
second(dateSecs)
```

dateSecs	*number*	optional. A date represented by the number of seconds since 00:00:00 GMT, January 1st, 1970. If no value is supplied, the current date is used.

Returns

A string containing the second, or ″″ if an error occurs.

Example

In this example, the result of *second* is ″16″.

```
<field sid="secondTestField">
   <label>Test second()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="second('890300356')"></value>
</field>
```

time

Returns the current time in ″hh:mm AM″ format.

Syntax

```
time()
```

Returns

A string containing the current time.

Example

In this example, if run at 3:22 in the afternoon, the result of *time* would be ″3:22 PM″.

```
<field sid="timeTestField">
   <label>Test time()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="time()"></value>
</field>
```

year

Returns the numeric year for the provided date in *dateSecs* or the current date if one is not provided. The provided date is a string representing the number of seconds since 00:00:00 GMT, January 1st, 1970.

An error occurs if *dateSecs* is not well-formed.

Syntax

```
year(dateSecs)
```

dateSecs	*number*	optional. A date represented by the number of seconds since 00:00:00 GMT, January 1st, 1970. If no value is supplied, the current date is used.

Returns

A string containing the year, or "" if an error occurs.

Example

In this example, the result of *year* is "1998".

```
<field sid="yearTestField">
   <label>Test year()</label>
   <format>
      <datatype>string</datatype>
      <constraints>
         <mandatory>on</mandatory>
      </constraints>
   </format>
   <value compute="year('890300356')"></value>
</field>
```

Details on XForms Function Calls

XFDL supports the use of XForms functions. These functions can be included in XPath expressions, and are only available when dealing with XForms-related processes.

Return Types

XForms functions can return the following data types:

- **boolean** — a true or false value.
- **string** — a group of alpha-numeric characters.
- **number** — a double precision number.
- **nodeset** — a set of nodes in the data model. The set may be empty, or it may contain or more nodes.

Empty Return Values

Some functions will return the string ″NaN″ when they are unable to compute a real return value. NaN means ″not a number″, and generally indicates an empty nodeset.

Boolean Functions

boolean-from-string

Converts a string to a boolean value. This is useful for converting string content, such as content from an instance data node, to a boolean result, which is required to set some of the properties on data elements, such as relevant and readonly.

Syntax

```
boolean-from-string(string)
```

string	*string*	the string to convert. The strings ″true″ and ″1″ are converted to true. All other strings are converted to false.

Returns

A boolean value.

Example

The following data model contains two instances. The first instance, called ″po″, contains the beginning of a table that will track the items that are being ordered. The second instance, called ″temps″, contains a temporary variable that tracks whether the form is being submitted.

```
<xforms:model functions="current">
   <xforms:instance id="po" xmlns="">
      <po>
         <order>
            <row>
               <product/>
               <unitCost>0</unitCost>
               <qty></qty>
               <lineTotal></lineTotal>
            </row>
         </order>
         <subtotal>0</subtotal>
         <tax>0</tax>
         <total>0</total>
      </po>
   </xforms:instance>
   <xforms:instance id="temps" xmlns="">
      <root>
         <submitting>false</submitting>
      </root>
</xforms:instance>
```

The following bind determines whether each row in the table is relevant:

```
<xforms:bind nodeset="order/row[not(last())]"
   relevant="boolean-from-string( if( qty > 0 or
   instance('temps')/submitting='false', 'true', 'false'))"/>
```

This bind uses an *if*, *instance*, and *boolen-from-string* function to create following logic: if the row has a quantity greater than zero, or the form is not being submitted (as tracked by the submitting element in the temps instance), then return "true"; otherwise, return "false". Since the *if* function returns these values as strings, the *boolean-from-string* function is used to convert the result to a boolean value, which then sets the relevance of the nodeset.

if

Creates a basic if/then/else decision point in an XPath expression.

Syntax

```
if(condition, then, else)
```

condition	*boolean*	an XPath expression that is evaluated as true or false.
then	*string*	a value that is returned if the condition is true.
else	*string*	a value that is returned if the condition is false.

Returns

Returns the *then* parameter if the condition is true, or the *else* parameter if the condition is false.

Example

The following data model contains two instances. The first instance, called ″po″, contains the beginning of a table that will track the items that are being ordered. The second instance, called ″temps″, contains a temporary variable that tracks whether the form is being submitted.

```
<xforms:model functions="current">
   <xforms:instance id="po" xmlns="">
      <po>
         <order>
            <row>
               <product/>
               <unitCost>0</unitCost>
               <qty></qty>
               <lineTotal></lineTotal>
            </row>
         </order>
         <subtotal>0</subtotal>
         <tax>0</tax>
         <total>0</total>
      </po>
   </xforms:instance>
   <xforms:instance id="temps" xmlns="">
      <root>
         <submitting>false</submitting>
      </root>
</xforms:instance>
```

The following bind determines whether each row in the table is relevant:

```
<xforms:bind nodeset="order/row[not(last())]"
   relevant="boolean-from-string( if( qty > 0 or
   instance('temps')/submitting='false', 'true', 'false'))"/>
```

This bind uses an *if*, *instance*, and *boolen-from-string* function to create following logic: if the row has a quantity greater than zero, or the form is not being submitted (as tracked by the submitting element in the temps instance), then return ″true″; otherwise, return ″false″. Since the *if* function returns these values as strings, the *boolean-from-string* function is used to convert the result to a boolean value, which then sets the relevance of the nodeset.

Number Functions

avg

Averages the values for a set of nodes. The strings values of the nodes are converted to numbers, added together, and then divided by the number of nodes.

Syntax

```
avg(nodeset)
```

nodeset	*XPath*	an XPath reference to a set of nodes.

Returns

A number representing the average, or NaN if any node is the nodeset is not a number or the nodeset is empty.

Example

The following model contains the beginnings of a table that tracks test scores for students:

```
<xforms:model>
   <xforms:instance xmlns="">
      <root>
         <students>
            <row>
               <name/>
               <studentNumber/>
               <score/>
            </row>
         </students>
         <lowScore/>
         <highScore/>
         <averageScore/>
      </root>
   </xforms:instance>
</xforms:model>
```

The following bind uses the *avg* function to populate the <averageScore> element:

```
<xforms:bind nodeset="averageScore"
   calculate="avg(../students/row/score)"/>
```

min

Determines the minimum value from a set of nodes. The string value of each node is converted to a number, then the minimum value is determined.

Syntax

```
min(nodeset)
```

nodeset	*XPath*	an XPath reference to a set of nodes.

Returns

A number representing the minimum value, or NaN if the nodeset was empty or one of the nodes in the set evaluated to NaN.

Example

The following model contains the beginnings of a table that tracks test scores for students:

```
<xforms:model>
   <xforms:instance xmlns="">
      <root>
         <students>
            <row>
               <name/>
               <studentNumber/>
               <score/>
            </row>
         </students>
         <lowScore/>
         <highScore/>
         <averageScore/>
      </root>
   </xforms:instance>
</xforms:model>
```

The following bind uses the *min* function to populate the <lowScore> element:

```
<xforms:bind nodeset="lowScore"
   calculate="min(../students/row/score)"/>
```

max

Determines the maximum value from a set of nodes. The string value of each node is converted to a number, then the minimum value is determined.

Syntax

```
max(nodeset)
```

nodeset	*XPath*	an XPath reference to a set of nodes.

Returns

A number representing the maximum value, or NaN if the nodeset was empty or one of the nodes in the set evaluated to NaN.

Example

The following model contains the beginnings of a table that tracks test scores for students:

```
<xforms:model>
   <xforms:instance xmlns="">
      <root>
         <students>
            <row>
               <name/>
               <studentNumber/>
               <score/>
            </row>
         </students>
         <lowScore/>
         <highScore/>
         <averageScore/>
      </root>
   </xforms:instance>
</xforms:model>
```

The following bind uses the *max* function to populate the <highScore> element:

```
<xforms:bind nodeset="highScore"
   calculate="max(../students/row/score)/>
```

count-non-empty

Determines the number of non-empty nodes in a nodeset. A node is considered to be non-empty if it can be converted into a string with a length greater than zero.

Syntax

```
count-non-empty(nodeset)
```

nodeset	*XPath*	an XPath reference to a set of nodes.

Returns

A number representing the number of non-empty nodes in the set.

Example

The following model contains the beginnings of a table that tracks test scores for students:

```
<xforms:model>
   <xforms:instance xmlns="">
      <root>
         <students>
            <row>
               <name/>
               <studentNumber/>
               <score/>
            </row>
         </students>
         <lowScore/>
         <highScore/>
         <averageScore/>
         <testsMarked/>
      </root>
   </xforms:instance>
</xforms:model>
```

The following bind uses the *count-non-empty* function to determine how many students have scores, then puts that value in the <testsMarked> element:

```
<xforms:bind nodeset="testsMarked"
   calculate="count-non-empty(../students/row/score)"/>
```

index

Determines which row of a repeat currently has the focus. Repeats are indexed with a one-based count. For example, the first row is row 1, the second is row 2, and so on.

Syntax

```
index(repeatID)
```

repeatID	*XPath*	an XPath reference to the *id* attribute of an *xforms:repeat* option.

Returns

A number indicating which row has the focus.

Example

The following button adds a row to a table of students and test scores. When the button is clicked, the *xforms:insert* action creates a new row within the table. This insert action uses the *index* function to determine which row in the table currently has the focus, and then places the new row after the row with the focus.

```
<button sid="addRow">
   <xforms:trigger>
      <xforms:label>Add Row</xforms:label>
      <xforms:action ev:event="DOMActivate">
         <xforms:insert nodeset="students/row"
            at="index('studentTable')" position="after"/>
         <xforms:setfocus control="studentTable"/>
      </xforms:action>
   </xforms:trigger>
</button>
```

String Functions

property

Queries the forms viewing engine to determine one of the following:
- **XForms version —** which version of XForms is being used.
- **XForms conformance —** which aspects of XForms are supported.

Syntax

```
property(string)
```

string	*string*	one of the following strings: • version • conformance-level

Returns

If querying the version, returns a string with the major and minor version of XForms. For example, ″1.0″.

If querying the conformance, XFDL forms viewing applications return full. Other applications may return other values, depending on their support for XForms.

Example

The following model contains a <conformance> data element that is populated by a bind. This bind uses the property function to check the conformance level of the application processing the form, and records that level in the form.

```
<xforms:model>
   <xforms:instance id="conformance" xmlns="">
      <root>
         <conformance/>
      </root>
   </xforms:instance>
   <xforms:bind nodeset="instance('conformance')/conformance"
      calculate="property('conformance-level')"/>
</xforms:model>
```

Date and Time Functions

now

Gets the current date and time from the system clock on the local computer.

Syntax

```
now()
```

Returns

A string representing the time in the following format: 2005-10-20T17:00:00Z.

Example

The following buttons submits a purchase order form. When the button is clicked, the actions are triggered in order. First, the *xforms:setvalue* action uses the *now* function to record the time of submission. Then, the *xforms:send* action submits the form to the server.

```
<button sid="submitPO">
   <xforms:trigger>
      <xforms:action ev:event="DOMActivate">
         <xforms:setvalue ref="timeStamp" value="now()"/>
         <xforms:send submission="sendPO"/>
      <xforms:label>Submit PO</xforms:label>
   </xforms:trigger>
</button>
```

days-from-date

Determines how many days difference there is between 1970-01-01 and the provided date.

Syntax

```
days-from-date(date)
```

date	*xsd:date*	a date in one of the formats specified below.

Formats

The date may be written in any of the following formats:

date The date written in the following format:

yyyy-mm-dd

Where:

- *yyyy* is a four digit year, such as 2005.
- *mm* is a two digit month, such as 02 or 10.
- *dd* is a two digit day, such as 05 or 22.

Note that the date must include the dashes. For example, June 21, 2005 would be written as:

2005-06-22

dateTime

The date and time written in the following format:

*yyyy-mm-dd*T*hh*:*mm*:*ss*Z

Where:

- *yyyy* is a four digit year, such as 2005.
- *mm* is a two digit month, such as 02 or 10.
- *dd* is a two digit day, such as 05 or 22.
- T is the time separator. You must include this.
- *hh* is a two digit hour (24 hour clock), such as 02 or 18.
- *mm* is a two digit minute, such as 03 or 55.
- *ss* is at least a two digit second, such as 08 or 43. The seconds value may also include a decimal fraction.
- Z is the timezone indicator. For no timezone adjustment, simply append Z to the string. To make a timezone adjustment, replace the Z with the following expression:

  (+ | -) *hh*:*mm*

  Where:

  – *hh* is a two digit hour (24 hour clock).

  – *mm* is a two digit minute.

Note that the dateTime must include the dashes and colons. For example, June 21, 2005 at 4:55 PM Central Daylight Savings Time would be written as:

2005-06-21T16:55:00-05:00

Returns

A number representing the number of days, or NaN if the input does not match the allowed formats.

Example

The following model might appear in an overdue notice for an online movie rental. The model contains a due date as well as a ″days late″ entry. The number of days the rental is late is calculated by a bind that performs the following processing: convert today’s date (retrieved with the *now* function) to days, then convert the due date to days, then subtract the due date from today’s date.

```
<xforms:model>
   <xforms:instance xmlns="">
      <root>
         <dueDate/>
         <daysLate/>
      </root>
   </xforms:instance>
   <xforms:bind nodeset="daysLate"
      calculate="days-from-date(now()) - days-from-date(../dueDate)"/>
</xforms:model>
```

seconds-from-dateTime

Determines how many seconds difference there is between 1970-01-01 and the provided date.

Syntax

```
seconds-from-dateTime(dateTime)
```

dateTime	*xsd:dateTime*	a date and time written in the following format:

*yyyy-mm-dd*T*hh*:*mm*:*ss*Z

Where:

- *yyyy* is a four digit year, such as 2005.
- *mm* is a two digit month, such as 02 or 10.
- *dd* is a two digit day, such as 05 or 22.
- T is the time separator. You must include this.
- *hh* is a two digit hour (24 hour clock), such as 02 or 18.
- *mm* is a two digit minute, such as 03 or 55.
- *ss* is at least a two digit second, such as 08 or 43. The seconds value may also include a decimal fraction.
- Z is the timezone indicator. For no timezone adjustment, simply append Z to the string. To make a timezone adjustment, replace the Z with the following expression:

  (+ | -) *hh*:*mm*

  Where:

  – *hh* is a two digit hour (24 hour clock).

  – *mm* is a two digit minute.

Note that the dateTime must include the dashes and colons. For example, June 21, 2005 at 4:55 PM Central Daylight Savings Time would be written as:

2005-06-21T16:55:00-05:00

Returns

A number representing the number of seconds, or NaN if the input does not match the allowed format.

Example

The following example shows an XForms model and a *button* item:

```
<xforms:model>
   <xforms:instance id="timeData" xmlns="">
      <root>
         <timeOpened/>
         <timeSubmitted/>
         <elapsedTime/>
      </root>
   </xforms:instance>
   <xforms:bind nodeset="elapsedTime"
      calculate="(seconds-from-dateTime(../timeSubmitted) -
      seconds-from-dateTime(../timeOpened)) div 60"/>
   <xforms:setvalue ev:event="xforms-ready"
      ref="instance('timeData')/timeOpened" value="now()"/>
</xforms:model>
<button sid="submit">
```

```
   <xforms:trigger>
      <xforms:action ev:event="DOMActivate">
         <xforms:setvalue ref="instance('timeData')/timeSubmitted"
            value="now()"/>
         <xforms:send submission="send"/>
      </xforms:action>
      <xforms:label>Submit</xforms:label>
   </xforms:trigger>
</button>
```

The model contains data elements that record when the form is first opened, when the form is submitted, and the elapsed time between the two. When the XForms model is first ready, an *xforms:setvalue* function is triggered in the model and records the time the form was opened. When the user clicks the submit button, an *xforms:setvalue* action is triggered in the button and records the time the form was submitted. The change in value also causes the *xforms:bind* to update, which calculates the elapsed time by converting both times into seconds, subtracting the timeOpened from the timeSubmitted, and then dividing by 60 to convert the result to minutes.

seconds

Converts a duration to an equal number of seconds. Durations may include days, hours, minutes, and seconds.

Syntax

```
seconds(duration)
```

duration	*xsd:duration*	a duration, written in the following format: P*n*Y*n*M*n*DT*n*H*n*M*n*S where: • P marks the string as a duration. • *n*Y gives the number of years. • *n*M gives the number of months. • *n*D gives the number of days. • T separates the date from the time. • *n*H gives the number of hours. • *n*M gives the number of minutes. • *n*S gives the number of seconds. For example, P1Y3M3DT12H34M21S is: 1 year, 3 months, 3 days, 12 hours, 34 minutes, and 21 seconds. If any of the values are zero, they may be omitted. For example, P120D is 120 days. Furthermore, the year and month values are ignored by this function. The T designator may only be absent if there are no time elements. The P designator must always be present. The number of seconds may include a decimal fraction.

Returns

A number (possibly fractional) representing the number of seconds, or NaN if the input does not match the allowed format.

Example

The following model converts days, hours and minutes into a total number of minutes. The model contains data elements for days, hours, and minutes. The user types these values into fields that are linked to the data elements. When data is entered, the *xforms:bind* uses the *concat* function to turn the data into a formatted duration string, converts that string to seconds using the *seconds* function, and then divides by sixty to convert the time to minutes. This value is then stored in the <totalTime> element.

```
<xforms:model>
   <xforms:instance xmlns="">
      <root>
         <days/>
         <hours/>
         <minutes/>
         <totalTime/>
      </root>
   </xforms:instance>
   <xforms:bind nodeset="totalTime"
      calculate="seconds(concat('P', ../days, 'DT', ../hours, 'H',
      ../minutes, 'M')) div 60"/>
</xforms:model>
```

months

Converts a duration to an equal number of months. Durations may include days and months.

Syntax

```
months(duration)
```

duration *xsd:duration* a duration, written in the following format:

P*n*Y*n*M*n*DT*n*H*n*M*n*S

where:
- P marks the string as a duration.
- *n*Y gives the number of years.
- *n*M gives the number of months.
- *n*D gives the number of days.
- T separates the date from the time.
- *n*H gives the number of hours.
- *n*M gives the number of minutes.
- *n*S gives the number of seconds.

For example, P1Y3M3DT12H34M21S is: 1 year, 3 months, 3 days, 12 hours, 34 minutes, and 21 seconds.

If any of the values are zero, they may be omitted. For example, P11M is 11 months. Furthermore, only the year and month values are used by this function. All other values are ignored.

Returns

A whole number representing the number of months, or NaN if the input does not match the allowed format.

Example

The following model converts years and months into a total number of months. The model contains data elements for years and months. The user types these values into fields that are linked to the data elements. When data is entered, the *xforms:bind* uses the *concat* function to turn the data into a formatted duration string, converts that string to months using the *months* function. This value is then stored in the <totalTime> element.

```
<xforms:model>
   <xforms:instance xmlns="">
      <root>
         <years/>
         <months/>
         <totalTime/>
      </root>
   </xforms:instance>
   <xforms:bind nodeset="totalTime"
      calculate="months(concat('P', ../years, 'Y', ../months, 'M'))"/>
</xforms:model>
```

Node-set Functions

instance

Locates a particular data instance within a given model.

Syntax

```
instance(instanceID)
```

instanceID	*string*	the value of the *id* attribute for the instance.

Returns

The instance nodeset.

Example

The following data model contains two instances. The first instance, called ″po″, contains the beginning of a table that will track the items that are being ordered. The second instance, called ″temps″, contains a temporary variable that tracks whether the form is being submitted.

```
<xforms:model functions="current">
   <xforms:instance id="po" xmlns="">
      <po>
         <order>
            <row>
               <product/>
               <unitCost>0</unitCost>
               <qty></qty>
               <lineTotal></lineTotal>
            </row>
         </order>
         <subtotal>0</subtotal>
         <tax>0</tax>
         <total>0</total>
      </po>
   </xforms:instance>
   <xforms:instance id="temps" xmlns="">
      <root>
         <submitting>false</submitting>
      </root>
</xforms:instance>
```

The following bind determines whether each row in the table is relevant:

```
<xforms:bind nodeset="order/row[not(last())]"
   relevant="boolean-from-string( if( qty > 0 or
   instance('temps')/submitting='false', 'true', 'false'))"/>
```

This bind uses an *if*, *instance*, and *boolen-from-string* function to create following logic: if the row has a quantity greater than zero, or the form is being submitted (as tracked by the submitting element in the temps instance), then return ″true″; otherwise, return ″false″. Since the *if* function returns these values as strings, the *boolean-from-string* function is used to convert the result to a boolean value, which then sets the relevance of the nodeset.

Utility Functions

choose

Given two nodesets, this function returns one of them (that is, it chooses between them) based on the results of an XPath expression.

Syntax

```
choose(boolean, nodeset1, nodeset2)
```

booelan	*XPath*	an XPath expression that results in a boolean value. On true, nodeset1 is used; on false, nodeset2 is used.
nodeset1	*XPath*	an XPath expression that evaluates to a nodeset.
nodeset2	*XPath*	an XPath expression that evaluates to a nodeset.

Returns

A nodeset.

Example

The following popup presents a list of either states of provinces, depending on whether the user indicates they are from the US or Canada. In this case, a full list of the states and provinces are included in the data model. The user selects US or Canada from a *radiogroup*. The popup then uses the *choose* function to select the right group of nodes from the data model: if the user selected the US (the *US* data element is **true**), the popup uses states; if not, the popup uses provinces.

```
<popup sid="stateProvince">
   <xforms:select1 ref="stateProvince" appearance="full">
      <xforms:label>Select your State/Province:</xforms:label>
      <xforms:itemset nodeset="choose(US='true',states/state,provinces/
         province)">
         <xforms:label ref="@show"></xforms:label>
         <xforms:value ref="."></xforms:value>
      </xforms:itemset>
   </xforms:select1>
</popup>
```

power

Calculates the value of x^n, where you supply x and n.

Syntax

```
power(xvalue, nvalue)
```

xvalue	*number*	the base number. This is raised to the power of n.
nvalue	*number*	the exponent. This is applied the x value.

Returns

A number representing the result, or NaN if the result is not a real number.

Example

The following model uses the power function to apply the pythagoran theorem to the sides of a triangle. The user enters sides a and b into fields that are linked to the <a> and <b> data elements. Side c is then culculated by the *xforms:bind*, which uses the *power* function to get the square root of of $a^2 + b^2$. This value is then stored in the <c> data element.

```
<xforms:model functions="power">
   <xforms:instance xmlns="">
      <root>
         <a/>
         <b/>
         <c/>
      </root>
   </xforms:instance>
   <xforms:bind nodeset="c"
      calculate="power(../a * ../a + ../b * ../b, 0.5)"/>
</xforms:model>
```

Usage Details

1. To use the *power* function, you must include the *power* function in the functions attribute of the <xforms:model> tag.

current

Returns the nodeset that is currently providing the context for the XPath expression. This is useful when you need to reset the context in the middle of an XPath expression.

Syntax

```
current()
```

Returns

Returns the context nodeset.

Example

The following model contains two data instances are used to create a form that converts currencies. The first instance contains the user-supplied information, including the amount and the currency they want to convert to. The second instance contains information about exchange rates.

```
<xforms:model functions="current">
   <xforms:instance xmlns="">
      <converter>
         <amount>100</amount>
         <currency>jpy</currency>
         <convertedAmount></convertedAmount>
      </converter>
   </xforms:instance>
   <xforms:instance xmlns="" id="convTable">
      <convTable date="20040212" currency="cdn">
```

```
         <rate currency="eur">0.59376</rate>
         <rate currency="mxn">8.37597</rate>
         <rate currency="jpy">80.23451</rate>
         <rate currency="usd">0.76138</rate>
      </convTable>
   </xforms:instance>
<xforms:model>
```

The following bind populates the <convertedAmount> element based on the data provided by the user and the exchange rate:

```
<bind nodeset="convertedAmount" calculate="../amount *
   instance('convTable')/rate[@currency=current()/../currency]"/>
```

The bind multiplies the <amount> by the proper conversion rate. The context node for this bind is established by the *nodeset* attribute as the <convertedAmount> element in the first instance. This means that all XPath is evaluated relative to this node. However, to determine the conversion rate we must refer to the second instance. To do this, we use the *instance* function to retrieve the second instance, and then check the <rate> elements in that instance.

Next, we find the correct rate by do a string comparison between the *currency* attribute of each <rate> element and the contents of the <currency> element in the first instance. However, the <currency> element is in the first instance, and we have already changed our context to the second instance by using the *instance* function. So, to refer to the <currency> element in the first instance we must use the *current* function, as shown:

```
current()/../currency
```

Using the current function resets our context to the <convertedAmount> node, which was established as our context node by the *nodeset* attribute on the bind. So this expression evaluates to:

```
convertedAmount/../currency
```

This retrieves the contents of the currency node in the first data element. This value is then compared to the *currency* attribute on each <rate> element until the correct <rate> element is located.

Usage Details

1. To use the *current* function, you must include the *current* function in the functions attribute of the <xforms:model> tag.

Quick Reference Tables

Table of Items and Form and Page Globals

The following table lists the available options for each item type:

Item	Available Options
action	activated; active; data; datagroup; delay; itemnext; itemprevious; printsettings; saveformat; transmitdatagroups; transmitformat; transmitgroups; transmititemrefs; transmititems; transmitnamespaces; transmitoptionrefs; transmitoptions; transmitpagerefs; type; url
box	bgcolor; border; fontinfo; itemlocation; itemnext; itemprevious; printbgcolor; printvisible; size; visible
button	activated; active; bgcolor; border; coordinates; data; datagroup; focused; fontcolor; fontinfo; format; help; image; imagemode; itemlocation; itemnext; itemprevious; justify; keypress; mouseover; next; previous; printbgcolor; printfontcolor; printsettings; printvisible; saveformat; signature; signatureimage; signdatagroups; signer; signformat; signgroups; signitemrefs; signitems; signnamespaces; signoptionrefs; signoptions; signpagerefs; size; transmitdatagroups; transmitformat; transmitgroups; transmititemrefs; transmititems; transmitnamespaces; transmitoptionrefs; transmitoptions; transmitpagerefs; type; url; value; visible; xforms:output; xforms:submit; xforms:trigger; xforms:upload
cell	activated; active; data; datagroup; group; itemnext; itemprevious; label; printsettings; saveformat; transmitdatagroups; transmitformat; transmitgroups; transmititemrefs; transmititems; transmitnamespaces; transmitoptionrefs; transmitoptions; transmitpagerefs; type; url; value
check	active; bgcolor; focused; fontcolor; fontinfo; help; itemlocation; itemnext; itemprevious; keypress; label; labelbgcolor; labelborder; labelfontcolor; labelfontinfo; mouseover; next; previous; printbgcolor; printfontcolor; printlabelbgcolor; printlabelfontcolor; printvisible; readonly; size; suppresslabel; value, visible; xforms:secret
checkgroup	acclabel; active; bgcolor; border; focused; format; help; itemlocation; itemnext; itemprevious; label; labelbgcolor; labelborder; labelfontcolor; labelfontinfo; mouseover; next; previous; printbgcolor; printlabelbgcolor; printlabelfontcolor; printvisible; readonly; suppresslabel; value; visible; xforms:select; xforms:select1
combobox	activated; active; bgcolor; border; focused; fontcolor; fontinfo; format; group; help; itemlocation; itemnext; itemprevious; justify; keypress; label; labelbgcolor; labelborder; labelfontcolor; labelfontinfo; mouseover; next; previous; printbgcolor; printlabelbgcolor; printfontcolor; printlabelfontcolor; printvisible; readonly; size; suppresslabel; value; visible; xforms:input; xforms:secret; xforms:select1
data	datagroup; filename; itemnext; itemprevious; mimedata; mimetype
field	active; bgcolor; border; focused; fontcolor; fontinfo; format; help; itemlocation; itemnext; itemprevious; justify; keypress; label; labelbgcolor; labelborder; labelfontcolor; labelfontinfo; mouseover; next; previous; printbgcolor; printlabelbgcolor; printfontcolor; printlabelfontcolor; printvisible; readonly; scrollhoriz; scrollvert; size; suppresslabel; value; visible; writeonly; xforms:input; xforms:secret; xforms:textarea
help	active; itemnext; itemprevious; value

Item	Available Options
label	active; bgcolor; border; fontcolor; fontinfo; format; help; image; imagemode; itemlocation; itemnext; itemprevious; justify; printbgcolor; printfontcolor; printvisible; size; suppresslabel; value; visible; xforms:output
line	fontcolor; fontinfo; itemlocation; itemnext; itemprevious; printfontcolor; printvisible; size; thickness; visible
list	active; bgcolor; border; focused; fontcolor; fontinfo; format; group; help; itemlocation; itemnext; itemprevious; keypress; label; labelbgcolor; labelborder; labelfontcolor; labelfontinfo; mouseover; next; previous; printbgcolor; printlabelbgcolor; printfontcolor; printlabelfontcolor; printvisible; readonly; size; suppresslabel; value; visible; xforms:secret; xforms:select1
pane	active; border; bgcolor; first; focused; itemlocation; itemnext; itemprevious; label; labelbgcolor; lablefontcolor; labelfontinfo; last; next; previous; printbgcolor; printlabelbgcolor; printlabelfontcolor; printvisible; suppresslabel; visible; xforms:group; xforms:switch
popup	activated; active; bgcolor; border; focused; fontcolor; fontinfo; format; group; help; itemlocation; itemnext; itemprevious; justify; keypress; label; mouseover; next; previous; printbgcolor; printfontcolor; printvisible; readonly; size; value; visible; xforms:secret; xforms:select1
radio	active; bgcolor; focused; fontcolor; fontinfo; group; help; itemlocation; itemnext; itemprevious; keypress; label; labelbgcolor; labelborder; labelfontcolor; labelfontinfo; mouseover; next; previous; printbgcolor; printlabelbgcolor; printfontcolor; printlabelfontcolor; printvisible; readonly; size; suppresslabel; value; visible
radiogroup	acclabel; active; bgcolor; border; focused; format; help; itemlocation; itemnext; itemprevious; label; labelbgcolor; labelborder; labelfontcolor; labelfontinfo; mouseover; next; previous; printbgcolor; printlabelbgcolor; printlabelfontcolor; printvisible; readonly; suppresslabel; value; visible; xforms:select1
signature	colorinfo; fullname; layoutinfo; itemnext; itemprevious; mimedata; signature; signdatagroups; signer; signformat; signitems; signitemrefs; signgroups; signnamespaces; signoptions; signoptionrefs; signpagerefs
slider	acclabel; active; bgcolor; border; focused; fontcolor; fontinfo; format; help; itemlocation; itemnext; itemprevious; label; labelbgcolor; labelborder; labelfontcolor; labelfontinfo; next; previous; printbgcolor; printfontcolor; printlabelcolor; printlabelfontcolor; printvisible; readonly; size; suppresslabel; value; visible; xforms:range
spacer	fontinfo; itemlocation; itemnext; itemprevious; label; size
table	active, bgcolor, border, first, focused, itemlocation, itemnext, itemprevious, last, next, previous, printbgcolor, printvisible, visible, xforms:repeat
toolbar	bgcolor; itemnext; itemprevious; mouseover
page globals	activated; bgcolor; border; focused; fontcolor; fontinfo; itemfirst; itemlast; keypress; label; mouseover; next; pagefirst; pageid; pagelast; pagenext; pageprevious; printbgcolor; printfontcolor; printsettings; saveformat; transmitformat
form globals	activated; bgcolor; dirtyflag; focused; fontcolor; fontinfo; formid; keypress; printbgcolor; printfontcolor; printing; printsettings; requirements; saveformat; transmitformat; triggeritem; version; webservices

Table of Options

The following table list the details for the available options:

Option	Details
acclabel	`<acclabel>`*`message`*`</acclabel>` **Default:** n/a **Items:** button; check; checkgroup; combobox; field; list; popup; radio; radiogroup; slider
activated	`<activated>on|maybe|off</activated>` **Default:** off **Items:** action; button; cell; combobox; popup; page global; form global
active	`<active>on|off</active>` **Default:** on **Items:** action; button; cell; check; checkgroup; combobox; field; help; label; list; popup; radio; radiogroup; slider **Note:** To prevent user input in a field, set the *readonly* option to **on**.
bgcolor	`<bgcolor>`*`color`*`</bgcolor>` **Default**: • for form — white • for page — the form bgcolor setting or default • for button — gray • for check, field, list, popup, radio — white • for label, table, and pane — transparent • all other items — the background color of the page **Items:** box; button; check; checkgroup; combobox; field; label; list; popup; radio; radiogroup; slider; toolbar; page globals; form globals
border	`<border>on|off</border>` **Default:** • for label — off • for all other items — on **Items:** box; button; checkgroup; combobox; field; label; list; pane; popup; radiogroup; slider; table

colorinfo	`<colorinfo>`
	`<`*`color_name`*$_1$`>`*`color`*`</`*`color_name`*$_1$`>`
	`...`
	`<`*`color_name`*$_n$`>`*`color`*`</`*`color_name`*$_n$`>`
	`</colorinfo>`
	
	Default: none
	
	Items: signature

Option	Details

coordinates	`<coordinates>`
	`   <x>X_coordinate</x>`
	`   <y>Y_coordinate</y>`
	`</coordinates>`
	
	Default: none
	
	Items: button

data	`<data>data_item</data>` **Default:** none **Items:** action; button; cell

datagroup	`<datagroup>`
	`   <datagroupref>datagroup_reference</datagroupref>`
	`   <datagroupref>datagroup reference</datagroupref>`
	`</datagroup>`
	
	Default: none
	
	Items: action; button; cell; data

delay	`<delay>`
	`   <type>repeat|once</type>`
	`   <interval>interval</interval>`
	`</delay>`
	
	Default: once with an interval of 0 seconds
	
	Items: action

dirtyflag	`<dirtyflag>on|off</dirtyflag>` **Default:** none **Items:** form globals

excludedmetadata	`<excludedmetadata>`
	`   <servernotarizations>`
	`      <notarization>Notarization</notarization>`
	`      ...`
	`      <notarization>Notarization</notarization>`
	`   </servernotarizations>`
	`</excludedmetadata>`
	
	Default: none
	
	Items: signature

filename	`<filename>file name</filename>` **Default:** none **Items:** data
first	`<first>item reference</first>` **Default:** none **Items:** table

Option	Details
focused	`<focused>on|off</focused>` **Default:** off **Items:** button; check; checkgroup; combobox; field; list; popup; radio; radiogroup; slider; page global; form global
focuseditem	`<focuseditem>`*sid*`</focuseditem>` **Default:** n/a **Items:** page global
fontcolor	`<fontcolor>`*color*`</fontcolor>` **Default:** • for check and radio — red • for all other items, the fontcolor set in the page or form globals, or black if no preference set **Items:** button; check; combobox; field; label; line; list; popup; radio; slider; page globals; form globals

fontinfo	`<fontinfo>`
	   `<fontname>`*font name*`</fontname>`
	   `<size>`*point size*`</size>`
	   `<effect>`effect_1`</ae>`
	   `...`
	   `<effect>`effect_n`</effect>`
	`</fontinfo>`
	
	* effects are optional
	
	Default: the fontinfo set in page or form globals, or Helvetica 8 plain if no characteristics set
	
	Items: box; button; check; combobox; field; label; line; list; popup; radio; slider; spacer; page globals; form globals

format	`<format>`
	   `<dataype>`*data type*`</datatype>`
	   `<presentation>`*presentation settings*`</presentation>`
	   `<constraints>`*constraint settings*`</constraints>`
	`</format>`
	
	* presentation and constraint settings are optional
	
	Default:
	• for data type — ASCII string
	• for presenation settings — depends on data type
	• for constraint settings — depends on data type
	
	Items: button; checkgroup; combobox; field; label; list; popup; radiogroup; slider

Option	Details
formid	`<formid>` `<title>`*`string`*`</title>` `<serialnumber>`*`string`*`</serialnumber>` `<version>`*`AA.Bb.cc`*`</version>` `</formid>` * format and check flags are optional, and multiple flags are valid **Default:** none **Items:** form globals
formid	`<fullname>`*`name`*`</fullname>` **Default:** none **Items:** signature
group	`<group>group name|group reference</group>` **Default:** none **Items:** cell; combobox; list; popup; radio
help	`<help>`*`item reference`*`</help>` **Default:** none **Items:** button; check; checkgroup; combobox; field; label; list; popup; radio; radiogroup; slider
image	`<image>`*`item reference`*`</image>` **Default:** none **Items:** button; label
imagemode	`<imagemode>clip|resize|scale</imagemode>` **Default:** resize **Items:** button; label
itemfirst	`<itemfirst>`*`item reference`*`</itemfirst>` **Default:** none **Items:** page global
itemlocation	`<itemlocation>` `<x>`*`x-coordinate`*`</x>` `<y>`*`y-coordinate`*`</y>` `</itemlocation>` * this is for absolute positioning. Refer to the itemlocation entry for examples of relative and offset positioning, or extent sizing. **Default:** • for the first item — the top left corner of the form • for all other items — vertically below the previously created item and horizontally at the left margin **Items:** box; button; check; checkgroup; combobox; field; label; line; list; popup; radio; radiogroup; slider; spacer

Option	Details
itemlast	`<itemlast>`*`item reference`*`</itemlast>` **Default:** none **Items:** page global
itemnext	`<itemnext>`*`item reference`*`</itemnext>` **Default:** none **Items:** action; box; button; cell; check; checkgroup; combobox; data; field; help; label; line; list; popup; radio; radiogroup; signature; slider; spacer; toolbar
itemprevious	`<itemprevious>`*`item reference`*`</itemprevious>` **Default:** none **Items:** action; box; button; cell; check; checkgroup; combobox; data; field; help; label; line; list; popup; radio; radiogroup; signature; slider; spacer; toolbar
justify	`<justify>left|right|center</justify>` **Default:** • for button and popup — center • for combobox, label, and field — left **Items:** button; combobox; field; label; popup
keypress	`<keypress>`*`key pressed`*`</keypress>` **Default:** none **Items:** button; check; combobox; field; list; popup; radio; page globals; form globals
label	`<label>`*`label text`*`</label>` **Default:** none **Items:** cell; check; checkgroup; combobox; field; list; pane; popup; radio; radiogroup; slider; spacer; page globals; form globals
labelbgcolor	`<labelbgcolor>`*`color`*`</labelbgcolor>` **Default:** transparent **Items:** check; checkgroup; combobox; field; list; pane; radio; radiogroup; slider
labelborder	`<labelborder>on|off</labelborder>` **Default:** off **Items:** check; checkgroup; combobox; field; list; radio; radiogroup; slider

Option	Details
labelfontcolor	`<labelfontcolor>`*color name*`</labelfontcolor>` **Default:** the item's labelfontcolor setting, or the global fontcolor setting or default **Items:** check; checkgroup; combobox; field; list; pane; radio; radiogroup; slider

labelfontinfo	`<labelfontinfo>`
	   `<fontname>`*font name*`</fontname>`
	   `<size>`*point size*`</size>`
	   `<effect>`$effect_1$`</effect>`
	   `...`
	   `<effect>`$effect_n$`</effect>`
	`</labelfontinfo>`
	
	* effects are optional
	
	Default: Helvetica, 8, plain
	
	Items: check; checkgroup; combobox; field; list; pane; radio; radiogroup; slider

last	`<last>`*item reference*`</last>` **Default:** none **Items:** table

layoutinfo	`<layoutinfo>`
	   `<pagehashes>`
	      `<pagehash>`
	         `<pageref>`$page\ sid_1$`</pageref>`
	         `<hash>`$pagehash_1$`</hash>`
	      `</pagehash>`
	      `...`
	      `<pagehash>`
	         `<pageref>`$page\ sid_n$`</pageref>`
	         `<hash>`$pagehash_n$`</hash>`
	      `</pagehash>`
	   `</pagehashes>`
	`</layoutinfo>`
	
	Default: none
	
	Items: signature

linespacing	`<linespacing>offset</linespacing>` **Default:** 0 **Items:** button, label, spacer
mimedata	`<mimedata encoding="`*format*`">`*data*`</mimedata>` **Default:** none **Items:** data, signature
mimetype	`<mimetype>`*MIME type*`</mimetype>` **Default:** none **Items:** data

Option	Details
mouseover	`<mouseover>on|off</mouseover>` **Default:** off **Items:** button; check; combobox; field; list; popup; radio; toolbar; page global
next	`<next>`*`item reference`*`</next>` **Default:** • when the form opens — the first non-toolbar item in the form's description that users can modify • when tabbing to subsequent items — the next item in the form's description that users can modify • when tabbing from the last item — the first item in the form's description that users can modify (can be a toolbar item) **Items:** button; combobox; check; checkgroup; field; list; popup; radio; radiogroup; slider; page globals
pagefirst	`<pagefirst>`*`page reference`*`</pagefirst>` **Default:** none **Items:** page global

pageid	`<pageid>`
	`   <serialnumber>`*`string`*`</serialnumber>`
	`</pageid>`
	
	Default: none
	
	Items: form globals

pagelast	`<pagelast>`*`page reference`*`</pagelast>` **Default:** none **Items:** page global
pagenext	`<pagenext>`*`page reference`*`</pagenext>` **Default:** none **Items:** page global
pageprevious	`<pageprevious>`*`page reference`*`</pageprevious>` **Default:** none **Items:** page global
previous	`<previous>`*`item_reference`*`</previous>` **Default:** the previous item in the form description **Items:** button; combobox; check; checkgroup; field; list; pane; popup; radio; radiogroup; slider; table

Option	Details
printbgcolor	`<printbgcolor>`*color*`</printbgcolor>` **Default:** • for form — white • for page — the form setting or default • for items — the specified or default item bgcolor **Items:** box; button; check; checkgroup; combobox; field; label; list; popup; radio; radiogroup; slider; page global; form global
printfontcolor	`<printfontcolor>`*color*`</printfontcolor>` **Default:** • for button, combobox, field, label, line, list, popup: the fontcolor setting or default for the item • for radio, check: red **Items:** button; check; combobox; field; label; line; list; popup; radio; slider
printing	`<printing>on|off</printing>` **Default:** off **Items:** form global
printlabelbgcolor	`<printlabelbgcolor>`*color*`</printlabelbgcolor>` **Default:** the item's specified or default labelbgcolor **Items:** check; checkgroup; combobox; field; list; pane; radio; radiogroup; slider
printlabelfontcolor	`<printlabelfontcolor>`*color*`</printlabelfontcolor>` **Default:** the item's specified or default labelfontcolor **Items:** check; checkgroup; combobox; field; list; pane; radio; radiogroup; slider; page globals; form global

Option	Details
printsettings	`<printsettings>` `<pages>`*`page list`*`</pages>` `<dailog>`*`dialog settings`*`</dialog>` `<border>on|off</border>` `<singlelinefieldsaslines>on|off</singlelinefieldsaslines>` `<scroll barsonfields>on|off</scroll barsonfields>` `<radioswithoutvalues>on|off</radioswithoutvalues>` `<radiosaschecks>on|off</radiosaschecks>` `<pagelayout>`*`layout setting`*`</pagelayout>` `</printsettings>` **Default:** the page list defaults to include all pages in the form, the dialog defaults to on, has the following settings: • orientation — portrait • copies — 1 • printpages active — on • printpages choices — all • border — off • singlelinefieldsaslines, scroll barsonfields, radioswithoutvalues, radiosaschecks, pagelayout — as set in form rendering software. **Items:** action; button; cell; page globals; form globals
printvisible	`<printvisible>on|off</printvisible>` **Default:** defaults to the *visible* setting for the item **Items:** box; button; check; checkgroup; combobox; field; label; line; list; popup; radio; radiogroup; slider
readonly	`<readonly>on|off</readonly>` **Default:** off **Items:** check; checkgroup; combobox; field; list; popup; radio; radiogroup; slider
requirements	`<requirements>` `<requirement>`*`requirement setting`*`s</requirement>` `<detected>off</detected>` `<requirements>` **Default:** none **Items:** form global
rtf	`<rtf>`*`rich text string`*`</rtf>` **Default:** a default rich text string is created using the *value*, *fontinfo*, *fontcolor*, *justify*, and *bgcolor* options **Items:** field
saveformat	`<saveformat>`*`mimetype`*`</saveformat>` **Default:** none **Items:** action; button; cell; form globals; page globals

Option	Details
scrollhoriz	`<scrollhoriz>never|always|wordwrap</scrollhoriz>` **Default:** never **Items:** field
scrollvert	`<scrollvert>never|always|fixed</scrollvert>` **Default:** never **Items:** field
signature	`<signature>`*string*`</signature>` **Default:** none **Items:** button; signature
signatureimage	`<signatureimage>`*data item reference*`</signatureimage>` **Default:** none **Items:** button
signdatagroups	`<signdatagroups>` `<filter>keep|omit</filter>` `<datagroupref>`*datagroup reference*`</datagroupref>` `<datagroupref>`*datagroup reference*`</datagroupref>` `</signdatagroups>` **Default:** keep **Items:** button; signature
signdetails	`<signdetails>` `<dialogcolumns>` `<property>`*attribute*$_1$`</property>` `<property>`*attribute*$_n$`</property>` `</dialogcolumns>` `<filteridentity>` `<filter>` `<tag>`*attribute*$_1$`</tag>` `<value>`*value*$_1$`</value>` `</filter>` `<filter>` `<tag>`*attribute*$_n$`</tag>` `<value>`*value*$_n$`</value>` `</filter>` `</filteridentity>` `</signdetails>` **Default:** all certificates are available, and the owner's common name and e-mail address is shown. **Items:** button; signature
signer	`<signer>string</signer>` **Default:** none **Items:** button; signature

Option	Details
signformat	`<signformat>` `MIME type; engine="signature engine"; verifier;` `cval; parameters` `</signformat>` **Default:** XFDL MIME type; Generic RSA signature engine; Basic verifier; do not sign current values **Items:** button; signature
signgroups	`<signgroups>` `<filter>keep|omit</filter>` `<groupref>`*group reference$_1$*`</groupref>` `...` `<groupref>`*group reference$_n$*`</groupref>` `</signgroups>` **Default:** keep **Items:** button; signature
signinstance	`<signinstance>` `<filter>`*instance filter*`</filter>` `<dataref`$_1$`>` `<model>`*model ID*`</model>` `<ref>`*XPath*`</ref>` `</dataref>` `...` `<dataref`$_n$`>` `...` `</dataref>` `</signinstance>` **Default:** keep **Items:** button; signature
signitems	`<signitems>` `<filter>keep|omit</filter>` `<itemtype>`*item type$_1$*`</itemtype>` `...` `<itemtype>`*item type$_n$*`</itemtype>` `</signitems>` **Default:** keep **Items:** button; signature
signitemrefs	`<signitemrefs>` `<filter>keep|omit</filter>` `<itemref>`*item reference$_1$*`</itemref>` `...` `<itemref>`*item reference$_n$*`</itemref>` `</signitemrefs>` **Default:** keep **Items:** button; signature

Option	Details

signnamespaces	`<signnamespaces>`
	   `<filter>keep|omit</filter>`
	   `<uri>`*namespace URI*$_1$`</uri>`
	   `...`
	   `<uri>`*namespace URI*$_n$`</uri>`
	`</signnamespaces>`
	
	Default: keep
	
	Items: button; signature

signoptionrefs	`<signoptionrefs>`
	   `<filter>keep|omit</filter>`
	   `<optionref>`*option reference*$_1$`</optionref>`
	   `...`
	   `<optionref>`*option reference*$_n$`</optionref>`
	`</signoptionrefs>`
	
	Default: keep
	
	Items: button; signature

signoptions	`<signoptions>`
	   `<filter>keep|omit</filter>`
	   `<optiontype>`*option type*$_1$`</optiontype>`
	   `...`
	   `<optiontype>`*option type*$_n$`</optiontype>`
	`</signoptions>`
	
	Default: keep
	
	Items: button; signature

signpagerefs	`<signpagerefs>`
	   `<filter>keep|omit</filter>`
	   `<pageref>`*page reference*$_1$`</pageref>`
	   `...`
	   `<pageref>`*page reference*$_n$`</pageref>`
	`</signpagerefs>`
	
	Default: keep
	
	Items: button; signature

size	`<size>`
	   `<width>`*width*`</width>`
	   `<height>`*height*`</height>`
	`</size>`
	
	* the unit of measurement is characters.
	
	Default: see “Default Sizes” on page 393
	
	Items: box; button; check; combobox; field; label; line; list; popup; radio; slider; spacer

suppresslabel	`<suppresslabel>on|off</suppresslabel>` **Default:** off **Items:** check; checkgroup; combobox; field; label; list; pane; radio; radiogroup; slider

Option	Details
texttype	`<texttype>text/plain|text/rtf<texttype>` **Default:** text/plain **Items:** field
thickness	`<thickness>`*thickness*`</thickness>` **Default:** 1 pixel **Items:** line

transmitdatagroups	`<transmitdatagroups>`
	   `<filter>keep|omit</filter>`
	   `<datagroupref>`*datagroup identifier*$_1$`</datagroupref>`
	   `...`
	   `<datagroupref>`*datagroup identifier*$_n$`</datagroupref>`
	`</transmitdatagroups>`
	
	Default: keep
	
	Items: action; button; cell

transmitformat	`<transmitformat>`*MIME type*`</transmitformat>` **Default:** application/vnd.xfdl **Items:** action; button; cell; form globals; page globals

transmitgroups	`<transmitgroups>`
	   `<filter>keep|omit</filter>`
	   `<groupref>`*group identifier*$_1$`</groupref>`
	   `...`
	   `<groupref>`*group identifier*$_n$`</groupref>`
	`</transmitgroups>`
	
	Default: keep
	
	Items: action; button; cell

transmititemrefs	`<transmititemrefs>`
	   `<filter>keep|omit</filter>`
	   `<itemref>`*item identifier*$_1$`</itemref>`
	   `...`
	   `<itemref>`*item identifier*$_n$`</itemref>`
	`</transmititemrefs>`
	
	Default: keep
	
	Items: action; button; cell

transmititems	`<transmititems>`
	   `<filter>keep|omit</filter>`
	   `<itemtype>`*item type*$_1$`</itemtype>`
	   `...`
	   `<itemtype>`*item type*$_n$`</itemtype>`
	`</transmititems>`
	
	Default: keep
	
	Items: action; button; cell

Option	Details

transmitnamespaces	`<transmitnamespaces>`
	   `<filter>keep|omit</filter>`
	   `<uri>`*namespace URI*$_1$`</uri>`
	   `...`
	   `<uri>`*namespace URI*$_n$`</uri>`
	`</transmitnamespaces>`
	
	Default: keep
	
	Items: action; button; cell

transmitoptionrefs	`<transmitoptionrefs>`
	   `<filter>keep|omit</filter>`
	   `<optionref>`*option identifier*$_1$`</optionref>`
	   `...`
	   `<optionref>`*option identifier*$_n$`</optionref>`
	`</transmitoptionrefs>`
	
	Default: keep
	
	Items: action; button; cell

transmitoptions	`<transmitoptions>`
	    `<filter>keep|omit</filter>`
	   `<optiontype>`*option type*$_1$`</optiontype>`
	   `...`
	   `<optiontype>`*option type*$_n$`</optiontype>`
	`</transmitoptions>`
	
	Default: keep
	
	Items: action; button; cell

transmitpagerefs	`<transmitpagerefs>`
	   `<filter>keep|omit</filter>`
	   `<pageref>`*page identifier*$_1$`</pageref>`
	   `...`
	   `<pageref>`*page identifier*$_n$`</pageref>`
	`</transmitpagerefs>`
	
	Default: keep
	
	Items: action; button; cell

triggeritem	`<triggeritem>`*item reference*`</triggeritem>` **Default:** the item reference of the item that triggered the submit or done **Items:** form globals
type	`<type>`*action type*`</type>` **Default:** select **Items:** action; button; cell
url	`<url>`*URL|item reference*`</url>` **Default:** none **Items:** action; button; cell

Option	**Details**
value	`<value>`*`setting`*`</value>` **Default:** depends on item **Items:** button; cell; check; checkgroup; combobox; field; help; label; list; popup; radio; radiogroup; slider
visible	`<visible>on|off</visible>` **Default:** on **Items:** box; button; check; checkgroup; combobox; field; label; line; list; popup; radio; radiogroup; slider

webservices	`<webservices>`
	   `<wsdl:name>`*`name of webservice`*`</wsdl>`
	`</webservices>`
	
	Default: none
	
	Items: form global

writeonly	`<writeonly>on|off</writeonly>` **Default:** off **Items:** field

xformsmodels	`<xformsmodels>`
	   `<xforms:model id="`*`name`*$_1$`">`
	      `<xforms:instance id="`*`name`*$_1$`">`
	         `instance`$_1$
	      `</xforms:instance>`
	      `...`
	      `<xforms:instance id="`*`name`*$_n$`">`
	         `instance`$_n$
	      `</xforms:instance>`
	   `</xforms:model>`
	   `...`
	   `<xforms:model id="`*`name`*$_n$`">`
	      `...`
	   `</xforms:model>`
	   `<xforms:bind `*`property setting`*$_1$`>`
	   `</xforms:bind>`
	   `...`
	   `<xforms:bind `*`property setting`*$_n$`>`
	   `</xforms:bind>`
	`</xformsmodels>`
	
	Default: none
	
	Items: form global

xforms:group	`<xforms:group model="`*`model ID`*`" ref="`*`XPath`*`">`
	   `...items in group...`
	`</xforms:group>`
	
	Default: none
	
	Items: pane

Option	Details

xforms:input	`<xforms:input model="model ID" ref="XPath">`
	`   <xforms:label>label text</xforms:label>`
	`   Alert Setting`
	`   Hint Setting`
	`   Help Setting`
	`</xforms:input>`
	
	Default: none
	
	Items: combobox; check; field

xforms:output	`<xforms:output model="model ID" ref="XPath"`
	`   mediatype="MIME type">`
	`   Alert Setting`
	`   Hint Setting`
	`   Help Setting`
	`</xforms:output>`
	
	Default: none
	
	Items: label

xforms:range	`<xforms:range single node binding start="start"`
	`   end="end" step="step">`
	`   <xforms:label>label text</xforms:label>`
	`</xforms:range>`
	
	Default: none
	
	Items: slider

xforms:repeat	`<xforms:repeat id="name" model="model ID"`
	`   nodeset="XPath" startindex="index">`
	`      ...XFDL items...`
	`</xforms:repeat>`
	
	Default: none
	
	Items: field

xforms:secret	`<xforms:secret model="model ID" ref="XPath">`
	`   <xforms:label>label text</xforms:label>`
	`   Alert Setting`
	`   Hint Setting`
	`   Help Setting`
	`</xforms:secret>`
	
	Default: none
	
	Items: field

Option	Details

xforms:select	`<xforms:select model="`*model ID*`"`
	   `ref="`*XPath to element*`" appearance="full">`
	   `<xforms:label>`*label text*`</xforms:label>`
	   `<xforms:item`$_1$`>`
	      `<xforms:label>`*label for choice*`</xforms:label>`
	      `<xforms:value>`*value for choice*`</xforms:value>`
	   `</xforms:item`$_1$`>`
	   `...`
	   `<xforms:item`$_n$`>`
	      `...`
	   `</xforms:item`$_n$`>`
	   Alert Setting
	   Hint Setting
	   Help Setting
	`</xforms:select>`
	`OR`
	`<xforms:select model="`*model ID*`"`
	   `ref="`*XPath to element*`" appearance="full">`
	   `<xforms:label>`*label text*`</xforms:label>`
	   `<xforms:itemset nodeset="`*XPath to choices*`">`
	      `<xforms:label ref="`*XPath to label text*`"/>`
	      `<xforms:value ref="."></xforms:value>`
	   `</xforms:itemset>`
	   Alert Setting
	   Hint Setting
	   Help Setting
	`</xforms:select>`
	
	Default: none
	
	Items: checkgroup

Option	Details

xforms:select1	`<xforms:select1 model="model ID"`
	`ref="XPath to element" appearance="type"`
	`selection="input">`
	`<xforms:label>label text</xforms:label>`
	`<xforms:item`$_1$`>`
	`<xforms:label>label for choice</xforms:label>`
	`<xforms:value>value for choice</xforms:value>`
	`</xforms:item`$_1$`>`
	`...`
	`<xforms:item`$_n$`>`
	`...`
	`<xforms:item`$_n$`>`
	Alert Setting
	Hint Setting
	Help Setting
	`</xforms:select>`
	OR
	`<xforms:select1 model="model ID"`
	`ref="XPath to element" appearance="type"`
	`setting="input">`
	`<xforms:label>label text</xforms:label>`
	`<xforms:itemset nodeset="XPath to choices">`
	`<xforms:label ref="XPath to label text"/>`
	`<xforms:value ref="."></xforms:value>`
	`</xforms:itemset>`
	Alert Setting
	Hint Setting
	Help Setting
	`</xforms:select>`
	
	Default: none
	
	Items: checkgroup, combobox, list, popup, radiogroup

xforms:submit	`<xforms:submit submission="ID">`
	`<xforms:label>label text</xforms:label>`
	Alert Setting
	Hint Setting
	Help Setting
	`</xforms:submit>`
	
	Default: none
	
	Items: action; button

xforms:switch	`<switch id="name" ref="XPath">`
	`<xforms:case`$_1$` id="name" selection="boolean"`
	`...XFDL items...`
	`</xforms:case`$_1$`>`
	`...`
	`<xforms:case`$_n$` id="name" selection="boolean"`
	`...XFDL items...`
	`</xforms:case`$_n$`>`
	`</switch>`
	
	Default: none
	
	Items: pane

Option	Details

xforms:textarea	
	```
	<xforms:textarea model="*model ID*" ref="*XPath*">
	   <xforms:label>*label text*</xforms:label>
	   *Alert Setting*
	   *Hint Setting*
	   *Help Setting*
	</xforms:textarea>
	```
	 Default: none
	
	Items: field

xforms:trigger	
	```
	<xforms:trigger model="*model ID*" ref="*XPath*">
	   <xforms:label>*label text*</xforms:label>
	   *XForms Action*
	   *Alert Setting*
	   *Hint Setting*
	   *Help Setting*
	</xforms:trigger>
	```
	 Default: none
	
	Items: action; button

xforms:upload	
	```
	<xforms:upload model="*model ID*" ref="*upload XPath*"
	   mediatype="*MIME type*">
	   <xforms:label>*label text*</xforms:label>
	   <xforms:mediatype ref="*mediatype XPath*"/>
	   <xforms:filename ref="*filename XPath*"/>
	   *Alert Setting*
	   *Hint Setting*
	   *Help Setting*
	</xforms:upload>
	```
	 Default: none
	
	Items: button

Cross Reference Table for Items and Options

	action	box	button	cell	check	checkgroup	combobox	data	field	help	label	line	list	pane	popup	radio	radiogroup	signature	slider	spacer	table	toolbar	page global	form global (global page)
acclabel			•		•	•	•		•				•		•	•	•		•					
activated	•		•	•			•								•								•	•
active	•		•	•	•	•	•		•	•	•		•	•	•	•	•		•		•			
bgcolor		•	•		•	•	•		•		•		•	•	•	•	•		•		•	•	•	•
border		•	•		•	•	•		•		•		•	•	•	•	•		•		•			
colorinfo																		•						
coordinates			•																					
data	•		•	•																				

	action	box	button	cell	check	checkgroup	combobox	data	field	help	label	line	list	pane	popup	radio	radiogroup	signature	slider	spacer	table	toolbar	page global	form global (global page)
datagroup	•		•	•				•																
delay	•																							
dirtyflag																								•
excluded metadata																		•						
filename								•																
first														•							•			
focused			•		•	•	•		•				•	•	•	•	•		•		•		•	•
focuseditem																							•	
fontcolor			•		•		•		•		•	•	•		•	•			•				•	•
fontinfo		•	•		•		•		•		•	•	•		•	•			•	•			•	•
format			•			•	•		•		•		•		•		•		•					
formid																								•
fullname																		•						
group				•			•						•		•	•								
help			•		•	•	•		•		•		•		•	•	•		•					
image			•								•													
imagemode			•								•													
itemfirst																							•	
itemlast																							•	
itemlocation		•	•		•	•	•		•		•	•	•	•	•	•	•		•	•	•			
itemnext	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•		
itemprevious	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•		
justify			•				•		•		•				•									
keypress			•		•		•		•				•		•	•							•	•
label				•	•	•	•		•				•	•	•	•	•		•	•			•	•
labelbgcolor					•	•	•		•				•	•		•	•		•					
labelborder					•	•	•		•				•			•	•		•					
labelfontcolor					•	•	•		•				•	•		•	•		•					
labelfontinfo					•	•	•		•				•	•		•	•		•					
last														•							•			
layoutinfo																		•						
linespacing			•								•									•				
mimedata								•										•						
mimetype								•																
mouseover			•		•	•^{2}	•		•				•		•	•	•^{2}					•	•	

	action	box	button	cell	check	checkgroup	combobox	data	field	help	label	line	list	pane	popup	radio	radiogroup	signature	slider	spacer	table	toolbar	page global	form global (global page)
next			•		•	•	•		•				•	•	•	•	•		•		•		•	
pagefirst																							•	
pageid																							•	
pagelast																							•	
pagenext																							•	
pageprevious																							•	
previous			•		•	•	•		•				•	•	•	•	•		•		•		•	
printbgcolor		•	•		•	•	•		•		•		•	•	•	•	•		•		•		•	•
printfontcolor			•		•		•		•		•	•	•		•	•			•				•	•
printing																								•
printlabel bgcolor					•	•	•		•				•	•		•	•		•					
printlabel fontcolor					•	•	•		•				•	•		•	•		•					
printsettings	•		•	•																			•	•
printvisible		•	•		•	•	•		•		•	•	•	•	•	•	•		•		•			
readonly					•	•	•		•				•		•	•	•		•					
requirements																								•
rtf									•															
saveformat	•		•	•																			•	•
scrollhoriz									•															
scrollvert									•															
signature			•															•						
signature image			•																					
sign datagroups			•															•						
signdetails			•															•						
signer			•															•						
signformat			•															•						
signgroups			•															•						
signinstance			•															•						
signitemrefs			•															•						
signitems			•															•						
signname spaces			•															•						
signoptionrefs			•															•						

	action	box	button	cell	check	checkgroup	combobox	data	field	help	label	line	list	pane	popup	radio	radiogroup	signature	slider	spacer	table	toolbar	page global	form global (global page)
signoptions			•															•						
signpagerefs			•															•						
size		•	•		•		•		•		•	•	•		•	•			•	•				
suppresslabel					•	•	•		•		•		•	•		•	•		•					
texttype									•															
thickness												•												
transmit datagroups	•		•	•																				
transmit format	•		•	•																			•	•
transmit groups	•		•	•																				
transmit itemrefs	•		•	•																				
transmititems	•		•	•																				
transmit namespaces	•		•	•																				
transmit optionrefs	•		•	•																				
transmit options	•		•	•																				
transmit pagerefs	•		•	•																				
triggeritem																								•
type	•		•	•																				
url	•		•	•																				
value			•	•	•	•	•		•	•	•		•		•	•	•		•					
visible		•	•		•	•	•		•		•	•	•	•	•	•	•		•		•			
webservices																								•
writeonly									•															
xformsmodels																								•
xforms:group														•										
xforms:input						•	•		•															
xforms:output			•[1]								•													
xforms:range																			•					
xforms:repeat																					•			
xforms:secret									•															
xforms:select						•							•											

	action	box	button	cell	check	checkgroup	combobox	data	field	help	label	line	list	pane	popup	radio	radiogroup	signature	slider	spacer	table	toolbar	page global	form global (global page)
xforms: select1						•	•						•		•		•							
xforms:submit	•		•																					
xforms:switch														•										
xforms: textarea									•															
xforms:trigger	•		•																					
xforms:upload			•																					

[1]*xforms:output* cannot be the immediate child of a button; however, it can be a descendent through the *xforms:trigger*, *xforms:submit*, or *xforms:upload* options.

[2]For *checkgroup* and *radiogroup*, the *mouseover* option is active for each item in the group, not for the group as a whole. This means that you can use the <xforms:extension> element to make changes to individual items in the group. For more information, see “checkgroup” on page 58 and “radiogroup” on page 63.

Default Sizes

The following table shows the default basic item and bounding box sizes:

Item	Default Item Size	Bounding Box Size
box	**width:** 1 character **height:** 1 character Smaller than 1 not allowed in either dimension	Same as default item size
button	**width:** width of text **height:** height of text (text is the *value* option) **or** size of embedded image if it exists	Same as default item size
check	**width:** 1 character **height:** 1 character	**width:** larger of 1 character or label width **height:** label height plus 1 character
checkgroup	**width:** width checkgroup contents plus 6 pixels (for default locations, this is the width of the widest check label, plus the width of the check box, plus 6 pixels) **height:** the height of the checkgroup contents plus 6 pixels (for default locations, this is 1 character for each check in the group, plus 5 pixels between each check, plus 6 pixels)	Same as default item size
combobox	**width:** larger of label width and widest cell [3] **height:** 1 character	Same as default item size [2]
field	**width:** 30 characters **height:** 1 character	**width:** larger of item width and label width [2] **height:** height of item plus height of label [2]
label	**width:** 1 character if label empty, otherwise label width **height:** 1 character if label empty, otherwise label height **or** size of embedded image if it exists	Same as default item size
line	**width:** 30 character **height:** 1 pixel One dimension must be 0 [1]	Same as default item size

Item	Default Item Size	Bounding Box Size
list	**width:** larger of label width and widest cell [3] **height:** number of cells in list	**width:** larger of item width and widest cell [2] **height:** height of item plus height of label
pane	**width:** the width of the pane contents, plus 6 pixels **height:** the height of the pane contents, plus 6 pixels	Same as default item size
popup	**width:** larger of label width and widest cell [3] **height:** 1 character	Same as default item size [2]
radio	**width:** 1 character **height:** 1 character	**width:** larger of 1 character or label width **height:** label height plus 1 character
radiogroup	**width:** width radiogroup contents plus 6 pixels (for default locations, this is the width of the widest radio label, plus the width of the radio button, plus 6 pixels) **height:** the height of the radiogroup contents plus 6 pixels (for default locations, this is 1 character for each radio in the group, plus 5 pixels between each radio, plus 6 pixels)	Same as default item size
slider	**width:** 20 characters **height:** 3 characters	
spacer	**width:** 1 character if label empty, otherwise label width **height:** 1 character if label empty, otherwise label height (label is invisible)	Same as default item size
table	**width:** the width of the table contents plus 6 pixels **height:** the height of the table contents, plus 1 pixel for each relevant row, plus 6 pixels	Same as default item size

Usage Details

1. For *line* items, either height or width must be set to zero. The *thickness* option specifies the thickness (in pixels) of the line in the dimension containing zero (0).
2. This includes a scroll bar if one appears.
3. The cell's width comes from the cell's *value* option setting.

Order of Precedence of Filters

signature and transmission filters are applied with an order of precedence. This prevents potential filter conflicts, in which one filter might stipulate that you omit and item while another filter might stipulate that you keep an item.

When using signatures, note that the *mimedata* option is always omitted in the following scenarios, regardless of the signature filters in use:
- The *mimedata* option in a *signature* item is always omitted from the signature that item represents.
- The *mimedata* option in a *data* item that stores a signature image (see the *signatureimage* option) is always omitted from the signature that image represents.

Filters are applied in the following order:

Filter	Behavior If keep flag is used	Behavior If omit flag is used	Usage Details
1. Filter namespaces, based on *transmitnamespaces* or *signnamespaces* setting.	Keeps only elements and attributes in the namespaces indicated; throws others out.	Omits only elements and attributes in the namespaces indicated; throws them out.	An element is kept if any of its children are kept, even if it is in the wrong namespace.
2. Filter types of items, based on *transmititems* or *signitems* setting.	Keeps only those types indicated; throws others out, including their options.	Omits only those types indicated; throws them out, including their options.	
3. Filter types of options based on *transmitoptions* and *signoptions* setting.	In the items that remain, keeps all option types indicated; throws others out.	In the items that remain, omits all option types indicated.	
4. Filter specific pages based on *transmitpagerefs* or *signpagerefs* settings.	Keeps the pages whose tags are specified. Settings in *transmitnamespaces*, *transmititems*, and *transmitoptions* are respected.	Omits the pages whose tags are specified. Overrides settings in *transmitnamespaces*, *transmititems*, and *transmitoptions.*	The page does not entirely disappear from the source code; the page tags still exist.
5. Filter groups of items based on *transmitdatagroups* and *transmitgroups*, or *signdatagroups* and *signgroups* settings.	Keeps those items whose tags are specified, even if the items are of a type that should not be kept according to a *transmitnamespaces* or *transmititems* setting.	Omits those items whose tags are specified, even if the items are of a type that should be kept according to a *transmitnamespaces* or *transmititems* setting.	This option's settings override those in *transmitpagerefs* and *signpagerefs*.

Filter	Behavior If keep flag is used	Behavior If omit flag is used	Usage Details
6. Filter specific items based on *transmititemrefs* or *signitemrefs* settings.	Keeps the items whose tags are specified; overrides previous settings if necessary. Settings in *transmitoptions* and *signoptions* are respected.	Omits the items specified; overrides the previous settings if necessary.	This option's settings override those in *transmitnamespaces*, *transmititems*, *transmitgroups*, *transmitpagerefs* and *transmitdatagroups* or *signitems*, *signgroups*, *signpagerefs* and *signdatagroups*.
7. Filter specific options based on *transmitoptionrefs* and *signoptionrefs* settings.	Regardless of all other settings above, keeps the specific option instances referred to; does not keep any other options; in the case of items that will be omitted except for a single option, the item will be kept, with its original sid, and only the one option.	Omits the options specified; overrides the previous settings if necessary.	This option's settings override all other filters, including *transmitnamespaces*, *transmititems*, *transmitdatagroups*, *transmitgroups*, *transmititemrefs*, *transmitpagerefs*, *transmitoptions* or *signitems*, *signdatagroups*, *signgroups*, *signitemrefs*, *signpagerefs*, and *signoptions*.
8. Filter based on *signinstance* settings.	Regardless of all other settings, keeps the data elements indicated.	Regardless of all other settings, omits the data elements indicated.	This option's settings override all other filters.

Example

This example uses the transmit-family of options. The order of precedence would be the same for the sign-family of options.

```
<page sid="Page1">
   <global sid="global"></global>
   <button sid="submitButton">
      <value>Filter Submission</value>
      <type>done</type>
      <url>http://www.server.dmn/cgi-bin/processForm</url>
      <transmitnamespaces>
         <filter>omit</filter>
         <uri>http://www.ibm.com/xmlns/prod/XFDL/Custom</uri>
      </transmitnamespaces>
      <transmititems>
         <filter>omit</filter>
         <itemtype>data</itemtype>
      </transmititems>
      <transmitdatagroups>
         <filter>keep</filter>
         <datagroupref>enclosures</datagroupref>
         <datagroupref>related</datagroupref>
      </transmitdatagroups>
      <transmititemrefs>
         <filter>omit</filter>
         <itemref>Page1.data2</itemref>
      </transmititemrefs>
```

```
         <transmitoptions>
            <filter>omit</filter>
            <optiontype>filename</optiontype>
         </transmitoptions>
      </button>
      <button sid="encloseButton">
         <image>encloseImageData</image>
         <type>enclose</type>
         <datagroup>
            <datagroupref>enclosures</datagroupref>
            <datagroupref>related</datagroupref>
         </datagroup>
      </button>
      <data sid="data1">
         <custom:id>324</custom:id>
         <datagroup>
            <datagroupref>enclosures</datagroupref>
         </datagroup>
         <filename>jobdescr.frm</filename>
         <mimedata encoding="base64-gzip">dfksdfsdfhsdhs</mimedata>
      </data>
      <data sid="data2">
         <datagroup>
            <datagroupref>related</datagroupref>
         </datagroup>
         <filename>resume.doc</filename>
         <mimedata encoding="base64-gzip">dfhsjdfsjhfjs</mimedata>
      </data>
      <data sid="encloseImageData">
         <filename>c:\images\enclose.jpg</filename>
         <mimedata encoding="base64-gzip">
            aswWWW8MjfbyhsUELKKEFir8dfdUUUmnskshie3mkjkkeiIIUIUOl
            fRlgdsoepgejgjj1sd/3/6nnII/fjkess9Wfgjgkggkllgakkk2kl
         </mimedata>
      </data>
   </page>
```

As a result of the filtering, the following would happen (see result form description below):

- The custom ″id″ option would be stripped from the ″data1″ item, as a result of the *transmitnamespaces* setting.
- The ″encloseImageData″ data item would be stripped from the form, as a result of the *transmititems* setting.
- The ″data1″ data item would remain in the form, as a result of the *transmitdatagroups* setting, but would not contain the custom ″id″ option.
- The ″data2″ data item would be stripped from the form, as a result of the *transmititemrefs* setting.
- The *filename* option would be stripped from ″data1″, as a result of the *transmitoptions* setting.

The form description that would be received once filtering was applied would look like this:

```
   <page sid="page1">
      <global sid="global"></global>
      <button sid="submitButton">
         <value>Filter Submission</value>
         <type>done</type>
         <url>http://www.server.dmn/cgi-bin/processForm</url>
         <transmitnamespaces>
            <filter>omit</filter>
            <uri>http://www.ibm.com/xmlns/prod/XFDL/Custom</uri>
         </transmitnamespaces>
         <transmititems>
            <filter>omit</filter>
            <itemtype>data</itemtype>
         </transmititems>
         <transmitdatagroups>
            <filter>keep</filter>
            <datagroupref>enclosures</datagroupref>
            <datagroupref>related</datagroupref>
         </transmitdatagroups>
         <transmititemrefs>
            <filter>omit</filter>
            <itemref>page1.data2</itemref>
         </transmititemrefs>
         <transmitoptions>
            <filter>omit</filter>
            <optiontype>filename</optiontype>
         </transmitoptions>
      </button>
      <button sid="encloseButton">
         <image>encloseImageData</image>
         <type>enclose</type>
         <datagroup>
            <datagroupref>enclosures</datagroupref>
            <datagroupref>related</datagroupref>
         </datagroup>
      </button>
      <data sid="data1">
         <datagroup>
            <datagroupref>enclosures</datagroupref>
         </datagroup>
         <mimedata encoding="base64-gzip">dfksdfsdfhsdhs</mimedata>
      </data>
   </page>
```

Color Table

You can specify a color using either the color's name, its RGB triplet, or the hex value for the color. Each value in the RGB triplet is a number from 0 to 255 inclusive, representing the amount of primary color (red, green or blue) required to produce the secondary color. Zero represents the least amount of a color and 255 represents the greatest amount of a color.

For example, the statement:

```
<bgcolor>255,255,255</bgcolor>
```

is equivalent to:

```
<bgcolor>white</bgcolor>
```

The following pages list the names and RGB triplet values for the available colors.

Note: The transparent color has no RGB equivalent.

RGB	Color Name
240 248 255	alice blue
240 248 255	aliceblue
250 235 215	antique white
250 235 215	antiquewhite
255 239 219	antiquewhite1
238 223 204	antiquewhite2
205 192 176	antiquewhite3
139 131 120	antiquewhite4
127 255 212	aquamarine
127 255 212	aquamarine1
118 238 198	aquamarine2
102 205 170	aquamarine3
69 139 116	aquamarine4
240 255 255	azure
240 255 255	azure1
224 238 238	azure2
193 205 205	azure3
131 139 139	azure4
245 245 220	beige
255 228 196	bisque
255 228 196	bisque1
238 213 183	bisque2
205 183 158	bisque3
139 125 107	bisque4
0 0 0	black

RGB	**Color Name**
255 235 205	blanched almond
255 235 205	blanchedalmond
0 0 255	blue
138 43 226	blue violet
0 0 255	blue1
0 0 238	blue2
0 0 205	blue3
0 0 139	blue4
138 43 226	blueviolet
165 42 42	brown
255 64 64	brown1
238 59 59	brown2
205 51 51	brown3
139 35 35	brown4
222 184 135	burlywood
255 211 155	burlywood1
238 197 145	burlywood2
205 170 125	burlywood3
139 115 85	burlywood4
95 158 160	cadet blue
95 158 160	cadetblue
152 245 255	cadetblue1
142 229 238	cadetblue2
122 197 205	cadetblue3
83 134 139	cadetblue4
127 255 0	chartreuse
127 255 0	chartreuse1
118 238 0	chartreuse2
102 205 0	chartreuse3
69 139 0	chartreuse4
210 105 30	chocolate
255 127 36	chocolate1
238 118 33	chocolate2
205 102 29	chocolate3
139 69 19	chocolate4
255 127 80	coral
255 114 86	coral1
238 106 80	coral2
205 91 69	coral3
139 62 47	coral4
100 149 237	cornflower blue

RGB	Color Name
100 149 237	cornflowerblue
255 248 220	cornsilk
255 248 220	cornsilk1
238 232 205	cornsilk2
205 200 177	cornsilk3
139 136 120	cornsilk4
0 255 255	cyan
0 255 255	cyan1
0 238 238	cyan2
0 205 205	cyan3
0 139 139	cyan4
184 134 11	dark goldenrod
0 100 0	dark green
189 183 107	dark khaki
85 107 47	dark olive green
255 140 0	dark orange
153 50 204	dark orchid
233 150 122	dark salmon
143 188 143	dark sea green
72 61 139	dark slate blue
47 79 79	dark slate gray
47 79 79	dark slate grey
0 206 209	dark turquoise
148 0 211	dark violet
184 134 11	darkgoldenrod
255 185 15	darkgoldenrod1
238 173 14	darkgoldenrod2
205 149 12	darkgoldenrod3
139 101 8	darkgoldenrod4
0 100 0	darkgreen
189 183 107	darkkhaki
85 107 47	darkolivegreen
202 255 112	darkolivegreen1
188 238 104	darkolivegreen2
162 205 90	darkolivegreen3
110 139 61	darkolivegreen4
255 140 0	darkorange
255 127 0	darkorange1
238 118 0	darkorange2
205 102 0	darkorange3
139 69 0	darkorange4

RGB	Color Name
153 50 204	darkorchid
191 62 255	darkorchid1
178 58 238	darkorchid2
154 50 205	darkorchid3
104 34 139	darkorchid4
233 150 122	darksalmon
143 188 143	darkseagreen
193 255 193	darkseagreen1
180 238 180	darkseagreen2
155 205 155	darkseagreen3
105 139 105	darkseagreen4
72 61 139	darkslateblue
47 79 79	darkslategray
151 255 255	darkslategray1
141 238 238	darkslategray2
121 205 205	darkslategray3
82 139 139	darkslategray4
47 79 79	darkslategrey
0 206 209	darkturquoise
148 0 211	darkviolet
255 20 147	deep pink
0 191 255	deep sky blue
255 20 147	deeppink
255 20 147	deeppink1
238 18 137	deeppink2
205 16 118	deeppink3
139 10 80	deeppink4
0 191 255	deepskyblue
0 191 255	deepskyblue1
0 178 238	deepskyblue2
0 154 205	deepskyblue3
0 104 139	deepskyblue4
105 105 105	dim gray
105 105 105	dim grey
105 105 105	dimgray
105 105 105	dimgrey
30 144 255	dodger blue
30 144 255	dodgerblue
30 144 255	dodgerblue1
28 134 238	dodgerblue2
24 116 205	dodgerblue3

RGB	Color Name
16 78 139	dodgerblue4
178 34 34	firebrick
255 48 48	firebrick1
238 44 44	firebrick2
205 38 38	firebrick3
139 26 26	firebrick4
255 250 240	floral white
255 250 240	floralwhite
34 139 34	forest green
34 139 34	forestgreen
220 220 220	gainsboro
248 248 255	ghost white
248 248 255	ghostwhite
255 215 0	gold
255 215 0	gold1
238 201 0	gold2
205 173 0	gold3
139 117 0	gold4
218 165 32	goldenrod
255 193 37	goldenrod1
238 180 34	goldenrod2
205 155 29	goldenrod3
139 105 20	goldenrod4
192 192 192	gray
0 0 0	gray0
3 3 3	gray1
26 26 6	gray10
255 255 255	gray100
28 28 28	gray11
31 31 31	gray12
33 33 33	gray13
36 36 36	gray14
38 38 38	gray15
41 41 41	gray16
43 43 43	gray17
46 46 46	gray18
48 48 48	gray19
5 5 5	gray2
51 51 51	gray20
54 54 54	gray21
56 56 56	gray22

RGB	**Color Name**
59 59 59	gray23
61 61 61	gray24
64 64 64	gray25
66 66 66	gray26
69 69 69	gray27
71 71 71	gray28
74 74 74	gray29
8 8 8	gray3
77 77 77	gray30
79 79 79	gray31
82 82 82	gray32
84 84 84	gray33
87 87 87	gray34
89 89 89	gray35
92 92 92	gray36
94 94 94	gray37
97 97 97	gray38
99 99 99	gray39
10 10 10	gray4
102 102 102	gray40
105 105 105	gray41
107 107 107	gray42
110 110 110	gray43
112 112 112	gray44
115 115 115	gray45
117 117 117	gray46
120 120 120	gray47
122 122 122	gray48
125 125 125	gray49
13 13 13	gray5
127 127 127	gray50
130 130 130	gray51
133 133 133	gray52
135 135 135	gray53
138 138 138	gray54
140 140 140	gray55
143 143 143	gray56
145 145 145	gray57
148 148 148	gray58
150 150 150	gray59
15 15 15	gray6

RGB	Color Name
153 153 153	gray60
156 156 156	gray61
158 158 158	gray62
161 161 161	gray63
163 163 163	gray64
166 166 166	gray65
168 168 168	gray66
171 171 171	gray67
173 173 173	gray68
176 176 176	gray69
18 18 18	gray7
179 179 179	gray70
181 181 181	gray71
184 184 184	gray72
186 186 186	gray73
189 189 189	gray74
191 191 191	gray75
194 194 194	gray76
196 196 196	gray77
199 199 199	gray78
201 201 201	gray79
20 20 20	gray8
204 204 204	gray80
207 207 207	gray81
209 209 209	gray82
212 212 212	gray83
214 214 214	gray84
217 217 217	gray85
219 219 219	gray86
222 222 222	gray87
224 224 224	gray88
227 227 227	gray89
23 23 23	gray9
229 229 229	gray90
232 232 232	gray91
235 235 235	gray92
237 237 237	gray93
240 240 240	gray94
242 242 242	gray95
245 245 245	gray96
247 247 247	gray97

RGB	**Color Name**
250 250 250	gray98
252 252 252	gray99
0 255 0	green
173 255 47	green yellow
0 255 0	green1
0 238 0	green2
0 205 0	green3
0 139 0	green4
173 255 47	greenyellow
192 192 192	grey
0 0 0	grey0
3 3 3	grey1
26 26 26	grey10
255 255 255	grey100
28 28 28	grey11
31 31 31	grey12
33 33 33	grey13
36 36 36	grey14
38 38 38	grey15
41 41 41	grey16
43 43 43	grey17
46 46 46	grey18
48 48 48	grey19
5 5 5	grey2
51 51 51	grey20
54 54 54	grey21
56 56 56	grey22
59 59 59	grey23
61 61 61	grey24
64 64 64	grey25
66 66 66	grey26
69 69 69	grey27
71 71 71	grey28
74 74 74	grey29
8 8 8	grey3
77 77 77	grey30
79 79 79	grey31
82 82 82	grey32
84 84 84	grey33
87 87 87	grey34
89 89 89	grey35

RGB	Color Name
92 92 92	grey36
94 94 94	grey37
97 97 97	grey38
99 99 99	grey39
10 10 10	grey4
102 102 102	grey40
105 105 105	grey41
107 107 107	grey42
110 110 110	grey43
112 112 112	grey44
115 115 115	grey45
117 117 117	grey46
120 120 120	grey47
122 122 122	grey48
125 125 125	grey49
13 13 13	grey5
127 127 127	grey50
130 130 130	grey51
133 133 133	grey52
135 135 135	grey53
138 138 138	grey54
140 140 140	grey55
143 143 143	grey56
145 145 145	grey57
148 148 148	grey58
150 150 150	grey59
15 15 15	grey6
153 153 153	grey60
156 156 156	grey61
158 158 158	grey62
161 161 161	grey63
163 163 163	grey64
166 166 166	grey65
168 168 168	grey66
171 171 171	grey67
173 173 173	grey68
176 176 176	grey69
18 18 18	grey7
179 179 179	grey70
181 181 181	grey71
184 184 184	grey72

RGB	Color Name
186 186 186	grey73
189 189 189	grey74
191 191 191	grey75
194 194 194	grey76
196 196 196	grey77
199 199 199	grey78
201 201 201	grey79
20 20 20	grey8
204 204 204	grey80
207 207 207	grey81
209 209 209	grey82
212 212 212	grey83
214 214 214	grey84
217 217 217	grey85
219 219 219	grey86
222 222 222	grey87
224 224 224	grey88
227 227 227	grey89
23 23 23	grey9
229 229 229	grey90
232 232 232	grey91
235 235 235	grey92
237 237 237	grey93
240 240 240	grey94
242 242 242	grey95
245 245 245	grey96
247 247 247	grey97
250 250 250	grey98
252 252 252	grey99
240 255 240	honeydew
240 255 240	honeydew1
224 238 224	honeydew2
193 205 193	honeydew3
131 139 131	honeydew4
255 105 180	hot pink
255 105 180	hotpink
255 110 180	hotpink1
238 106 167	hotpink2
205 96 144	hotpink3
139 58 98	hotpink4
205 92 92	indian red

RGB	Color Name
205 92 92	indianred
255 106 106	indianred1
238 99 99	indianred2
205 85 85	indianred3
139 58 58	indianred4
255 255 240	ivory
255 255 240	ivory1
238 238 224	ivory2
205 205 193	ivory3
139 139 131	ivory4
240 230 140	khaki
255 246 143	khaki1
238 230 133	khaki2
205 198 115	khaki3
139 134 78	khaki4
230 230 250	lavender
255 240 245	lavender blush
255 240 245	lavenderblush
255 240 245	lavenderblush1
238 224 229	lavenderblush2
205 193 197	lavenderblush3
139 131 134	lavenderblush4
124 252 0	lawn green
124 252 0	lawngreen
255 250 205	lemon chiffon
255 250 205	lemonchiffon
255 250 205	lemonchiffon1
238 233 191	lemonchiffon2
205 201 165	lemonchiffon3
139 137 112	lemonchiffon4
173 216 230	light blue
240 128 128	light coral
224 255 255	light cyan
238 221 130	light goldenrod
250 250 210	light goldenrod yellow
211 211 211	light gray
211 211 211	light grey
255 182 193	light pink
255 160 122	light salmon
32 178 170	light sea green
135 206 250	light sky blue

RGB	Color Name
132 112 255	light slate blue
119 136 153	light slate gray
119 136 153	light slate grey
176 196 222	light steel blue
255 255 224	light yellow
173 216 230	lightblue
191 239 255	lightblue1
178 223 238	lightblue2
154 192 205	lightblue3
104 131 139	lightblue4
240 128 128	lightcoral
224 255 255	lightcyan
224 255 255	lightcyan1
209 238 238	lightcyan2
180 205 205	lightcyan3
122 139 139	lightcyan4
238 221 130	lightgoldenrod
255 236 139	lightgoldenrod1
238 220 130	lightgoldenrod2
205 190 112	lightgoldenrod3
139 129 76	lightgoldenrod4
250 250 210	lightgoldenrod yellow
211 211 211	lightgray
211 211 211	lightgrey
255 182 193	lightpink
255 174 185	lightpink1
238 162 173	lightpink2
205 140 149	lightpink3
139 95 101	lightpink4
255 160 122	lightsalmon
255 160 122	lightsalmon1
238 149 114	lightsalmon2
205 129 98	lightsalmon3
139 87 66	lightsalmon4
32 178 170	lightseagreen
135 206 250	lightskyblue
176 226 255	lightskyblue1
164 211 238	lightskyblue2
141 182 205	lightskyblue3
96 123 139	lightskyblue4
132 112 255	lightslateblue

RGB	Color Name
119 136 153	lightslategray
119 136 153	lightslategrey
176 196 222	lightsteelblue
202 225 255	lightsteelblue1
188 210 238	lightsteelblue2
162 181 205	lightsteelblue3
110 123 139	lightsteelblue4
255 255 224	lightyellow
255 255 224	lightyellow1
238 238 209	lightyellow2
205 205 180	lightyellow3
139 139 122	lightyellow4
50 205 50	lime green
50 205 50	limegreen
250 240 230	linen
255 0 255	magenta
255 0 255	magenta1
238 0 238	magenta2
205 0 205	magenta3
139 0 139	magenta4
176 48 96	maroon
255 52 179	maroon1
238 48 167	maroon2
205 41 144	maroon3
139 28 98	maroon4
102 205 170	medium aquamarine
0 0 205	medium blue
186 85 211	medium orchid
147 112 219	medium purple
60 179 113	medium sea green
123 104 238	medium slate blue
0 250 154	medium spring green
72 209 204	medium turquoise
199 21 133	medium violet red
102 205 170	medium aquamarine
0 0 205	mediumblue
186 85 211	mediumorchid
224 102 255	mediumorchid1
209 95 238	mediumorchid2
180 82 205	mediumorchid3
122 55 139	mediumorchid4

RGB	Color Name
147 112 219	mediumpurple
171 130 255	mediumpurple1
159 121 238	mediumpurple2
137 104 205	mediumpurple3
93 71 139	mediumpurple4
60 179 113	mediumseagreen
123 104 238	mediumslateblue
0 250 154	mediumspring green
72 209 204	mediumturquoise
199 21 133	mediumvioletted
25 25 112	midnight blue
25 25 112	midnightblue
245 255 250	mint cream
245 255 250	mintcream
255 228 225	misty rose
255 228 225	mistyrose
255 228 225	mistyrose1
238 213 210	mistyrose2
205 183 181	mistyrose3
139 125 123	mistyrose4
255 228 181	moccasin
255 222 173	navajo white
255 222 173	navajowhite
255 222 173	navajowhite1
238 207 161	navajowhite2
205 179 139	navajowhite3
139 121 94	navajowhite4
0 0 128	navy
0 0 128	navy blue
0 0 128	navyblue
253 245 230	old lace
253 245 230	oldlace
107 142 35	olive drab
107 142 35	olivedrab
192 255 62	olivedrab1
179 238 58	olivedrab2
154 205 50	olivedrab3
105 139 34	olivedrab4
255 165 0	orange
255 69 0	orange red
255 165 0	orange1

RGB	Color Name
238 154 0	orange2
205 133 0	orange3
139 90 0	orange4
255 69 0	orangered
255 69 0	orangered1
238 64 0	orangered2
205 55 0	orangered3
139 37 0	orangered4
218 112 214	orchid
255 131 250	orchid1
238 122 233	orchid2
205 105 201	orchid3
139 71 137	orchid4
238 232 170	pale goldenrod
152 251 152	pale green
175 238 238	pale turquoise
219 112 147	pale violet red
238 232 170	palegoldenrod
152 251 152	palegreen
154 255 154	palegreen1
144 238 144	palegreen2
124 205 124	palegreen3
84 139 84	palegreen4
175 238 238	paleturquoise
187 255 255	paleturquoise1
174 238 238	paleturquoise2
150 205 205	paleturquoise3
102 139 139	paleturquoise4
219 112 147	palevioletred
255 130 171	palevoletred1
238 121 159	palevioletred2
205 104 137	palevioletred3
139 71 93	palevioletred4
255 239 213	papaya whip
255 239 213	papayawhip
255 218 185	peach puff
255 218 185	peachpuff
255 218 185	peachpuff1
238 203 173	peachpuff2
205 175 149	peachpuff3
139 119 101	peachpuff4

RGB	Color Name
205 133 63	peru
255 192 203	pink
255 181 197	pink1
238 169 184	pink2
205 145 158	pink3
139 99 108	pink4
221 160 221	plum
255 187 255	plum1
238 174 238	plum2
205 150 205	plum3
139 102 139	plum4
176 224 230	powder blue
176 224 230	powderblue
160 32 240	purple
155 48 255	purple1
145 44 238	purple2
125 38 205	purple3
85 26 139	purple4
255 0 0	red
255 0 0	red1
238 0 0	red2
205 0 0	red3
139 0 0	red4
188 143 143	rosy brown
188 143 143	rosybrown
255 193 193	rosybrown1
238 180 180	rosybrown2
205 155 155	rosybrown3
139 105 105	rosybrown4
65 105 225	royal blue
65 105 225	royalblue
72 118 255	royalblue1
67 110 238	royalblue2
58 95 205	royalblue3
39 64 139	royalblue4
139 69 19	saddle brown
139 69 19	saddlebrown
250 128 114	salmon
255 140 105	salmon1
238 130 98	salmon2
205 112 84	salmon3

RGB	Color Name
139 76 57	salmon4
244 164 96	sandy brown
244 164 96	sandybrown
46 139 87	sea green
46 139 87	seagreen
84 255 159	seagreen1
78 238 148	seagreen2
67 205 128	seagreen3
46 139 87	seagreen4
255 245 238	seashell
255 245 238	seashell1
238 229 222	seashell2
205 197 191	seashell3
139 134 130	seashell4
160 82 45	sienna
255 130 71	sienna1
238 121 66	sienna2
205 104 57	sienna3
139 71 38	sienna4
135 206 235	sky blue
135 206 235	skyblue
135 206 255	skyblue1
126 192 238	skyblue2
108 166 205	skyblue3
74 112 139	skyblue4
106 90 205	slate blue
112 128 144	slate gray
112 128 144	slate grey
106 90 205	slateblue
131 111 255	slateblue1
122 103 238	slateblue2
105 89 205	slateblue3
71 60 139	slateblue4
112 128 144	slategray
198 226 255	slategray1
185 211 238	slategray2
159 182 205	slategray3
108 123 139	slategray4
112 128 144	slategrey
255 250 250	snow
255 250 250	snow1

RGB	Color Name
238 233 233	snow2
205 201 201	snow3
139 137 137	snow4
0 255 127	spring green
0 255 127	springgreen
0 255 127	springgreen1
0 238 118	springgreen2
0 205 102	springgreen3
0 139 69	springgreen4
70 130 180	steel blue
70 130 180	steelblue
99 184 255	steelblue1
92 172 238	steelblue2
79 148 205	steelblue3
54 100 139	steelblue4
210 180 140	tan
255 165 79	tan1
238 154 73	tan2
205 133 63	tan3
139 90 43	tan4
216 191 216	thistle
255 225 255	thistle1
238 210 238	thistle2
205 181 205	thistle3
139 123 139	thistle4
255 99 71	tomato
255 99 71	tomato1
238 92 66	tomato2
205 79 57	tomato3
139 54 38	tomato4
64 224 208	turquoise
0 245 255	turquoise1
0 229 238	turquoise2
0 197 205	turquoise3
0 134 139	turquoise4
238 130 238	violet
208 32 144	violet red
208 32 144	violetred
255 62 150	violetred1
238 58 140	violetred2
205 50 120	violetred3

RGB	Color Name
139 34 82	violetred4
245 222 179	wheat
255 231 186	wheat1
238 216 174	wheat2
205 186 150	wheat3
139 126 102	wheat4
255 255 255	white
245 245 245	white smoke
245 245 245	WhiteSmoke
255 255 0	yellow
154 205 50	yellow green
255 255 0	yellow1
238 238 0	yellow2
205 205 0	yellow3
139 139 0	yellow4
154 205 50	yellowgreen

The XFDL Compute System

An XFDL compute is an expression that controls the character content of an element at or below the element depth of an XFDL option. XFDL computes can be defined for custom options that are not in the XFDL namespace (a common practice used for computing intermediate results), but in such cases, the attribute must be properly namespace qualified (e.g. use xfdl:compute where the namespace prefix ’xfdl’ is associated with the XFDL namespace URI). The XFDL compute expression appears in a *compute* attribute. This section defines the infix notation for XFDL compute expressions.

Most XFDL processors only need to preserve the compute as character data, but some applications must parse the text of computes and construct expression tree data structures to represent all computes in a form. This is necessary if the application must change the content of options or suboptions that are referred to by a compute. This section describes the syntax and operation of computes.

Whitespace in Computes

XFDL computes automatically support the notion of free form text found in most programming languages. With the exception of the contents of quoted strings (see “Quoted Strings” on page 422) and both static and dynamic references (see “XFDL References to Elements” on page 423), unlimited whitespace is permitted. Adding S? before and after every lexical token in every BNF rule in this section would unnecessarily obfuscate the presentation of what is essentially the standard BNF for mathematical and conditional expressions. Therefore, it is stated once here for the reader that all whitespace appearing outside of quoted strings and other lexical tokens is ignored.

While whitespace is formally ignored, there do arise cases in which it is necessary to use whitespace to properly communicate the expression. These situations arise because lexical analyzers are ’greedy’ in the sense that they will match as much of a substring to the current token as possible. For example, in the boolean test a==b and c==d, the spaces before and after the keyword ’and’ are required because the element reference token that matches ’a’, ’b’, ’c’ and ’d’ also matches ’band’, ’andc’ and ’bandc’.

The XFDL compute is a normal XML attribute and is therefore subject to all of the normal XML processing rules associated with attributes of the default type (CDATA). This includes whitespace normalization as well as entity and character reference resolution. An XML parser is expected to convert each tab or newline character into a space, so these characters should be avoided when text editing. XML processors capable of serialization are expected to encode tabs and new lines in attribute values as character references since the attribute value normalization on input implies that these characters would not be in the attribute value if they were not encoded as character references in the input. In raw XML text, a new line is encoded as &#xA; which a subsequent XML parser will decode into a new line character in the attribute value.

Multiline Computes

Due to XML attribute value normalization, linefeeds are converted to spaces. Therefore, normal XML processing turns a multiline compute that has been carefully laid out by its author into a single line compute. To alleviate this problem, form authors are encouraged to put the character reference &#xA; at the end of each line of a multiline compute attribute.

A normal XML processor that reads &#xA; followed by a new-line character will translate the value to a real new-line followed by a space. The serialization algorithm of an XML processor will then output the reference &#xA; for the real new-line followed by a space. A subsequent parse by an XML processor will produce a real new-line followed by a space. Therefore, it is recommended that XFDL processors (which uses a normal XML processor for parsing input) serialize a real new-line followed by a space as the reference &#xA; followed by a new-line, which this restores the multi-line appearance of computes. This behavior must not be performed when serializing the XML for a digital signature.

Structure of Mathematical and Conditional Expressions

An XFDL compute can be either a mathematical or conditional expression. A conditional expression has three parts separated by the ternary ?: operator. The first part is a Decision, which yields a boolean result. The consequences for a true and false boolean result recurse to the definition of Compute, permitting arbitrary nesting of decision logic.

[49]	`Compute ::= Expr | Decision '?' Compute ':' Compute`

The decision logic can apply logical-or (|| or 'or'), logical-and (&& or 'and'), and logical negation (!) to the results of logical comparisons. The logical operators are left associative, and the comparison operators cannot be chained (e.g. a < b < c is illegal). The order of operations gives greatest precedence to negation, then logical-and, and least precedence to logical-or. To override this, parentheses can be used (e.g., the parentheses in (a<b || c<d) && e!=f cause the logical-or to occur first, and no parentheses are required if the logical-and should be performed first).

[50]	`Decision ::= Decision ('||' | 'or') AndDecision | AndDecision`
[51]	`AndDecision ::= AndDecision ('&&' | 'and') NotDecision | NotDecision`
[52]	`NotDecision ::= '!' Comparison | Comparison`
[53]	`Comparison ::= '(' Decision ')' | Expr ('<' | '>' | '<=' | '>=' | '==' | '!=') Expr`

A mathematical expression, denoted Expr, can include addition, subtraction, string concatenation (+.), multiplication, division, integer modulus, unary minus, and exponentiation. All mathematical operators are left associative except unary minus and exponentiation. Further, proper order of operations is observed. Parentheses can be used to override the order of operations as shown in the non-terminal symbol named Value (defined later).

[54]	`Expr ::= Expr '+' Term | Expr '-' Term | Expr '+.' Term | Term`
[55]	`Term ::= Term '*' NFactor | Term '/' NFactor | Term '%' NFactor | NFactor`
[56]	`Nfactor ::= Factor | '-' Factor`
[57]	`Factor ::= Value '^' NFactor | Value`

Table of Operators

The following table details the operators permitted in XFDL:

Type of Operator	Symbol	Operation
Additive	+	addition
	- (minus)	subtraction
	+.	concatenation
Multiplicative	*	multiplication
	/	division
	%	modulus (returns remainder)
Exponentiation	^	exponential
Relational	>	greater than
	<	less than
	<=	less than or equal to
	>=	greater than or equal to
	==	equal to
	!=	not equal to
Logical	&&	AND
	and	AND
	| |	OR
	or	OR
	! **Note:** *The reserved words* 'and' *and* 'or' *are case sensitive. Always use lower case.*	NOT
Unary Minus	- (minus)	take negative
Decision	x?y:z	assign the value of expression **y** to the result if expression **x** evaluates to true. Otherwise, assign the value of expression **z** to the result.
Assignment	=	assign right operand to left operand
Membership	. (dot)	structure membership
	[ ]	array membership
	->	indirect membership

Precedence of Operations

Operations are evaluated in the following order:

- membership

- exponentiation
- multiplicative and unary minus
- additive
- relational
- logical NOT
- logical AND
- logical OR
- conditional

Decision Operations and Namespace Qualification

Both decision operations and namespace use the colon (:) character as a token. In some cases, this makes it difficult to determine which colon is the decision operator token and which colon is the namespace token. For example, consider the following expression:

```
x ? custom:y : z
```

This is a variation on the common ″if x then y, else z″ expression. In this expression, the y statement is intended to be in the ″custom″ namespace. However, as it is currently written the expression does not clearly define which colon represents namespace and which colon represents the end of the y statement.

In cases such as these, parentheses should enclose the y statement of the expression, thereby providing a clear indication of where the y statement ends. For example:

```
x ? (custom:y) : z
```

Illegal Characters in XML Attributes

Note that the ampersand (&) and less-than (<) characters are not permitted in XML attribute values. Since XFDL computes appear in a compute attribute, these must be escaped with character or entity references (e.g. the entity references &amp; for the ampersand and &lt; for the less-than character). Hence, the less-than-or-equal symbol (<=) could be encoded as ’ &lt;=’.

Definition of Value

A value can be any of:

- a compute in parentheses, which provides an override for the order of operations.
- a quoted string (see “Quoted Strings”).
- an XFDL reference to an element whose text data should be obtained when the compute is evaluated (see “XFDL References to Elements” on page 423).
- the result of a function call (Section 3.8 Function Call Syntax).

```
[58]    Value ::= '(' Compute ')' | qstring | XFDLReference | FunctionCall
```

Quoted Strings

A quoted string is used to express a literal value in XFDL. The language rules for computes permit the recognition of a quoted string token using the italicized token name *qstring*. Whitespace before the open quote and after the close quote is ignored. Because XML allows attribute values to be either singly or doubly quoted,

the XFDL compute expression syntax allows both single and double quotes so that the XFDL author can avoid the use of entity references. Within an XFDL quoted string, any character is allowed except for the type of quote mark used to start the quoted string. Quoted strings can also be of arbitrary length in XFDL. To increase human readability, XFDL supports multiline string continuation. If the next non-whitespace character appearing after a closing quote is an open quote, then the closing quote, whitespace, and open quote are discarded from the input stream.
.

```
[59]    qstring ::= (('"' [^"]* '"')  | (''' [^']* '''))+
```

A literal must be quoted regardless of its type (i.e. character strings, numeric values, dates and so forth must all be quoted). However, the quoted string is in the XFDL compute attribute value, which is a quoted string in XML. Therefore, if double quotes are used to surround the compute expression, then literals must either be expressed with single quotes or by using character or entity references for double quotes (e.g. the entity reference &quot;). Likewise, if the compute attribute is surrounded by single quotes, then either double quotes must be used for the quoted strings or character or entity references (such as &apos;) must be used to express quoted strings.

Occasionally a literal value simply must include the type of quote mark used to surround the literal. In such cases, the XFDL author can either use entity references to surround the literal value or use backslash (\) escaping. For example, \' can be used to place a single quote in an XFDL quoted string that is surrounded by single quotes.

Backslash escaping is also supported for several other characters. The escape sequences `\n` and `\t` result in a new line and a tab, respectively, in the quoted string content. Since the backslash is the escaping character, it must also be escaped to be inserted into the string content (e.g., `\\`).

Note that since a quoted string is meant to be interpreted by the XFDL compute expression parser rather than the XML processor, it is recommended that backslash escaping be used rather than XML character and entity references. However, it is safe to use the XML mechanisms except for encoding a new-line in a quoted string. The special processing performed by XFDL processors to preserve the text layout of multiline computes may conflict with the use of an XML character reference for a new-line if it is followed by a space (please see "Multiline Computes" on page 419). While correct processing will still result due to attribute value normalization on the next parse, the text layout and surface string of the XFDL will be changed.

XFDL References to Elements

The simple character content of options and suboptions are obtained as the operands of XFDL compute expressions using XFDL references. XFDL references support forward and backward referencing. An XFDL reference can refer to any option or array element with simple character content.

The element containing the desired character content is identified using scope identifiers to negotiate a path through the parse tree. To traverse through the page and item levels and identify an option, the well-known 'dot' membership operator is used. The well-known square-bracket array notation can then be used to access suboptions to arbitrary element depth. For example, *Page2.Field2.value* would access the option with tag name <value> in the <field> element having a *sid* of **Field2** in the <page> having a *sid* of **Page2**.

Because each XFDL element's scope identifier (*sid*) is used to uniquely identify an element only within the surrounding parent element, XFDL can support relative referencing. For example, in an element identified as Field1, if a computation includes the reference *Field2.value*, this means that the character data of the value option in the item Field2 *on the same page* will be obtained. If Field2 is on a separate page, say Page2, then a compute in Field1 can still access its value using the fully-qualified reference *Page2.Field2.value*.

The context for interpreting a reference is also decided based on the form of the reference itself in combination with its location in the form. For example, in the XFDL below, the reference *Bill.value* appears in a compute that is attached to a grand-child suboption of the format option. However, the name after the rightmost 'dot' operator always refers to an option, and the name before the rightmost dot always refers to an item. Since the page is not specified, it is determined to be the page containing the reference.

```
<page sid= "CreditCardApp">
   <field sid="Bill">
      <label>Your monthly bill is:</label>
      <value>700</value>
      <readonly>on</readonly>
   </field>
   <field sid="MinPayment">
      <label>Enter payment amount:</label>
      <value></value>
      <format>
         <datatype>currency</datatype>
         <range>
            <min compute="Bill.value * '0.05'">35</min>
            <max compute="Bill.value">700</max>
         </range>
      </format>
   </field>
</page>
```

XFDL references can also grow arbitrarily below the option level using the array notation, allowing access to unbounded array element depth within any option. If an array element is not named, then the zero-based numeric position of the array element is used in the square brackets. If the array element is named, then the scope identifier can be used in the square brackets. For example, given the format option of the XFDL above, the reference *format[0]* yields **dollar** and the reference *format[range][1]* yields **700**. If a suboption is named, the numeric position can still be used, e.g. *format[1][1]* also yields **700**.

The above description covers static references. The XFDL referencing model also supports dynamic references. The left associative 'arrow' operator (->), also known as the indirect membership operator, expects to receive a static or dynamic reference as a left operand. The run-time value of the static or dynamic reference must conform to the syntax of the ItemRef non-terminal. The right operand of the indirect membership operator is an option reference. At run-time, the left operand is evaluated, yielding a static item reference to an XML element representing an XFDL item. This run-time item reference is combined with the right operand of the indirect membership operator to yield an option or array element whose simple data is the result of the evaluation.

The simplest example of a dynamic reference is retrieving the text of the selected cell in an XFDL list box or popup, as is discussed in “Details on Items” on page 35, because the *value* option of a list or popup is equal to the item reference of the selected cell item. Thus, given an example popup that offers a selection of days of the week, the text for the day of week selected by the user is obtained by *Popup_DayOfWeek.value->value*.

An option reference can simply refer to element tag names in the XFDL namespace without any namespace qualification. A namespace qualified scope identifier (the non-terminal NSsid below) must be used if the element being referenced is not in the XFDL namespace. In order to reference element in the empty namespace, XFDL supports the predeclared prefix *null*. For example, to reference an element *E* which has an empty namespace URI, use the reference *null:E*.

Below are the syntax rules for an XFDL reference. Note that unlike most other syntax rules for XFDL expressions, intervening whitespace is not allowed in an XFDL reference. An XFDL reference is treated as a single lexical token.

[60]	`XFDLReference ::= StaticRef | StaticRef '->' DynamicRef`
[61]	`StaticRef ::= ItemRef '.' OptionRef | OptionRef`
[62]	`ItemRef ::= ((sid '.')? sid '.')? sid`
[63]	`DynamicRef ::= DynamicRef '->' OptionRef | OptionRef`
[64]	`OptionRef ::= NSsid ('[' (Digit+ | NSsid) ']')*`
[65]	`NSsid ::= sid | (Letter (Letter | Digit | '_')*) ':' sid`

Referencing the XFDL Version

Since the XFDL version is represented by the XFDL namespace URI declaration in the root XFDL node, the normal *page.item.option* notation cannot reference the version number directly. However, as a convenience, it is allowable to refer the XFDL version as a global form option, as follows:

```
global.global.version
```

XFDL processors are expected to recognize this notation.

Function Call Syntax

Function calls run code that may be external to the XFDL form definition. A set of predefined functions (called *system* functions) for doing standard mathematical operations, string manipulations, and so on, is given in “Details on Function Calls” on page 279. The LibName allows functions to be grouped into separate namespaces, but the predefined system functions do not require a LibName. The names of the system functions are considered reserved words and should not be used as function names in other function libraries.

[66]	`FunctionCall := (LibName '.')? FunctionName '(' (Compute (',' Compute)*)? ')'`
[67]	`LibName ::= sid`
[68]	`FunctionName ::= sid`

Representing and Running XFDL Computes

Introduction

The XFDL compute engine implements a ’declarative’ computation system. The behavior of the algorithm is similar to that of a spreadsheet. Complicating factors for XFDL include dynamic references and side effect functions such as *set()*. When a form is first started, many nodes with computed values may need to be evaluated to find current display values. After the form is started, any specific change to a given node should result in updates to computationally dependent nodes. Further, the process is recursive in that when a computationally dependent node is updated, then it may have further computationally dependent nodes that must be queued for update.

The XFDL specification provides an abstract version of the desired algorithm, but does not place specific constraints on the data structure used to represent the compute system.

Cached Dependency Lists

On form startup, the compute system associates with each form node *F* a list of other form nodes that are computationally dependent on *F*. A node is dependent on *F* if a reference to *F* appears in the node’s compute. Figure 1 expresses the XFDL compute engine algorithm based on utilizing these dependency lists. The dependency lists are viewed as a *directed graph*, or digraph, of computational dependencies.

Each form node will have several flags associated with it: Visited, OnStack, Processed, Processing, and UsedToDeref. The **Visited** flag is require by depth first search. The **OnStack** flag prevents duplicate entries on the stack, which prevents circular referencing. The **Processed** flag is used in conjunction with OnStack to allow a form node to stay on the stack but not be processed again when the form node’s dependencies have all been processed. The **pProcessing** flag is used when *XFDLRunComputes()* is called recursively from within the *eval()* function (side effect functions such as *set()* call *UFLSetLiteral()*, so the compute system is called recursively during their evaluation). The Processing flag tells *XFDLRunComputes()* to terminate, returning control to *eval()*. The **UsedToDeref** flag indicates whether a given form node’s literal value is being used by any compute in the form to dereference another form node (explained in the next section). If this flag is set, then changing the node’s literal will cause a change to the dependencies in the form.

The dependency lists referred to in Figure 1 are constructed at build time. For each compute C_I, the function *CreateRefList()* traverses the compute parse tree for all references to existing nodes. For each reference in C_I to a form node F_R, the form node F_P (the parent node containing C_I) is added to the dependency list of F_R, except when F_P is equal to F_R (this exception allows support for self-referential computes as discussed below). Thus, when a node F_R changes, the new algorithm has a prebuilt list of dependent computes that must be re-evaluated.

The outermost loop of the new algorithm runs until it reaches quiescent state (in other words, it achieves closure on the change to element E). When E is null, the new algorithm pushes all computes for reevaluation. The inner loops of the original algorithm in Figure 1 were mainly designed to find which computes were pertinent (in other words, which ones needed to be re-evaluated). In the new algorithm, the new pertinent vertices of a changed vertex F_P are immediately known due to the precomputation of the dependency lists.

The implied behavior of the outermost loop is that it should terminate if there is a circular reference other than a self-reference. By omitting a node from its own dependency list, self-referential computes do not become circular references. However, circular references involving more than one node can be expressed using static references (for example, A=B, B=C, C=A), dynamic references, or even indirect referencing via XFDL function calls (for example, the assignment X = A+set("A", A+ "1")). To prevent circular referencing, the algorithm uses a ChangeStack rather than a ChangeList (queue), and it uses the flags OnStack and Processed to implement a *depth first search*, which is an algorithm that finds cycles in a search space. *Push()* will not push a node already on the stack.

```
RunXFDLComputes(F (form), C (computes), E (a changed element or null)) ::=
If E is not null, then ProcessLiteralChange(F, E)
Else      For each c ∈ C, ParentFormNode(c).Visited = 0
         For each c ∈ C,
            If not ParentFormNode(c).Visited,
               DFSPush(F, ParentFormNode(c))
      For each c ∈ C, ParentFormNode(c).Visited = 0
While not Empty(F.ChangeStack) and not TopStack(F.ChangeStack).processing Do
   $F_P$ = TopStack(F.ChangeStack)
   If $F_P$.Processed, then Pop(F.ChangeStack);
   else      $F_P$.processing = on
         Literal = eval(ChildComputeNode($F_P$), F.ChangeStack)
          If cval($F_P$) ≠ Literal, then
            cval($F_P$) = Literal
             ProcessLiteralChange(F, $F_P$)
         $F_P$.processing = off
         $F_P$.Processed = on
ProcessLiteralChange(F, E) ::=
If E.UsedToDeref, then
   ProcessDependencyChanges(F, E)
DFSPush(F, E)
For each entry e of F.ChangeStack from top to bottom,
   If e.Visited, then e.Visited = 0
   Else break loop
F.TotalChangeList = F.TotalChangeList ∪ E
DFSPush(F, E) ::=
E.visited = 1
For each f ∈ DependencyList(E)
   If not f.Visited and not f.OnStack
      DFSPushComputes(F, f)
Push(F.ChangeStack, E)
ProcessDependencyChanges(F, E) ::=
For each f ∈ DerefSubset(DependencyList(E)),
   NewRefList = CreateRefList(f, F)
   For each r ∈ (f.RefList ∪ NewRefList) - (f.RefList ∪ NewRefList)
      If r ∈ f.RefList, then
          DependencyList(r) = DependencyList(r) - f
      Else
         DependencyList(r) = DependencyList(r) + f
   f.RefList = NewRefList
Pop(S) ::= f = S.Pop(); f.OnStack = f.Processed = off
Push(S, f) ::= if not f.OnStack, then
      S.Push(f);
      f.OnStack=on;
      f.Processed=off
```

Figure 3. Algorithm Sketch for XFDL Compute Engine

Note, however, that we do not generate an error when a duplicate is found on the change stack. This is because the loop initialization pushes all computes when E is null. Thus, there are valid cases where we want to ignore the duplication without generating an error. Reporting true circular references can be done using a depth first search at design time with the compute system turned off.

Topological Sorting

When a node E changes, the function *ProcessLiteralChange()* does not simply push the elements in its dependency list, which would make **XFDLRunComputes** a simple depth first search of a digraph. Instead, it calls *DFSPush()*, which explores the computational dependencies of E using a depth first search. In other words, E is treated as the root of a depth first search tree of computational dependencies. *DFSPush()* does not push E until it has visited all of E's descendants (post-order visitation of the DFS tree). This has the effect of placing the dependencies in a linear order on the change stack (linear ordering in a directed acyclic graph is called topological sorting). Since an element E appears on the stack above its descendants, it is re-evaluated before its descendants. Since E's descendants are dependent on E, evaluating E before its descendants ensures that E will have the correct value before its descendants are evaluated.

Because the dependency graph is directed, it is still possible for a depth first search to take exponential time because it must explore a subtree rooted at *r* for each parent of *r*. The directed graph can be acyclic even though the corresponding undirected graph has cycles. If those cycles have a repeated structure (such as might be found in a form with many rows of identical construction), the work on each row can be constant but the work of a row may occur once for each change to the preceding row. If each row requires at least two units of work, then the computation is exponential in the number of rows.

Handling Dynamic References

A dynamic reference is a compute that includes the use of the arrow operator to obtain a value by dereferencing other values in the form. Dynamic references imply changes to the dependency lists during run-time. A number of enhancements are required to solve this problem efficiently.

The dependency list of each form node will be segregated into normal dependencies and dereference dependencies. The form node's UsedToDeref flag will be set if and only if its dependency list contains dereference dependencies. After changing the literal value of a form node whose UsedToDeref flag is set, certain dependencies will need to be re-evaluated (described below).

For each form node f containing a compute, we store a ReferenceList containing all form nodes referred to within the compute. Again, the list will be segregated into normal and dereference references. All subreferences in a dynamic reference except for the rightmost reference are classified as dereference dependencies. For example, in a compute containing *popup.value->value*, if the popup's value is "cell1", then the literal of *popup.value* is dereferenced, but *cell1.value* is not. This distinction is important because a change to *popup.value* will cause dependency changes whereas changing *cell1.value* will not.

Actually, the reference list of each compute must already be built as part of setting up the dependency lists. The reference lists are required since, for each form node F_P in f.ReferenceList, we must add *f* to F_P.DependencyList. Now we are simply deciding to retain the list for use in solving problems introduced by dynamic references. So, while building dependency lists, if F_P is classified as a dereference in *f*.ReferenceList, then *f* will be classified added a dereference dependency in F_P.DependencyList.

The *ProcessDependencyChanges()* function mentioned in Figure 1 can then rebuild the dependency lists efficiently. When processing a stack entry F_P, if the UsedToDeref flag is set, then some dependency lists may need to be changed. A form node *f* is considered to be in the ’dereference’ portion of F_P.DependencyList. Begin by creating a new reference list for *f*. Any form node F_P in *f*.ReferenceList but not in the new reference list implies the removal of the dependency entry *f* from F_P.DependencyList. Any form node F_P not in *f*.ReferenceList but in the new reference list implies the addition of the dependency entry *f* to F_P.DependencyList. Finally, the new reference list is assigned to *f*.ReferenceList.

Note that most of the dependency changes involve the addition and deletion of ’normal’ references and dependencies. For example, given a form node *f* with a compute of *popup.value->value*, if *popup.value* were to change from ″cell1″ to ″cell2″, then the ’normal’ entry of ″cell1″ in the *f*.ReferenceList would be deleted, and a ’normal’ entry of ″cell2″ would be added. Furthermore, the normal dependency on *f* would be removed from in *cell1.DependencyList* and added to *cell2.DependencyList*. It is possible that reference list and dependency list entries of type ’dereference’ can also be modified. An example would be a double dereference; in other words, the change of *x.value* in a form containing the compute *x.value->value->value*. Thus, an entry *f* in F_P.DependencyList is a dereference if changing node F_P implies the need to rebuild *f*.ReferenceList (which in turn implies changes to other dependency lists, possibly including F_P.DependencyList).

The statements above assert that the algorithm will only rebuild the references lists of nodes marked as ’dereference’ dependencies in F_P.DependencyList. Due to the high cost of calling *UFLDereference()*, the algorithm should not rebuild reference list entries that result from static references. Static reference nodes are distinguished from dynamic reference nodes in the compute tree. Note that the leftmost subreference of a dynamic reference is also static, so it will also not be recomputed. Instead, the value in the reference cache will be used (see section “Reference Caching”).

Reference Caching

When a compute is parsed into a parse tree of compute nodes, the nodes representing static references currently cache the results of *UFLDereference()* so that future evaluations of the compute can proceed without a costly search of the form. Although the ReferenceList of each node contains all references, these cached references are stored in compute nodes for instantaneous access by *eval()*.

In the current API, caching of dynamic references was not performed because the implementation had no way of knowing whether a change occurred that would affect the validity of the cached value. The algorithm is able to cache dynamic references. When the ReferenceList of a given form node is reconstructed, the dynamic references in the compute node associated with that form node will also be re-evaluated and re-cached.

Note that dynamic references call *UFLDereference()* once for the leftmost subreference plus once per arrow in the dynamic reference. The reference cache will cache the last *UFLDereference()* as this is the final results required by the *eval()* function. However, the first *UFLDereference()* on the leftmost subreference will also be cached. The leftmost subreference is static, so storing it can be used to optimize the process of resolving dependency changes (see “Handling Dynamic References” on page 428).

Re-entrancy

XFDLRunComputes() is no longer shown as returning a value. This is because the returned change list, formerly denoted Z, is actually a property of the form F and is now denoted F.TotalChangeList. This member is initialized to emptiness when the form is first created. It is updated by running computes, but it is only reinitialized to emptiness when an application takes ownership of the list from the form. This allows an application to make numerous changes to the form before using the F.TotalChangeList to update structures external to the form (such as a database or a corresponding GUI).

The change list itself is now also a property of the form, denoted F.ChangeStack, and it also is initialized to emptiness during form creation. Notice that it is not initialized to emptiness at the start of run computes. This is done to support re-entrancy. If the call to *eval()* runs a side effect function that in turn calls *UFLSetLiteral()*, then *XFDLRunComputes()* will be reinvoked. The new function instance will continue attempting to process the same change stack. Since the change stack forbids duplicates, *XFDLRunComputes()* cannot recurse indefinitely on a form of bounded size (side-effect functions such as *duplicate()* can create new pieces of form with each run, so if the *duplicate()* duplicates itself, an infinite loop can occur). When the instance of *XFDLRunComputes()* that was called recursively encounters a stack entry marked ’processing’, this indicates that it is time to return back to *eval()*, which ultimately returns to the previous instance of *XFDLRunComputes()*.

Duplicate Entries on the Total Change List

The algorithm in Figure 1 explicitly shows how it uses the OnStack and Processed flags of each form node to prevent duplicate entries in the change stack. The algorithm also uses the set union operator to indicate that duplicates will not be allowed on the TotalChangeList. This is done using another flag, OnChangeList, which is cleared on form node creation. An attempt to add a form node to the TotalChangeList is preceded by a test of this flag. If it is set, then the form node is already on the change list and will not be added again. If the flag is clear, then the form node will be added to the TotalChangeList, then its OnChangeList flag will be set.

When the API function *UFLGetChangeList()* is called, the OnChangeList flags of each element in the TotalChangeList must be cleared before passing ownership of the TotalChangeList from the form to the API caller.

Missing References

If an XFDL reference refers to a non-existent element or an element with array content, then the reference simply resolves to the empty string. If a dynamic subreference (i.e. the left operand of the dereference operator) refers to a non-existent element, then the containing compute is disabled. If the desired form node is created later, the compute is immediately re-activated.

Handling of Element Deletion

When an element is deleted, it may contain a compute. The XFDL processor must remove dependency list entries associated with the elements referenced by the compute expression being destroyed. Moreover, references to the element being deleted may appear on change lists and reference caches in the compute system. These must be removed as well. If a compute refers to an element that is being deleted, then the compute will be handled according to the rules for missing references (see the previous section for how these are handled).

Limitations

It is possible to create cases in which the algorithm halts a sequence of computations that do not technically form circular logic, especially when side effect functions such as *set()* are used. In other words, the case would halt by itself if the algorithm would allow duplicates on the change stack. However, there is a difference between circular logic and circular references. According to the XFDL specification, circular referencing is forbidden (except for self references), and the compute system simply does not continue evaluation of a node C that is already on the stack.

A second limitation, also involving side effect functions, can actually cause an infinite loop in the algorithm. The problem occurs when a side effect function creates a new portion of the form. If the new portion of the form contains computations that cause the continued recursive creation of new portions of the form, then each new portion of form has elements that are distinct from all previous elements and hence do not technically even cause a circular reference. A depth first search halts on any form of finite size, but if computations result in unbounded form growth, then it is correct behavior for a depth first search to run indefinitely exploring the new regions of the form.

A third limitation is that a given compute may still be fired more than once. This is not the result of circular reference in the dependency digraph but rather that the same descendant is reachable along multiple paths (the undirected graph corresponding to the dependency digraph has cycles, but the digraph contains no way to get from the descendant back to the ancestor). In other words, a node *r* has multiple parents because a depth first search of a digraph can yield multiple DFS trees each containing *r* with one of its parents.

To solve this problem with static references only, a linear ordering could be created for the directed acyclic graph of dependencies, and the edges of each node could be resorted (in linear total time) so that the push order would respect the linear ordering. However, this approach is impractical for XFDL. Due to dynamic referencing, the linear ordering would need to be maintained dynamically. Furthermore, dynamic changes to the linear ordering would cause dynamic changes to the order of elements in the change stack. More advanced methods could be developed to account for this, but the problem expands when the implicit dependencies introduced by side effect functions are taken into account. Although the compute system could run faster by running less computes, the cost of operations necessary to account for dynamic references is prohibitive, and implementing the method would not be worth the trouble due to the inability to achieve correctness with side effect functions. Thus, the linear ordering imposed by *DFSPush()* is temporal, and changes to nodes used in dynamic references may invalidate the order such that extra computes will sometimes run more than once. Those who wish to prevent functions such as viewer.messageBox from running more than once must still resort to protecting these computes with conditional logic and the *toggle()* function.

XForms and XFDL Computes

XForms provides its own methods to compute values for data in that is in the XForms model. However, in same cases it may be either prefereable or necessary to use XFDL computes. For example, XFDL computes are required to make changes to the presentation layer that are not related to data, such as color changes and so on.

In general, using XForms computes to manipulate data and XFDL computes to manipulate the presentation layer will create a clean separation of duties that creates few conflicts. However, be aware that when a form is first loaded, the XForms engine overwrites all *value* and *rtf* options that are linked to the model by a single node binding. This means that any XFDL computes on those options will be removed from the form.

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.

Trademarks

Index

Program Number:

Printed in USA

S325-2600-00