June 8, 2007

XML Forms Architecture (XFA) Specification

Version 2.5

Contents

Preface

This specification is a reference for XML Forms Architecture (XFA). It is intended for use in developing applications that create XFA templates (which represent forms awaiting fill-in) and applications that process XFA forms. Such XFA processing applications may be simple stand-alone form-fill in applications, or they may be a set of client-server applications that work together to fill-in and process a form.

Intellectual Property

The general idea of using templates and processing rules to build interactive forms is in the public domain. Anyone is free to devise templates using unique structures and apply customized processing rules to them. However, Adobe Systems Incorporated owns the copyright for the particular template-based grammar and processing rules constituting the *XFA Specification*, the written specification for the Adobe XML Architecture. Thus, these elements of the *XFA Specification* and Adobe XML Architecture may not be copied without Adobe's permission.

Adobe will enforce its copyrights Adobe's intention is to maintain the integrity of the Adobe XML Architecture standard. This enables the public to distinguish between the Adobe XML Architecture and other interchange formats for electronic documents, transactions and information. However, Adobe desires to promote the use of the Adobe XML Architecture for form-related interactions among diverse products and applications. Accordingly, Adobe gives anyone copyright permission to use the Adobe XML Architecture, subject to the conditions stated below, to:

- Prepare files whose content conforms to the Adobe XML Architecture
- Write drivers and applications that produce output represented in the Adobe XML Architecture
- Write software that accepts input in the form of the Adobe XML Architecture specifications and displays, prints, or otherwise interprets the contents
- Copy Adobe's copyrighted grammar, as well as the example code to the extent necessary to use the Adobe XML Architecture for the purposes above

The condition of such intellectual property usage is:

- Anyone who uses the copyrighted grammar, as stated above, must include the appropriate copyright notice.

This limited right to use the example code in this document does not include the right to use other intellectual property from Adobe, or the software in any of Adobe's products that use the Adobe XML Architecture, in whole or in part, nor does it include the right to use any Adobe patents, except as may be permitted by an official Adobe Patent Clarification Notice (see [Adobe-Patent-Notice] in the Bibliography).

Adobe, the Adobe logo, and Acrobat are either trademarks or registered trademarks of Adobe Systems Incorporated in the United States and/or other countries. Nothing in this document is intended to grant you any right to use these trademarks for any purpose.

Document Contents

This reference is a complete guide to XFA. It describes the various XML grammars that comprise XFA and it explains the rules for processing those grammars in conjunction with data supplied from sources outside the XFA form.

This reference is presented in the following major parts:

- Part 1: XFA Processing Guidelines. This part contains narrative chapters that introduce XFA and provide rules for processing XFA in conjunction with data received from an outside source. These rules provide a standard interpretation of XFA expressions and data, which helps to ensure that XFA-related behavior in XFA processing applications is equivalent. More importantly it helps to ensure that XFA processing applications that contribute to sequential processing of XFA documents do not have mis-matched expectations.
- Part 2: XFA Grammar Specifications. This part provides a set of references that describe the elements and attributes of each of the grammars that comprise XFA. Each chapter describes one of these grammars.
- Part 3: Other XFA-Related References. Each chapter in this part contains reference material for non-XML expressions used with XFA. Although the standards described in these chapters are an important part of XFA processing, they are not considered XFA grammars.
- Part 4: Appendices, Bibliography, Glossary and Index. This part contains appendices that provide adjunct information referenced by the narrative chapters in Part 1. It also contains a bibliography, a glossary and an index.

Intended Audience

You should read this specification if you are developing a template designing application or if you are developing an XFA processing application. This is especially true if either type of application is intended to work with the Adobe® XFA-compatible products, such as Acrobat® and LiveCycle® Designer.

Non-technical readers may benefit from reading the chapter, "Introduction to XML Forms Architecture (XFA)" on page 15.

Perspective Used in Describing Processing Guidelines

This document is written from the perspective of an XFA processing application. That is, this document describes the steps an XFA processing application must perform to properly interpret XFA, especially in the context of accepting outside data into an XFA form.

The narrative in this document describes the XFA processing rules as through the processing application were using XML Document Object Models (DOMs) as its internal representation of XFA constructs and of data. It also assumes all property references are in terms of objects. While such representations are not required, they are the most common way of implementing code that processes XML data.

Notwithstanding this document's focus on DOMs and objects, nothing in this specification demands that the same internal data structures be employed by any particular implementation. Similarly, notwithstanding the use of terms such as "object" associated with object-oriented languages, nothing in this specification constrains what programming language(s) may be used by any particular implementation. However conforming implementations must provide the same external functionality and must employ the same external data structures.

Associated Schemas

Many of the XFA grammars described in this specification are contained in an attachment carried within this PDF file. See "Schemas" on page 1085 for instructions concerning how to extract and use those schemas.

Although these schemas can be used to validate the XML syntax of XFA documents, such validation is not normally part of XFA form processing. Most people filling out a form cannot resolve errors detected during such validation. It is expected that XFA documents will be generated by software and will be correct by design. Hence, these schemas are more likely to be used in development.

XML validation differs from XFA form validation, which validates the content entered into the form. XFA form validation is described in "Validations" on page 329.

XFA also supports the use of XML data documents which are or may be separate from the form itself. A simple schema language based on annotated sample data is defined for this purpose in "Data Description Specification" on page 834. This facility is not used for validation as such; data which does not conform to the data description is simply ignored. However the data description does control the merging of data into the form and the subsequent generation of a new XML data document. See "Exchanging Data Between an External Application and a Basic XFA Form" on page 108.

Related Documentation

This document replaces the previous version of this specification. This version and previous versions back to 2.0 are available at http://adobe.com/go/xfa_specifications.

What's New

A complete list of enhancements in XFA versions 2.0 through 2.5, labelled by version, is given in the appendix "History of Changes in This Specification" on page 1070.

Caution: In XFA 2.5 some previously endorsed syntax is still legal, but deprecated. It will be removed in a future version of the specification. See "Deprecated XFA Template Features" on page 1084.

Conventions

This document uses notational and graphical conventions as a shorthand for conveying more complex information.

Notational Conventions

This document uses typefaces and character sequences to indicate the roles and connotations of expressions.

Typefaces

The following table describes typeface usage in this document:

Typeface	Identifies ...
`monospaced`	XML and XFA expressions: `<abc>apple</abc>` Named XML and XFA objects that appear in a paragraph: A `pageSet` element represents an ordered set of display surfaces. **Note:** Named XFA objects in a paragraph are frequently not tagged with the monospaced typeface because their identity as such is assumed to be understood.
italics	Definition of a term: *Fixed data* (*boilerplate*) includes any text, lines, … that remain unchanged throughout the life of the form. Document title: *PDF Reference*
Hypertext link	Hypertext links to other parts of this document: …, as described in "Conventions" on page x. Hypertext links to references in the "Bibliography" on page 1088: …, as described in the *PDF Reference* [PDF]. Hypertext links to element descriptions that appear in one of this document's references: For more information see the `field` syntax description. Hypertext links to URLs: Those notations are available at http://www.unicode.org/uni2book/Preface.pdf.

Unicode Character Codes

Character codes are given using the notation described in the preface to *The Unicode Standard, Version 3.0*. Those notations are available at http://www.unicode.org/uni2book/Preface.pdf, p. xxvii. Character names are as given in the Unicode character tables.

Document Object Model Notation

A Document Object Model (DOM) is a representation of tree-structured data inside a computer's memory. To facilitate discussion of the DOMs used by XFA, this specification uses a particular notation to describe their contents, as defined below.

Nodes are expressed in the following form:

```
[node-type (name)]
```

where *node-type* represents the general type of the node and *name* represents the value of the `name` property of the node.

If the node has a `value` property and the value is of interest, it is shown as:

```
[node-type (name) = "node-value"]
```

where *node-value* represents the value of the `value` property.

If properties other than `name` and `value` are of interest, they are expressed in the following form:

```
[node-type (name) property-name="property-value"...]
```

where *property-name* represents the name of any one of the node's properties, and *property-value* represents the value of the corresponding property.

Indenting is used to show descent of one node from another, representing containment of the object represented by the child node within the object represented by the parent node. For example, the following shows the representation within a DOM of a subform named `Cover` enclosing a field named `FaxNo`. The field has interesting properties `value`, `w`, and `h`.

```
[subform (Cover)]
   [field (FaxNo) = "555-1212" w="1.5in" h="0.17in"]
```

"Tree Notation" on page 116 illustrates how this notation is used to describe XFA Data DOM.

Optional Terms

Within syntax definitions square brackets surround optional terms. Nesting of square brackets represents a term (inside the inner brackets) that is allowed only if another term (inside the outer brackets) is present. For example, consider the following:

```
HH[:MM[:SS[.FFF]]][z]
```

This syntax definition states that the `HH` term is mandatory. The `:MM` term is optional and does not require the presence of any other term. The `:SS` term is optional but may only be included if the `:MM` term is included. Similarly the `.FFF` term is optional but may only be included if the `:SS` term is included. Finally, the `z` term is optional and does not require the presence of any other term.

The meaning of the individual terms varies from context to context and is explained in the text where the syntax definition appears.

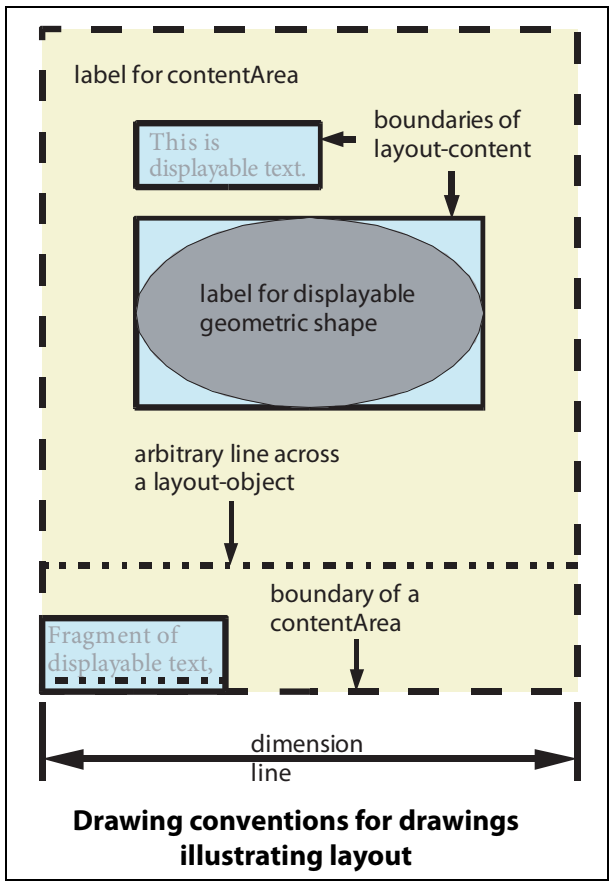
Caution: Square brackets only have this meaning in syntax definitions. When they appear inside a scripting example or an XFA-SOM expression they represent literal square-bracket characters in the script or XFA-SOM expression.

Other types of brackets or braces including "(" and ")", "{" and "}", always represent literally the character which they depict.

Graphical Conventions

Layout Drawing Conventions

Some drawings in this specification portray displayable objects, such as blocks of text, positioned upon a page. Such drawings use certain conventions which are illustrated at right. Each such drawing represents a page or a portion of a page resulting from a layout operation. Objects shown in 40% gray in the drawing would be actually visible on the page when it was rendered. Objects shown in black in the drawing give additional information that would not be visible. In addition to the color difference, visible text is shown in a serif typeface, whereas other text is shown in a san-serif typeface.



Drawing conventions for drawings illustrating layout

Object boundaries are shown with dashed or solid black rectangles. Dashed lines show the boundaries of predefined physical layout regions on the page. Solid lines show the boundaries of the nominal extent for content that is displayed upon the page. Neither of these boundaries would be visible on the page. Some objects may optionally have visible borders. The borders of an object may coincide with the boundaries of the object's nominal extent, but they are not required to. To avoid confusion borders are not shown unless relevant, and where they are shown they are in 40% gray and offset from the object's boundaries.

Some drawings show an object with a solid outline and a dot-dashed line just inside, and parallel to, the solid outline. This represents a place where a single original object has been split into two or more fragments during the layout process. Dot-dashed lines are also used for arbitrary lines that have a meaning specific to the drawing. Dimension lines and extension lines are solid.

Part 1: XFA Processing Guidelines

This part contains narrative chapters that introduce XFA and provide rules for processing XFA in conjunction with data received from an outside source. These rules provide a standard interpretation of XFA expressions and data, which helps to ensure that XFA-related behavior in XFA processing applications is equivalent. More importantly it helps to ensure that XFA processing applications that contribute to sequential processing of XFA documents do not have mis-matched expectations.

1 Introduction to XML Forms Architecture (XFA)

The XML Forms Architecture (XFA) provides a template-based grammar and a set of processing rules that allow businesses to build interactive forms. At its simplest, a template-based grammar defines fields in which a user provides data.

The open nature of XFA provides a common grammar for describing interactive forms. This common grammar provides a common basis for form-related interactions between form processing applications produced by diverse businesses.

Key Features

XFA forms provide a wide range of capabilities.

- Workflow: Data presentation, data capture and data editing, application front-end, printing.
- Dynamic interactions: From interactive, human edited forms with dynamic calculations, validations and other events to server-generated machine-filled forms.
- Dynamic layout: Forms can automatically rearrange themselves to accommodate the data supplied by a user or by an external data source, such as a database server.
- Scalability: Single-page static forms, dynamic document assemblies based on data content, large production runs containing hundreds of thousands of transactions.

XFA is similar to PDF interactive forms introduced in PDF 1.2, which is also known as AcroForm, with the following differences:

- XFA can be used in XML-based workflows.
- XFA separates data from the XFA template, which allows greater flexibility in the structure of the data supported and which allows data to be packaged separately from the form.
- XFA can specify dynamically-growing forms.
- XFA can specify Web interactions, such as HTTP and Web Services (WSDL). Such interactions can be used to submit data to a server or to request a server perform a calculation and return the result.
- XFA works with other XML grammars.

Scenarios for Using a Form Described by XFA

An XFA template describes how a form should appear and behave. It can play a role in several situations: interacting with a user, printing forms, and processing machine-generated data.

An XFA template may describe a range of form characteristics, such as the following:

- Appearance of the form, including fields, layout and graphics
- Default data to be used for fields
- Types of data expected, including checks on the validity of provided data
- Scripts associated with specific events, such as the user clicking a particular field

Interacting with a User

An XFA form interacts with a user in several ways. It presents an electronic version of an electronic form, which the user fills out. In supply data or selecting buttons, the user may trigger a variety of actions, that affect the form or that initiate an interaction with another server. The user may also invoke features that make the form more accessible.

Form Appearance

After opening a template, a user sees an interactive form that represents the layout, graphics, and fields defined in the XFA template.

The interactive form presents data associated with fields. Initially, the only data in the form are default values defined in the template. As the user provides data in fields, the default values are replaced with user-provided values. Date, time, and numeric values are displayed in a manner appropriate for the user's locale.

The user interacts with the form, supplying values and selecting options. The user's input and selections are reflected in the form.

As the user enters data, parts of the form or fields may automatically grow to accommodate data entered by the user or a machine-generated source.

Actions the User Can Trigger

XFA templates may be designed to allow a user to initiate various actions, such as the following:

- Calculations. Entering data into a field may cause the values of other fields to be recalculated.
- Data checks. Entering data into a field may initiate a series of validity checks on the entered value.
- Web Services (WSDL) interactions.
- Submission of data to a server.

Accessibility and Field Navigation

XFA templates can specify form characteristics that improve accessibility and guide the user through filling out a field.

- *Visual clues*. Fields may display default values that provide hints about the desired input values. In addition to fields, XFA template may aid the user, by providing radio buttons, check boxes, and choice lists.
- *Accelerator keys*. An XFA template may include accelerator keys that allow users to move from field to field, by typing in a control sequence in combination with a field-specific character.
- *Traversal order*. An XFA template may be defined with a traversal order, which allows the user to tab from one field to the next.
- *Speech*. An XFA template supports speech enunciation, by allowing a form to specify the order in which text descriptions associated with a field should be spoken.
- *Visual aids*. XFA template may specify text displayed when the tooltip hovers over a field or a subform.

Printing Forms

An XFA processing application can be requested to print a blank form or one that is filled out. The data for the filled-out form can come from a database, from an XML data file generated by an application, or from a previous interactive session or sessions in which data was manually entered.

During this process, the form may be printed with print view which differs from the view seen by users in interactive sessions. For example the print view might have the following differences:

- Signature fields appear as underlines for hand-signing rather than as widgets for digital signing.
- Some of the data on the form is printed as barcodes rather than text.
- Summary data is computed and printed on the first page of the form.

In addition there may be different print views for duplex (two-sided) and simplex (one-sided) printers.

Processing Machine-Generated Data

Most people think of an interactive form as something that interacts with a user, but XFA templates may specify interactions that are entirely machine oriented. For example, an XFA template may describe a machine-interactive form that consumes data produced by another machine, performs some calculations and scripts, and then submits the updated data to another machine. The sources of data could be a data base front-end, a barcode reader, or a client application submitting data.

Family of XFA Grammars

XFA is fully compliant with [XML1.0]. In keeping with the hierarchical nature of XML, XFA divides the definition of the form into a set of functional areas. Each functional area is represented by an individual XML grammar. The whole set of XFA grammars can be packaged inside a single XML document known as an XDP (XML Data Package). This is the form used to move all or part of an XFA form from one application to another.

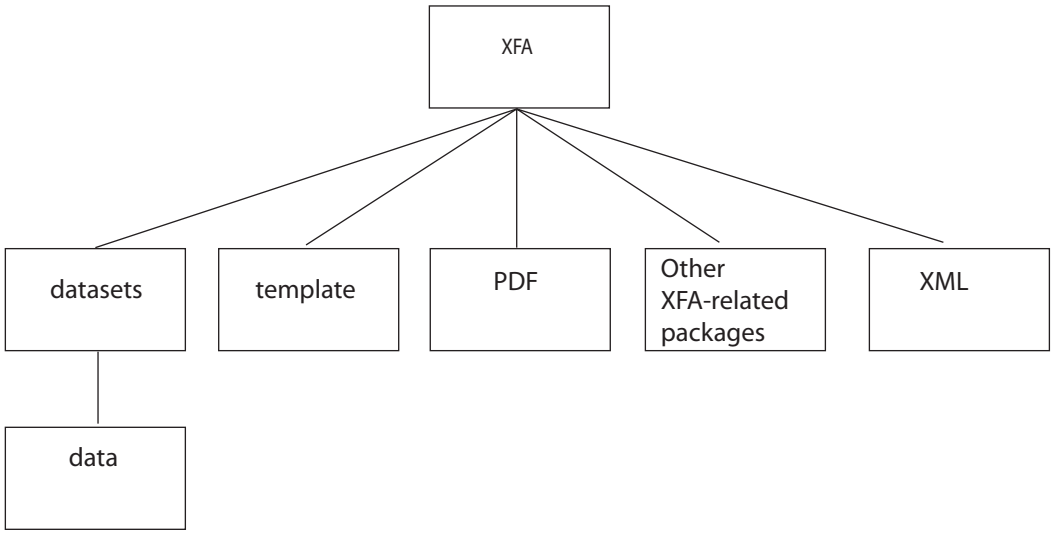
The XFA grammars are also reflected in the internal representation of XFA forms, when a form is being processed. These internal representations are Document Object Models (DOMs) of the sort with which XML-oriented programmers are familiar.

Representation of an XFA Form

The following table shows the major XFA grammars. There are some specialized grammars that are not shown here.

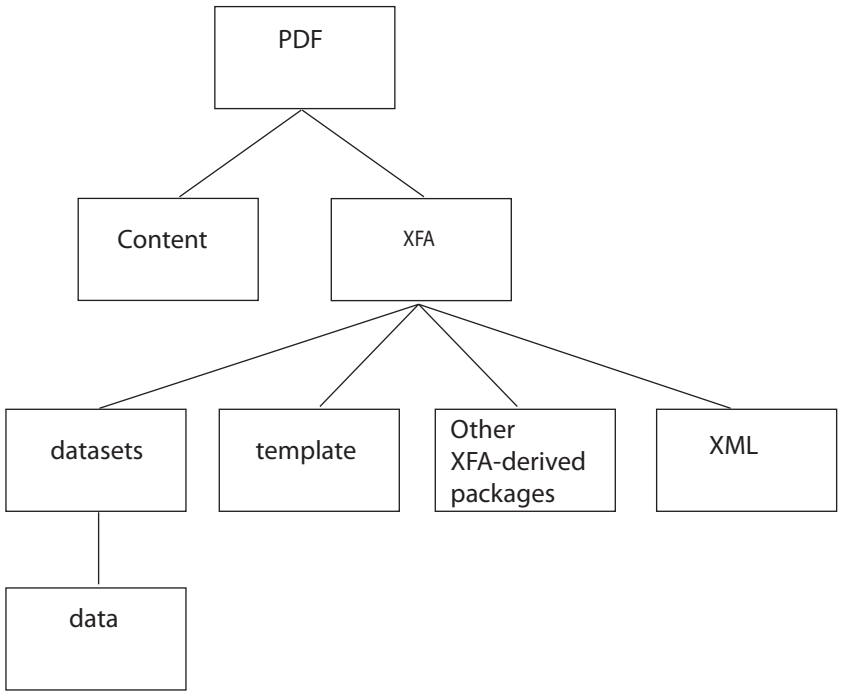
XFA grammar	Description
datasets	Contains all the sets of data used with the form.
data	Contains the data held by fields in the form. Each item of this data may have been defined as a default, entered by a user, obtained from an external file, database, or network service, or it may result from a calculation.
dataDescription	Defines the schema for the data. XFA can operate without a data schema but respects it when it is provided.

XFA grammar	Description
Other datasets	Other datasets are defined as needed for special purposes. For example when a partially-filled form is saved a special dataset may be created to hold information indicating that calculations have been manually overridden in certain fields.
template	Controls the appearance and behavior of the form.
PDF	Page background and certification information described by a PDF object. Although PDF is not an XML format, it is represented as a stream within an XML object. If such a form is displayed or printed, the template objects are drawn on top of the PDF content.
Other XFA-related grammars	The XFA grammar defines other grammars to define such information as Web connections and localization information.
Other XML	Application-defined information may be included along with the XFA-defined information. This can be expressed as any valid XML as long as the outermost element does not use an XFA namespace.



Instead of XFA containing PDF, PDF may contain XFA.

Note: When Acrobat opens such a document, it invokes the XFA plug-in, which supports XFA grammars.



The Relationship between XFA and PDF

XFA has a more abstract view of a form than PDF. XFA concentrates on the logic of a form whereas PDF concentrates on the appearance. However it is possible to combine the two. When the two are combined the result is a form in which each page of the XFA form overlays a PDF background. This architecture is sometimes referred to as XFAF (XFA Foreground).

XFAF has the advantage that the PDF can be tweaked to give fine control over the appearance of each page. However it has the disadvantage that the position of each field and of all the other page content is fixed on the page. Dynamic subforms cannot be used with XFAF. Within a page each field's content can change, or the field can be made invisible, but the field is fixed in size. This type of form corresponds to an assemblage of traditional pre-printed paper forms.

The alternative which might be called *full XFA* is to express all of the form, including boilerplate, directly in XFA. This makes it possible for the XFA processor to adjust the layout and appearance of the form as needed. For example, a list of dependants can grow to exactly the number of dependants and no more. An telephone bill can list just those charges that apply, rather than listing all possible charges with amounts of 0.00 for the ones that do not apply. A fully dynamic form of this type can be shorter when printed and can escape the busy look of pre-printed forms. On the other hand laying out a form of this type takes more CPU time. Also, XFA lacks some of the visual refinements that are available in PDF. For example in PDF characters can be individually positioned for kerning. In XFA a piece of text is a logical unit which flows wherever it must on the page, so individual positioning of characters is not possible.

Packaging an XFA Form for Application Interchange

When the family of XFA grammars used for an XFA form are moved from one application to another, they must be packaged as an XML Data Package (XDP). The XDP may be a standalone document or it may in turn be carried inside a PDF document.

XML Data Package (XDP)

XDP provides a mechanism for packaging form components within a surrounding XML container. XDP is [XML1.0] compliant and hierarchical in nature. Each of the grammars listed on page 17 is encapsulated within an element. Such subelements include the XFA template, PDF objects, XFA data and the data schema, the XFA form DOM, and custom non-XFA XML. These subelements are referred to in this specification as *packets* carried by the XDP.

It is possible for an XDP to contain all the packets of a form but it is not required. For example when a form is sent via e-mail a complete XDP is attached so that the recipient can manipulate the form using an XFA application. On the other hand when an XFA processing application submits a form's data to a server it may well send an XDP containing only the data packet.

PDF Document

An XDP may be included as an object in a PDF document, mirroring the structure illustrated on page 18. Any of the form packets may be included within the XDP.

XDP Versus PDF Packaging

Each of the packaging options, standalone XDP and XDP within PDF, has advantages and disadvantages.

Packaging form components within an XML container makes it easy for XML-based applications to produce or consume XFA forms using standard tools. In addition the XML components are human readable and can be modified with a simple text editor. Finally, when in XDP form an XFA document may be validated using the schemas which are attached to this specification. See "Schemas" on page 1085 for more information.

Packaging an XDP within PDF has the advantage that it is somewhat more compact, because PDF is compressed. Also the combination of PDF plus its enclosed XDP can be certified (digitally signed) in ways that a standalone XDP cannot.

Major Components of an XFA Form: XFA Template and Data

This section provides a high-level description of the major components of an XFA form, which are XFA template and the data provided by a user or by a server.

XFA distinguishes between template and data. The template defines presentation, calculations and interaction rules. Data is customer's application data plus whatever other data sets may be carried with the form. Though they are often packaged together, template and data are separate entities.

XFA Template

XFA template is the XFA subelement that describes the appearance and interactive characteristics of an interactive form. It was designed from the ground up to be an XML-based template language.

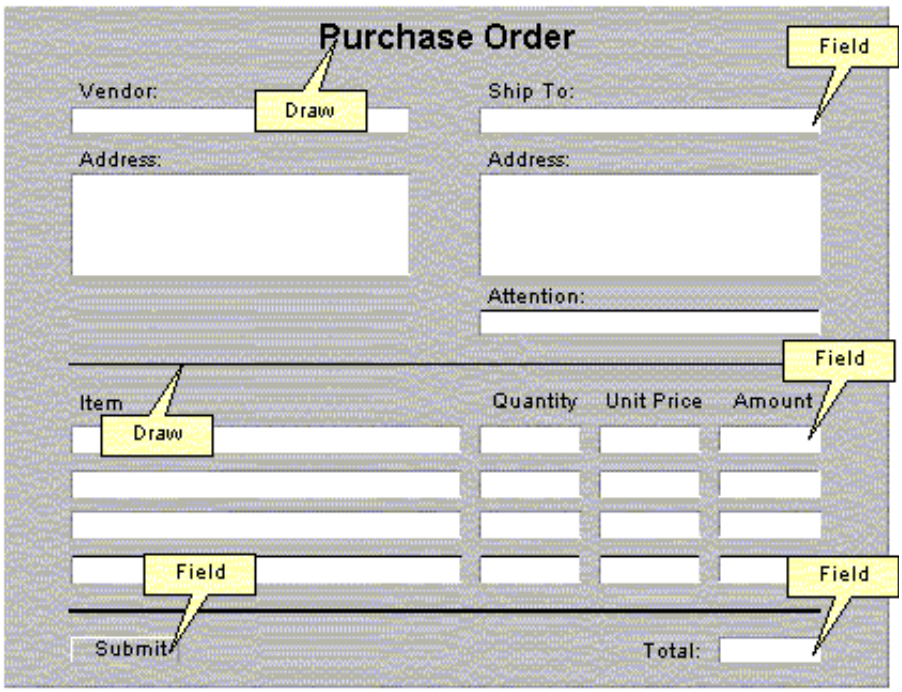
XFA follows a declarative model in which elements in an XFA template describe the components of the form. That is, an XFA template does not need to include any procedures. However scripts may be included to provide enhanced or custom functionality.

About XFA Template

Most people are consumers of forms, rather than producers or designers of forms. Yet, in order for a software product to utilize forms, someone first had to expend a degree of thought and work towards the act of creating a form. This specification is focused on the task of form creation, and it is important to distinguish between the "form" that the creator designs, and the "form" that a consumer handles — they both represent the same form, but at two very different stages in the form's life-cycle. XFA clearly distinguishes between the two stages via the following terminology:

- Form — what a person filling out a form works with, which is given life by an XFA processing application such as Acrobat.
- Template — what the form designer creates, which represents the potential for a form. A template is a collection of related subforms and processing rules.

Consider the following diagram:



A simple XFA form

This is an example of a form. To an end user (form consumer), it represents something to be filled out, by entering data into the white spaces. This user makes little or no distinction between a blank form and a filled one, other than the presence of data. In fact, the absence of data in a particular data entry element can be as meaningful as the presence of data.

In contrast, a form designer views it as a vehicle for capturing, rendering and manipulating data. As such, the designer is concerned with issues of layout, interaction and processing. A template is a specification of capture, rendering and manipulation rules that will apply to all form instances created from that template.

When selecting a form to be filled interactively, the user perceives that s/he is selecting a "blank" form. The user is performing an operation similar to starting a new document in a word processor, by first selecting a template. The user directs an XFA processing application to use this template to construct a "form", which at first appears blank. As the data is entered the association between the template and the entered data is captured in an entity known as the *Form DOM*.

Suppose the user chooses to break off filling the form before it is complete. The complete state of the form is saved by the XFA processing application as a local file containing an XDP. This state includes the template, the Form DOM, data entered so far and a record of calculations that the user has chosen to override. When the user restarts the XFA processing application it reloads the complete form state and the user is able to resume just where he or she left off.

When the user has finished filling out the form he or she clicks on a submit button. The submit button is defined within the template and its characteristics were determined by the form creator. The submit button may cause the data to be submitted to a server on the web via HTTP or SOAP, or to a local database. Whichever of these submission methods is used, usually only the user data is submitted. However the form creator may direct the application to submit the the whole form or some subset of it.

Containers of Fixed and Variable Content

XFA template distinguishes between containers for fixed content and containers for variable content.

The draw element, a container for fixed content

Fixed content (*boilerplate*) includes any text, lines, rectangles, arcs, or images that remain unchanged throughout the life of the form, except in regards to its container placement, container size, and container inclusion/omission. Fixed data is defined by the template designer as the contents of `draw` elements. "A simple XFA form"includes callouts that indicate a few of the many `draw` elements on this form.

An XFA template does not have to include any fixed content. The fixed content can be expressed as PDF instead, using the XFA Foreground (XFAF) architecture. However if this is done the presence of the fixed content, which page it is on and where it is upon the page are also fixed. By contrast when the fixed content is expressed as `draw` elements (a non-XFAF form) it can be dynamically omitted or moved within the form as required. For more information see "The Relationship between XFA and PDF" on page 19.

The field element, a container for variable data

Variable data can change during the life of the form and is provided by any of the following: the template as a default value, the person filling out the form, an external source such as a data base server, a calculation, and other sources. The template can specify default data for fields. The other sources of data come into play when an XFA processing application represents the template and associated grammars as an interactive form.

Variable data is described in field elements. "A simple XFA form"includes callouts that indicate a few of the many `field` elements on this form. Note, the submit-button in the lower-left corner is also a field.

Containers of Other Containers

Template containers provide the structure for the form, by collecting other containers into repeatable sets. Containers of other containers include the following:

- exclusion group: Container of multiple fields. Exclusion groups represent a radio-button grouping.
- subform: Container of other containers. The subforms that appear in a form may be pre-determined (as in a static form) or may expand to accommodate the data bound to it (as in a dynamic form).
- page area or area: An abstract representation of the physical page or other area in which other containers are placed. The area does not change in size, but if its capacity is exceeded, a new area is created.

Laying Out the Containers (and Their Data) to Create the Form's Appearance

Each subform and area is a little document in and of itself. Subforms are assembled together in order to create the final document. Subforms also support repeating, optional and conditional data groups. These allow the construction of a form which responds to structure in the bound data. For example the form can automatically add more pages to accomodate all of the supplied data records. When the XFA processing application creates the interactive form, it reconciles the template and the data by adding enough copies of the appropriate subform to accommodate the data.

Allowing the data to drive the number of subforms used in a form has several advantages. It is less error-prone than predefining multiple instances of the subform, and the template designer need not guess at a maximum possible number to handle all anticipated cases. In addition, because XFA Template is a declarative language, there is no need to write script to create such instances when the content is bound.

An important feature of XFA is that a template can stand alone. It doesn't need data to bring it to life. A template without data is the equivalent of a blank form, ready for filling.

Scripted Components of an XFA Template

It is important to understand that scripting is optional. The template designer can take advantage of scripting to provide a richer user experience, but all of the features described so far operate without the use of scripts. Script creation is part of the template designing process.

XFA supports scripting in ECMAScript [ECMAScript], but it also defines its own script language, FormCalc ["FormCalc Specification" on page 891]. Often, the scripts attached to a form are similar to those attached to a spread-sheet. FormCalc has been designed as an expression-oriented language, with simple syntax for dealing with groups of values in aggregate operations.

Both ECMAScript and FormCalc expose the same object model. Scripting almost always works with data values, so these are easily referenced. Indeed, XFA defines a complete Scripting Object Model ("Scripting Object Model" on page 73). A key feature of XFA-SOM is that it manages relative references. For example, when defining an invoice detail line, a template designer might set up fields `unitPrice`, `quantity` and `amount`. The calculation for `amount` would simply be `unitPrice*quantity`.

XFA-SOM would resolve the references by finding the correct instances of `unitPrice` and `quantity` in the following situations:

- If the instances of those field names are in other subforms
- When there are multiple instances of those field names in the same subform

Because of the declarative nature of XFA Template, the largest use of scripting is for field calculations. A field with such a script typically is protected against data entry, and instead gets its value from an expression involving other fields. A field's calculation automatically fires whenever any field on which it depends changes (those fields may, in turn, also have calculated values dependent on other fields, and so on).

Similar to calculation, a field can have a validation script applied that validates the field's value, possibly against built-in rules, other field values or database look-ups. Validations typically fire before significant user-initiated events, such as saving the data.

Finally, scripts can be assigned to user actions, for example, such as when the user enters data and when the user clicks on a field. Scripts can also be activated by events that occur behind the scenes, such as assembling data to send to a web service.

Data

Typically, XFA variable content is the customer's XML data, matching the customer's schema. Data could also come from a database, an HTTP POST response, a web service interaction, default data supplied by the template or other source. Often, form data elements are plain text, but may also include rich text and graphics.

XFA defines a data value to be an XFA name/value pair, where the value is plain or rich text, or a graphic. Data values may contain nested data values. An XFA name is a string suitable for identifying an object. A valid XFA name must be a valid XML name, as defined in [XML], with the additional restriction that it must not contain a colon (:) character.

XFA also defines a data group: the provider of structure in the data. Data groups may contain data values and other data groups. As stated above, the data is typically structured according to the customer's schema; data values and data groups are represented as abstract structures, inferred from the customer's data. The abstract structure helps the XFA processing application create an XFA form that reflects the structure and content of the data. This process (called data binding) is described in the next section.

It is important to note that XFA doesn't have to treat the data as a read-only source of variable content. Many forms-based workflows involve round-tripping: load the data into the template, edit or augment it, and save out the modified data. XFA can be instructed to remain true to the data's original structure when saving. When data values and groups are logically moved to match the structure of the template, the form designer has an option as to whether saving the data will or will not reflect those moves.

While data is often authored via legacy applications or database queries, it can also be authored through an interactive form filling applications, such as Acrobat version 6 and greater.

Data Binding: Making the Connection Between XFA Template and Data

When an XFA processing application introduces data to an XFA form, it associates each piece of information to a container, such as a field or a subform. This process is called *data binding*.

Generally, XFA data binding attempts to map like-named data values to template fields and data groups to template subforms. Data and template structures often don't match. XFA processing defines default binding rules to handle such mismatches. Alternatively, the template designer may choose to provide data binding rules in the template definition. If those alternative do not provide desired results, the template designer may change the structure and/or content of the XML data, by specifying XSLT scripts the XFA processing application uses to pre-process and post-process data. See also "Basic Data Binding to Produce the XFA Form DOM" on page 155 and "Dealing with Data in Different XML Formats" on page 423.

Data binding alters neither the template nor the data. That is, data binding is internal to the XFA processing application.

Unbound data values and groups (those that don't have matches in the template structure) are preserved and won't be lost or moved if the data is saved.

The binding operation can create forms with repeated subforms, in which multiple run-time instances of the subform are created to accommodate the multiple instances present in the data. A form with such a capability is called a *dynamic* form. A form without such a capability (that is, with the capability switched off) is called a *static* form. When using the XFAF architecture, in which fixed content is expressed as PDF, the form must be static.

In addition, XFA data binding is designed to handle like-named data values, as well as like-named data groups. It is common in forms processing for there to be multiple data values present with the same name. For example, invoice data generated from a database query could easily have multiple item, description, unit price and quantity fields in a relatively flat structure. In addition, there may be repeating data groups. XFA defines default binding rules to ensure that these map intuitively to like-named fields or subforms in the template. The basic rules for dealing with multiple instances of the same name are:

- The relative order of sibling items that have different names is not important, may be ignored and does not need to be maintained; and
- The relative order of sibling items that have the same name is important, must be respected and maintained

For more information, see "Basic Data Binding to Produce the XFA Form DOM" on page 155.

Lifecycle of an XFA Form

This section describes the common steps in the lifecycle of an interactive form based on XFA.

Creating an XFA Template

There are several ways a form designer can create a template:

- Using a graphical layout tool, such as Adobe LiveCycle® Designer
- Automatically by software, a capability provided by SAP® Smart Forms conversion
- Hand-edit XFA Template files

In a template designing application with a graphic interface, a template designer can start with a blank canvas and place objects, or the author can start with a schema, for example, XML-Schema [XML-Schema], data source or data file. When starting with a schema, the designer can select portions or all of the schema tree and place them on the template canvas, and the design tool will create subform/field structure, with properly-typed fields, template-defined bindings. layout constraints, and processing rules.

Filling Out an XFA Form

Opening a Form

When a user directs an XFA processing application to open an XFA form, the application goes through the following general steps:

1. Draw the initial form. The XFA processing application uses the XFA template to determine the initial form appearance and to obtain default data. If the application supports only form printing or machine-provided data, this task is omitted.

2. Associate data with specific fields. If data is included in the XFA form, the application creates logical connections between the data and the fields.

3. Display data in fields. The application displays the data associated with each field. Time, date, or number data may be formatted. The data may result in the form appearance changing, as fields grow or subforms are added to accommodate the data.

4. Activate events. The application activates events associated with creation of the form. Such events can include interacting with a server to obtain data used in the form.

5. Update form. The XFA processing application executes calculations and data validations for any fields whose values have changed.

The above description is a simplification. Much more goes on behind the scenes, as will be described in later chapters. In addition, the actions described above may take place in different sequences.

Providing Data to the Form

The user provides data by bringing a field into focus and then entering data. A field can be brought into focus by using a mouse to select the field or by entering key(board) sequences. The data is processed as described in Step 5. on page 26.

Saving an In-Progress Version of the Form

XFA processing applications will typically allow users to save an in-progress or committed version of a form. (See "Committing a Form" on page 26.)The method for requesting the save is application-specific. Typically the form is serialized in XDP format with the data, the Form DOM, and other state information included in the XDP. State information commonly includes a list of all the fields in which the user has overridden calculations.

Committing a Form

After completing a form, the user is ready to submit the form. That is, the user has supplied all information required by the form and repaired any errors reported by the form. The form is ready to be released for further processing.

XFA forms may use digital signatures. The user typically clicks a button to sign the form. The digital signature may include only data, in which case it provides non-repudiable proof that the user did enter that data. Alternately the signature may be applied to other parts of the document to prove that they have not been tampered with since the signature was applied.

Typically, the template supplies the form with a submit button. The button has properties which control the submit operation. In addition the button may specify a validation test and/or a script that must be run before the data is submitted. Typically, if the validation or script fails, the user is asked to make corrections and resubmit the form.

Once the data has passed all required validation it is submitted across the network. When the processing application has successfully submitted the form content, the form is said to be *committed*. That is, the form has entered the next stage of processing. For example when a customer of an online store commits the purchase form, the store then charges the purchase to the customer's credit card and ships the merchandise.

XFA also supports sending the filled form as an e-mail attachment or via physical conveyance of a printed copy. These methods do not require a submit button. Typically the filled form can be saved as a file or sent to a printer via the user interface of the XFA processing application.

Processing Form Data, a Server Application Task

XFA processing applications may establish themselves as servers designed for machine-based XFA form processing. See "Processing Machine-Generated Data" on page 17. In one scenario, such an application

might accept data submitted by a client. Such submission is described in "Committing a Form" on page 26. The server application would accept the data and template and create its own interactive form, as described in "Opening a Form" on page 25; however, it would not display any of the form content.

The server application may perform various tasks, such as performing additional calculations, incorporating other data into the form, and submitting the data to a database.

Static versus Dynamic Forms

XFA distinguishes between static and dynamic forms.

In a static form the form's appearance and layout is fixed, apart from the field content. When the template is bound to data (merged), some fields are filled in. Any fields left unfilled are present in the form but empty (or optionally given default data). These types of forms are uncomplicated and easy to design, and suffice for many applications.

In XFA 2.5 a new and simpler mechanism is introduced for static forms. This mechanism uses PDF to represent the appearance of the boilerplate and employs XFA grammar only to represent the form logic. This mechanism is known as XFA Foreground (XFAF). Such forms are easier to construct because they use a subset of the XFA grammar. At the same time the use of PDF for the boilerplate makes it possible to refine the appearance of the boilerplate to a high degree. For example it is possible to accurately reproduce the kerning of an existing preprinted paper form.

Dynamic forms change in appearance in response to changes in the data. They can do this in several ways. Forms may be designed to change structure to accommodate changes in the structure of the data supplied to the form. For example, a page of the form may be omitted if there is no data for it. Alternatively changes can occur at a lower level. For example, a field may occupy a variable amount of space on the page, resizing itself to efficiently hold its content. Dynamic forms use paper or screen real estate efficiently and present a cleaner, more modern look than static forms. On the other hand a dynamic form cannot rely on a PDF representation of its boilerplate, because the positioning and layout of the boilerplate change as the fields grow and shrink or as subforms are omitted and included. Hence dynamic forms require that the boilerplate be included in the template in a form that the XFA processor can render at run-time. This rendering does not have the exquisite control over appearance that PDF yields, for example it does not support kerning. Furthermore it uses more CPU time than a static XFAF form.

The next few chapters discuss matters that apply to all XFA forms whether they are static or dynamic.

"Template Features for Designing Static Forms" on page 29 discusses the subset of template features used by XFAF forms. These core features are also fundamental to dynamic forms.

"Object Models in XFA" on page 62 discusses the ways in which the template, data, and other objects are represented inside an XFA processor and the ways in which they interact.

"Exchanging Data Between an External Application and a Basic XFA Form" on page 108 discusses the way in which data in an external document is parsed and loaded into a static form and how it is written out.

"Representing and Processing Rich Text" on page 188 describes the syntax used for rich text.

Often forms have repeated sections or repeated components, for example the same logo may appear on multiple pages of a form. In addition different forms may have sections or components in common. "Template Features for Designing Forms with Repeating Sections" on page 195 describes template features which can be used to reuse declarations within a form or across forms.

The next set of chapters discuss matters that apply only to dynamic forms.

"Template Features for Designing Dynamic Forms" on page 217 describes additional template features needed by dynamic forms.

Forms may be designed with containers that stretch or shrink to accomodate varying amounts of data within individual fields. Containers of this sort are called growable containers. "Layout for Growable Objects" on page 237explains how the content of forms with a fixed data structure but with growable containers is laid out.

Forms may also adjust themselves to data which varies in structure. "Dynamic Forms" on page 286 describes the operation of data-driven forms in which subforms are included or excluded depending upon the data.

"Layout for Dynamic Forms" on page 310 describes the differences between layout for forms with growable containers but fixed data structure and layout for forms with variable data structure.

The remaining chapters in Part 1discuss specialized features of XFA. These can be used in both static and dynamic forms.

"Automation Objects" on page 322 describes the ways in which calculations, validations, and a variety of events are linked into other objects on the form and how they interact.

"Scripting" on page 353 describes the scripting facilites.

"Forms That Initiate Interactions with Servers" on page 375 describes the ways in which forms can interact with servers across the network via either HTTP or WSDL/SOAP.

"User Experience" on page 406 describes the assumptions that XFA makes about the functioning of the user interface and documents the facilities for enhanced accessibility.

XFA forms can deal directly with almost all XML data, but occasionally it is desireable for the form to see a reorganized view of the data. "Dealing with Data in Different XML Formats" on page 423 describes the facilites built into XFA for reorganizing data while loading it, and in most cases transforming it back to the original organization when it is written out.

"Security and Reliability" on page 460 deals with issues of authentication, trust, and non-repudiability as well as protection against attack.

2 Template Features for Designing Static Forms

This chapter describes the template features used to create static forms. Such forms have a fixed appearance, maintaining the same layout, regardless of the data entered in them.

Starting with XFA 2.5 a simplified format for static forms is introduced. Using this format the static appearance of each page is expressed as PDF. Only the logic of the form, the fields which contain user data and the subforms that relate the fields, are expressed in XFA. This format is called XFA Foreground or XFAF.

This chapter provides a basic description of the forms represented by XFA templates. It is intended for use by form designers and others who do not need to understand the more detailed processing guidelines of XFA forms. Accordingly, this chapter uses the terms *elements* and *attributes* to describe template entities, as does the "Template Specification" on page 482. Subsequent chapters in Part 1: XFA Processing Guidelines use the terms *objects* and *properties* to describe such entities. This shift in terminology reflects a transition to describing processing guidelines relative to XML Document Object Model (DOM) representations of an XFA form.

Form Structural Building Blocks

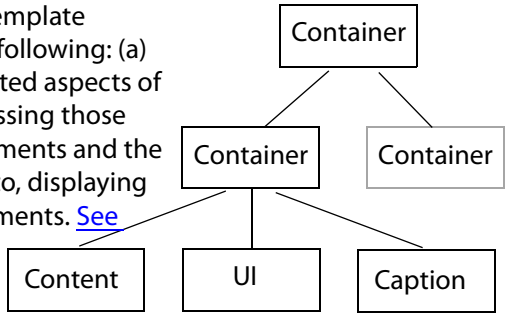
This section describes the most significant XFA elements and their relationships. Such elements fall in the groups: "Container Elements", "Content Elements", and "User Interface". These groups are defined in the XFA template grammar.

A form is presented to the user as a combination of fixed background text and images (also known as *boilerplate*), as well as the variable content present in the fields and typically provided by a user. The end user, while very aware of a form's interactive content, generally doesn't make a distinction between that content and the field that houses it. However, the distinction is important to the form designer. Content elements tend to be concerned with data-related issues, such as data type and limitations. In contrast, container elements are concerned with presentation and interaction.

The following figure illustrates the relationship between container elements, content elements, and UI elements.

Types of structural building blocks in an XFAF template

Container element is a type of XFA template element that specifies either of the following: (a) content elements and the form-related aspects of navigating to, displaying and processing those elements; or (b) other container elements and the form-related aspects of navigating to, displaying and processing those container elements. See "Container Elements" on page 30.



Content element is a type of XFA template element that houses datatyped pcdata (text) or images. Such pcdata or graphic elements may be defined as default data in the content element. See "Content Elements" on page 32.

UI element is a type of XFA template element that describes how data should be presented to a form user. See "User Interface" on page 33.

Caption element provides static text that labels a container.

Note: This specification allows for captions on many different containers, but the Acrobat 8.0 implementation of XFAF only supports the visual presentation of captions on button and barcode fields. However captions can still be used to supply text for an `assist` element. Also, this limitation does not apply to dynamic forms or old-style non-XFAF static forms.

Container Elements

The term *container element* refers to an element that houses content and the form-related aspects of dealing with its content, such as the following:

- Variable content or the potential for variable content. Variable content includes text and images.
- Caption for the container.
- Formatting and appearance such as a border around the container, text formatting and localization, and barcode formatting.
- Accessibility, such as traversal order between containers and speech order for the text associated with a container.
- User interaction, as described in "User Interface".
- Calculations that consider the content of this and other containers.
- Validations used to qualify data associated with the content element.
- Other interactions, such as form or keyboard events and web service interactions.

Containers Associated with Variable Content

Field

A `field` element is the workhorse of a template and represents a data-entry region. A user typically interacts with `field` elements by providing data input. Fields provide a pluggable user interface ("User

Interface" on page 33) and support for a broad variety of content data-typing ("Content Types" on page 36).

The following is an example of a `field` element that produces a data-entry region capable of accepting textual input. The field is positioned at an (x,y) coordinate of (0,0) and has a width of 1 inch and a height of 12 points.

Example 2.1 A field accepting data input

```
<field name="ModelNo" x="0" y="0" w="1in" h="12pt"/>
```

For more information, please see the syntax description of the `field` element.

Exclusion Group

An exclusion group is a non-geographical grouping of fields, where one of the fields provides the value for the exclusion group.

The fields in an exclusion group exhibit mutual exclusivity commonly associated within radio-buttons or ballot/check-boxes, as shown at right. Only one of the objects may have a value or be selected by the user. The value of an exclusion group is the value of the selected or "on" field. (Example 2.2)

The fields contained in exclusion groups should be restricted to those containing checkButton widgets. The behavior of exclusion groups containing other types of fields is undefined.

Exclusion groups may be defined with or without a default value. The default value for an exclusion group is the default value provided by one of the fields in the group. An error exists if more than one field within an exclusion group provides a default value.

Example 2.2 Check-button exclusion group with a default value

```
<exclGroup … >
   <field … >
      <ui>
         <checkButton shape="round" … />
      </ui>
      <items>
         <integer>1</integer>
      </items>
      <value>
         <text>1</text>
      </value>
   </field>
   <field … >
   <ui>
      <checkButton shape="round" … />
   </ui>
   <items>
      <integer>2</integer>
   </items>
   </field>
</exclGroup>
```

For more information, please see the syntax description of the `exclGroup` element.

Containers That Group Other Container Elements

Template

A `template` is a non-geographical grouping of subforms. The template represents the form design as a whole, enclosing all of the elements and intelligence of the form.

The following is an example of a `template` element that describes a form comprised of two text fields:

Example 2.3 Template element enclosing a simple form

```
<template xmlns="http://www.xfa.org/schema/xfa-template/2.5/">
   <subform name="Device" x="2" y="3">
      <field name="ModelNo" x="0" y="0" w="1in" h="12pt"/>
      <field name="SerialNo" x="0" y="16pt" w="1in" h="12pt"/>
   </subform>
</template>
```

Subform

Common paper forms often contain sections and subsections that are easily distinguished from one another. For example, there are three distinct sections for header, detail and summary information in the diagram "A simple XFA form" on page 21. The form is really a collection of these sections and subsections, each of which XFA refers to as a subform. Some of the features offered by `subform` elements include:

- Management of scope of element names in scripting operations, as described in "Scripting Object Model" on page 73
- Validation of the content of the subform as a whole, as described in "Validations" on page 329
- Hierarchical data binding, as described in "Basic Data Binding to Produce the XFA Form DOM" on page 155

The `subform` element provides the level of granularity that a form object library would use. A form object library is a tool used by form designers to store commonly used groupings of form container objects, for example, company letterhead.

The following is an example of a `subform` element that encloses two text fields:

Example 2.4 Subform element enclosing two fields

```
<subform name="Device">
   <field name="ModelNo" x="0" y="0" w="1in" h="12pt"/>
   <field name="SerialNo" x="0" y="16pt" w="1in" h="12pt"/>
</subform>
```

For more information, please see the syntax description of the `subform` element.

Exclusion Group

See "Exclusion Group" on page 31.

Content Elements

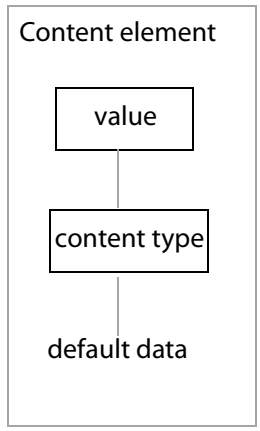
A content element is a type of XFA template element that houses datatyped pcdata (text) or images. Such pcdata or graphic elements may be defined as default data in the content element.

Most containers have a notion of a value. This value can be used in calculations and may be persisted when the form's variable content is saved. For `field` and `exclGroup` containers, the value is the container's content, available through the `value` subelement.

The following diagram illustrates the relationship between elements in a content element.

A value element represent references a datatype element and specifies whether the data may be overridden by the form user or by some other source.

A content type element defines the type of the data. It may also include default data, which is used when the form is displayed. Examples of such datatype elements are `date`, `decimal`, and `image`. Content types are later described in "Content Types" on page 36.



Structure of a content element, such as a field or exclGroup element

User Interface

The XFA architecture makes a clear distinction between the content of a container and the user interface (UI) required to render that content and provide interaction. While there often is a relationship between content and UI (e.g., date content would normally be captured with a date-oriented UI), the separation allows both the application and the form designer some degree of flexibility in choosing the right UI. This separation allows the form designer to exert some control over the user interface, selecting the widget most appropriate for each instance of a given type of content.

Each container may have a ui subelement for specifying user interface for the container. That element, in turn, may contain an optional child element, specifying a possible user interface for the container. If the UI element contains no children or is not present, the application chooses a default user interface for the container, based on the type of the container's content.

The chapter "User Experience" on page 406 provides more information on the user interface described by XFA templates.

Basic Composition

This section describes the basic aspects of creating a template. Such issues include measurements and positioning graphic elements within a parent container. "Basic Layout" on page 45 describes how container elements are placed on a page.

Measurements

All measurements are comprised of two components:

- The quantity or value of the measurement
- The (optional) unit of the measurement

The following is an example of fields containing different font elements with the equivalent field and font size expressed in a variety of different measurements.

Example 2.5 Equivalent measurements in field and font size declarations

```
<field name="f1" y="1in" h="72pt" w="4in">
   <font typeface="Helvetica" size="72pt"/>
</field>
<field name="f2" y="2in" h="1in" w="4in">
   <font typeface="Helvetica" size="1in"/>
</field>
<field name="f3" y="3in" h="1" w="4in">
   <font typeface="Helvetica" size="1"/>
</field>
```

Values

All measurements have a quantity or value, which is expressed in a particular unit that may either be explicitly stated or implied. Common uses of measurements include the description of a length or width of an element, the position of an element, or an offset from a coordinate.

The format of a measurement is a value, consisting of the following parts:

- An optional sign character — one of "+" (the default) or "-"
- A number — a number with optional fractional digits
- An optional unit identifier

The following are examples of measurement and their interpretations:

- 1in — one inch
- -5.5cm — minus five and a half centimeters
- 30pt — thirty points
- 0 — a measurement of zero with the unit omitted

Units

All measurements are expressed in a particular unit which may be specified as a suffix to the value. The unit is known by a short textual identifier, such as "in" for inches. The default unit is assumed to be inches. In other words, the following are equivalent:

- 3.5in
- 3.5

The following list is the set of allowable units and the corresponding identifiers:

- cm — centimeters
- in — inches (This specification considers one inch to be exactly 2.54 centimeters.)
- mm — millimeters
- pt — points (This specification considers a point to be exactly 1/72 of an inch.)

Note that a unit specification is not required or implied when the measurement value is zero. Not all elements may support all possible types of units, as described in "Restrictions" (below).

Angles

Certain measurements requires the specification of an angle. Angles are always specified in degrees and are measured counterclockwise from a line parallel to the X axis.

Restrictions

Individual elements may place restrictions on measurements; in these cases the corresponding specification of the element will clearly describe the restrictions — if no restriction is noted, then the element does not exhibit any restrictions on measurements.

For instance, the specification for an element may:

- Restrict the use of the sign character, limiting the measurement to either a positive or negative value
- Restrict the value, limiting the measurement to whole numbers

Border Formatting

A UI element may describe formatting characteristics for a border around the widget.

Borders

A border is a rectangle around a widget. Independent control over the appearance of sides and corners of the rectangle is provided.

The border has its own margins, independent of the widget's margins. It is possible for the widget to overlap the border. However the widget always draws on top of the border.

A border is comprised of one or more optional:

- Edges — the sides of the box, described by `edge` elements
- Corners — the intersections of the edges, described by `corner` elements

The border is rendered starting at the top-left corner and proceeding clockwise, using up edge and corner elements in the order of rendering. Thus the first `edge` element represents the top edge, the next represents the right edge, then the bottom, then the left. Similarly the `corner` elements are in the order top-left, top-right, bottom-right and then bottom-left.

The border can also include a fill specification, indicating that the area enclosed by the border is to be filled with some sort of pattern or other shading. As with the border itself, the widget always draws on top of the fill.

Thickness

Borders have outlines that are rendered according to one or more pen-strokes. The `edge` and `corner` elements within a `border` represent pen-strokes.

Each of these elements possesses an attribute which determines the thickness of the stroke, and as the thickness increases the stroke appears to become wider and spread further toward the inside of the border. This growth toward the inside ensures that the graphical representation of the border fits within the nominal extent of the border.

Fill

The fill element indicates how the region enclosed by the border is to be filled. Types of fill include:

- None
- Solid
- Hatching and crosshatching
- Stippling of two colors
- Gradient fills:
  - Linear
  - Radial

The fill element has a child color element. One can think of this as specifying the background color for the fill. The fill element also has a child element specifying the type of fill, such as solid, pattern, and stipple. This child, in turn, has its own child color element. This second color can be thought of as the foreground color. For example, the following would create a fill of horizontal black lines on a gray background.

Example 2.6 Fill with horizontal black lines on a gray background

```
<fill>
   <color value="128,128,128"/>
   <pattern type="horizontal">
      <color value="0,0,0"/>
   </pattern>
</fill>
```

Note: If the fill element is omitted or empty the result is a solid white fill.

Content Types

Within a template default data is represented within value elements, as shown in the following examples. The content types are identified in bold.

Example 2.7 Placeholder for user-provided data (provided during data binding)

```
<field y="10mm" x="10mm" w="40mm" h="10mm">
   <value>
      <text/>
   </value>
</field>
```

Example 2.8 Default data, which may be replaced by user-provided data

```
<field y="10mm" x="10mm" w="40mm" h="10mm">
   <value>
      <text>Hello, world.</text>
   </value>
</field>
```

Text

The text content type element enclose text data, as shown in Example 2.8. Text is any sequence of Unicode characters. Alphabetic text includes any Unicode character classified as a letter in the Basic Multilingual Plane (BMP), described in [Unicode-3.2]. An alphanumeric character is any Unicode character classified as either a letter or digit in the BMP.

Date, Time, and DateTime

The date, time, and dateTime elements enclose text content that complies with the corresponding canonical patterns specified in "Canonical Format Reference" on page 887. For example, the canonical forms for the date content type follow:

```
YYYY[MM[DD]]
YYYY[-MM[-DD]]
```

In most cases, people filling out a form will supply date, time and dateTime data in a format specific for the locale. As described in "Localization and Canonicalization" on page 138, such localized forms are converted to a canonical format before being represented as data in the form. Similarly, canonical data is converted into a localized form before being present to the user.

Boolean, Integer, Decimal, and Float

The boolean, decimal, float, and integer content type elements may enclose numbers that comply with the corresponding canonical number forms, as specified in "Canonical Format Reference" on page 887.

In most cases, people filling out a form will supply numeric data in a form specific for the locale and using local-specific symbols. As described in "Localization and Canonicalization" on page 138, such localized numeric data is converted to a canonical format before being represented as data in the form. Similarly, canonical data is converted into a localized form before being present to the user.

<table>
<tr><th>Numeric content type</th><th>Description</th></tr>
<tr><td>boolean</td><td>The content of a boolean element must be one of the following:
<table>
<tr><th>Digit</th><th>Meaning</th></tr>
<tr><td>0 (U+0030)</td><td>Logical value of false</td></tr>
<tr><td>1 (U+0031)</td><td>Logical value of true</td></tr>
</table></td></tr>
<tr><td>integer</td><td>An optional leading sign (either a plus or minus, Unicode character U+002B or U+002D respectively), followed by a sequence of decimal digits (Unicode characters U+0030 - U+0039). There is no support for the expression of an exponent. Examples of canonical integer content are shown below:

12
-3</td></tr>
</table>

Numeric content type	Description

decimal	A sequence of decimal digits (Unicode characters U+0030 - U+0039) separated by a single period (Unicode character U+002E) as a decimal indicator. Examples of canonical decimal content are shown below:
	`+12.`
	`1.234`
	`.12`
	`-123.1`
	In decimal content types, the number of leading digits and fractional digits may be retained.
	Note: If a person filling out a form supplies non-conforming data for a decimal content type, the XFA processing application may chose to automatically convert the data into the decimal equivalent.

float	An optional leading sign (either a plus or minus, Unicode character U+002B or U+002D respectively), followed by a sequence of decimal digits (Unicode characters U+0030 - U+0039) separated by a single period (Unicode character U+002E) as a decimal indicator. Examples of canonical float content are shown below:
	`1.33E-4`
	`.4E3`
	`3e4`
	Note: If a person filling out a form supplies non-conforming data as the value of a decimal content type, the XFA processing application may chose to automatically convert the data into the float equivalent.

Absent Content

When no content is present, the content shall be interpreted as representing a null value, irrespective of properties that specify null characteristics (`bind.nullType`).

Decimal Point (Radix Separator)

Data associated with decimal and float content types must include a decimal point. To maximize the potential for data interchange, the decimal point is defined as '.' (Unicode character U+002E). No thousands/grouping separator, or other formatting characters, are permitted in the data.

Images

The `image` content type element may enclose an image. XFA fields may accept images as data from a user or from an external source.

Note: The image formats supported by an XFA processing is application dependent.

Images Provided as Data

The template may provide an image as a default value for the field (Example 2.9) and an image may be supplied by a user (Example 2.9) or by an external source (Example 2.10), such as a database server.

A field image is either embedded in the image element as PCDATA or is referenced with a URI.

The user interface provides an imageEdit widget that allows images to be supplied as URIs. The imageEdit properties specify whether the image is embedded in the image element or represented as a reference.

Example 2.9 Field with an editable image value

```
<field name="ImageField1" w="80.44mm" h="28.84mm">
   <ui>
      <imageEdit data="embed"/>
   </ui>
   <value>
      <image contentType="image/jpg">/9j/4AAQSkZJRgABAQEASA…
…
wcUUUVzm5//Z</image>
   </value>
</field>
```

Example 2.10 Field with an image value supplied by an external source

```
<field name="signature"
   y="58.42mm" x="149.86mm" w="44.45mm" h="12.7mm" anchorType="topCenter">
   <value>
      <image aspect="actual"
         href="http://internal.example.com/images/signatures/JSmith.jpg">
   </value>
   <bind match="global"/>
</field>
```

"Image Data" on page 132 describes processing guidelines for including in the XFA Data DOM image href references that appear in data.

Scaling the Image to Fit the Container (aspect)

Image elements contain a property (`aspect`) that specifies how the image should be adjusted to fit within the dimension of the container. See "Images" on page 47.

External Data

The `exData` content type may enclose foreign data, which is PCDATA that represents the actual data content of the specified content type. The actual data is encoded as specified by `exData`.

When no data content is provided, the data content may be interpreted as representing a null value. This behavior is dependent upon the context of where the data content is used. For instance, a field may interpret empty data content as null based upon its `bind.nullType` property.

Formatting Text That Appears as Fixed or Variable Content

Text may appear in many places in an XFA form. It may appear as fixed content, such as that in a caption, or it may appear as variable content, such as that supplied by the user or some other source. This section explains how such text can be formatted.

Variable content text may be formatted as specified by text patterns known as *picture clauses*. For example when displaying a monetary amount to the user a picture clause can insert a currency symbol, thousands

separators and a decimal (radix) indicator, all appropriate to the locale. On input the picture clause strips away this extraneous information so that internally the form deals with pure numbers in a canonical format. Fixed content text such as caption text is not subject to any picture clause.

Next fixed or variable content text is formatted based on the containing element's paragraph-formatting and font-formatting properties, which may themselves be inherited from an ancestor container. If the text content is derived from rich text, the formatting contained in the rich text over-rides formatting described by the containing element.

Finally, the resulting data may be further formatted and displayed as a barcode using various barcode styles.

Layout Strategies

There are two layout strategies for a layout container, *positioned layout* and *flowing layout*. In positioned layout, each layout object is placed at a fixed position relative to its container. In flowing layout, objects are placed one after the other in sequence, and the position of each object is determined by all the previous objects to be placed in the same container.

Most layout containers can use either positioned or flowing layout. However blocks of text always use flowing layout internally. The words and embedded objects within a block of text flow from left to right or right to left in lines and the lines stack from top to bottom. A `pageArea` object, which represents a physical display surface, has the opposite limitation; it can only use positioned layout.

In XFAF forms top-to-bottom layout is used by the outer subform to represent the flow of pages. All other layout objects use positioned layout, except for the words and embedded objects within blocks of text.

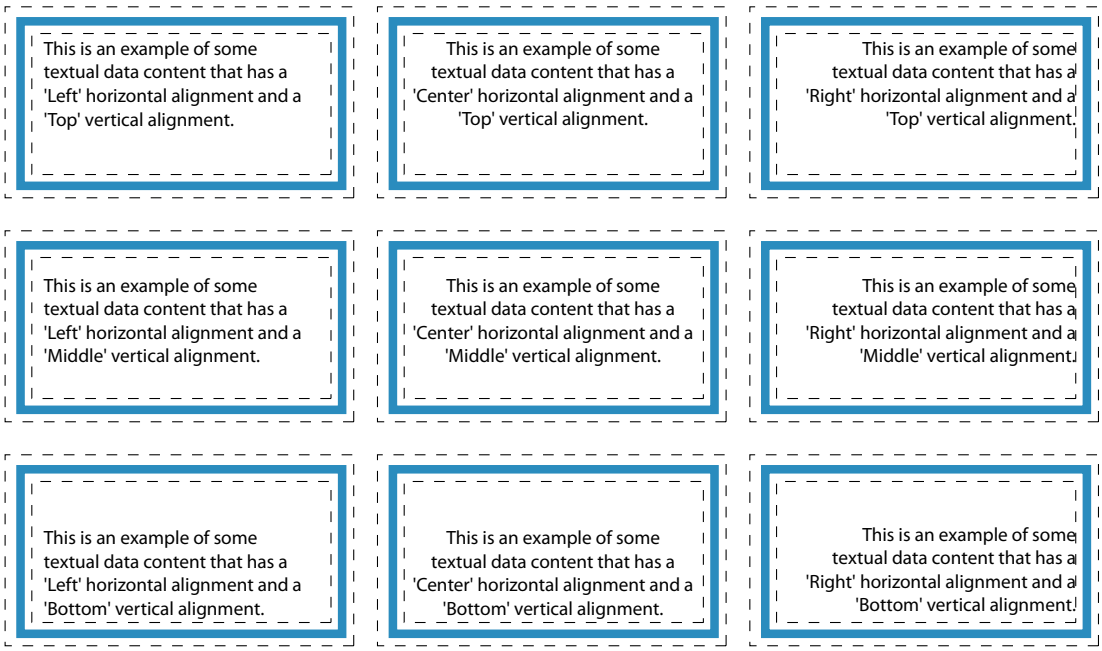
Paragraph and Font Formatting of Fixed and Variable Content

General text formatting characteristics, such as alignment and margins, are specified by the `para` element and font characteristics, such as font name and size, are specified by the `font` element.

Whether text is provided by the template or by an external source (such as a user filling out a form or a database server), text is organized as a series of records separated by newline indicators. In the case of user data, the newline indicator is inserted by the data loader as described in "Exchanging Data Between an External Application and a Basic XFA Form" on page 108. Each record consists of a stream of characters. In the case of rich text, each record may also include formatting markup and embedded objects.

Alignment and Justification

Text in a container element can be automatically aligned or justified relative to the vertical and/or horizontal borders of the containing element, as illustrated below. Alignment is specified in the para element associated with the field, as shown in the following examples.

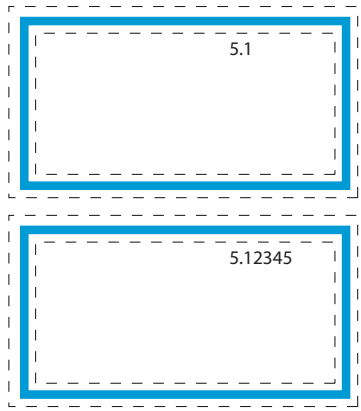


Text justification

Example 2.11 Horizontal and vertical alignment

```
<field name="example">
   <para hAlign="left" vAlign="top"/>
   <value>
      <text>This is an exmple of some textual data content that has a
"left" horizontal alignment and a "top" vertical alignment."</text>
   </value>
</field>
```

Text data that uses a radix separator (usually a floating point number) may be horizontally aligned around the radix separator, with the radix being a set distance from the right, as shown at right. A field element with such alignment is shown in the following example.



Example 2.12 Horizontal alignment relative to the radix separator

```
<field name="example" y="1in" h="1in" w="3in">
   <para hAlign="radix" radixOffset=".5in"/>
   <value>
      <text>5.1</text>
   </value>
</field>
<field name="example2" y="2in" h="1in" w="3in">
   <para hAlign="radix" radixOffset=".5in"/>
```

```
   <value>
      <text>5.12345</text>
   </value>
</field>
```

"Flowing Text Within a Container" on page 52 describes text placement and justification in more detail.

Line Height

To flow text and other flowed objects, the application may have to choose break points for wrapping. The result is that the flowed content is rendered as a series of one or more lines. The height of each line is known as the line height. If line height is not specified in a `para` element, it is typically determined from the current `font` element. If multiple fonts are used in a line, the processing application must use the maximum line height asserted by any of the fonts in the line.

Other Formatting

In addition to alignment, justification and line height, the `para` element may specify left and right margins, radix offset, spacing above and below the paragraph, tab stops and tab indenting.

In addition to line height, the `font` element may specify baseline shift, strike-through characteristics, overline, underline, angle, size, typeface, and weight.

Formatting Rich Text for Fixed and Variable Content

Rich text is text data that uses a subset of HTML and CSS markup conventions to signify formatting such as bold and underline. Rich text may also include embedded text objects. XFA supports the subset of HTML and CSS markup conventions described in "Rich Text Reference" on page 1027.

Rich text may appear in data supplied to the XFA form. Rich text may also appear in XFA templates as field captions or default text values.

Rich Text Used for Formatting

Rich text data is formatted as specified by the markup specifications in the rich text. The markup specifications take precedence over formatting specifications in the containing element, which appear in the `font` and `para` elements.

In general, GUI-based template design applications and XFA processing applications provide formatting buttons that allow users to apply styling characteristics to text. For example, the UI in such applications may provide a **Bold** button the user applies to selected text. In response, the application converts the entire body of in-focus text into a rich text representation and encapsulates the selected text within a `<b>` element, as shown in the example in the following example.

In the following example, the markup instructions specify the font family should be Courier Std and specify that the words "second" and "fourth" should be bold, as illustrated below the example XFA. Also notice the appearance of the attribute `allowRichText="1"`, which tells an XFA processing application that its UI must support entry of rich text values for the field.

Example 2.13 Rich text provided as a default field value

```
<field w="4in" h="4in">
   <value>
```

```
      <exData contentType="text/html">
         <body xmlns="http://www.w3.org/1999/xhtml">
            <p style="font-family:'Courier Std'">The<b> second </b>and
               <b> fourth </b>
               words are bold.
            </p>
         </body>
      </exData>
   </value>
   <ui>
      <textEdit allowRichText="1"/>
   </ui>
</field>
```

Produces:

The **second** and **fourth** words are bold.

The chapter "Representing and Processing Rich Text" on page 188 provides more detailed information on rich text.

Rich Text Used to Embed Objects

Embedded objects are objects which are imported into and flow with the surrounding text. Embedded objects can include variable data that is defined at run time. For example, the content of a field can be inserted into fixed text as an embedded object. An embedded object may contain non-textual data such as an image, in which case the embedded object is treated like a single (possibly rather large) character.

This application of rich text is described in "Rich Text That Contains External Objects" on page 193.

Formatting Variable Content

Variable content text may be formatted using picture clauses. Such formatting is in addition to that specified for paragraphs and fonts for rich text.

After variable content text is formatted using picture clauses, it may be further formatted as barcode representations, as described in "Barcode Formatting of Variable Text" on page 44.

Picture-Clause Formatting in General

XFA carries the distinction between data and presentation to the field level. Often, the user wishes to see individual field values embellished (e.g., with thousand separator, currency symbol). Yet the underlying data element may not contain those embellishments. An XFA picture clause provides the mapping between the two representations. The `format` element, a child of the `field` element, has a `picture` property, which specifies the presentation of the field's data.

While it may make sense to present a value in a verbose format (e.g., "Thursday, June 26 2003"), it could prove onerous if users have to enter data in this format. It might prove equally awkward to force users to enter dates in the underlying `date` element format (e.g., "20030626"). XFA-Template allows a second picture clause to be associated with the field, as a property of the field's `ui` element. For example, an American might enter the date as "06/26/2003", and have it presented as "Thursday, June 26 2003" after

tabbing out of the field. The picture clause associated with the `ui` element is referred to as an input mask. See "Picture Clause Specification" on page 991.

Locale-Specific Picture Clauses

A locale is identified by a language code and/or a country code. Usually, both elements of a locale are important. For example, the names of weekdays and months in the USA and in the UK are formatted identically, but dates are formatted differently. So, specifying an English language locale would not suffice. Conversely, specifying only a country as the locale may not suffice either — for example, Canada, has different currency formats for English and French.

When developing internationalized applications, a locale is the standard term used to identify a particular nation (language and/or country). A locale defines (but is not limited to) the format of dates, times, numeric and currency punctuation that are culturally relevant to a specific nation.

Localized formatting can be specified within the template (using picture clauses) or by allowing a picture clause to specify various formats for the supported locales. The XFA processing application then chooses the picture clause for the current locale or specified by the template locale.

Barcode Formatting of Variable Text

Barcodes are not a different data type but a different way of presenting textual or binary data. In XFA, a barcode is the visual representation of a field. The content of the field supplies the data that is represented as a barcode. For example,

Example 2.14 Field represented as a barcode

```
<field h="10.16mm" name="InventoryNumber" w="48mm" >
   <ui>
      <barcode dataLength="10" textLocation="belowEmbedded"
         type="code3Of9" wideNarrowRatio="3.0"/>
   </ui>
   <font typeface="Courier New"/>
   <value>
      <text>1234567890</text>
   </value>
</field>
```

Assume that the field `InventoryNumber` contains the default text. When printed the field appears as follows (actual size):

In the example, the `field` element's width (`w`) and height (`h`) attributes control the width and height of the barcode, hence the dimensions and spacing of the individual bars. In addition the field font, specified by the `font` element, controls the font used for the human-readable text embedded inside the barcode. The `type` attribute of the barcode element determines what type of barcode is printed. The example uses a 3-of-9 barcode. 3-of-9 barcodes are very common but not tightly specified so there are many parameters controlling the appearance of the barcode. Other attributes of the `barcode` element determine how

many characters are in the barcode (`dataLength`), whether there is human-readable text and where it is located (`textLocation`), and the ratio of the width of wide bars to narrow bars (`wideNarrowRatio`). There are other applicable parameters which are defaulted in the example.

Barcode readers are used only with printed forms; therefore, although a field may be designated as a barcode field it need not appear as a barcode in an interactive context. When the field has input focus some sort of text widget must be presented so that the user can edit the data. When the field does not have input focus the appearance is application-defined.

For more information about using barcodes in XFA see "Using Barcodes" on page 360.

Basic Layout

This section describes the most common aspects of how widgets are arranged and presented on the presentation medium. It explains how the objects that appear on a page are positioned relative to the page and to one another. This section also describes text flow and justification.

Box Model

In order to understand the ways in which relative positions are calculated it is necessary to understand the box model. Layout employs a box model in which model objects (both containers and displayable entities) are represented by simple rectangles called nominal extents. Each nominal extent is aligned with the X and Y axes and represents the amount of physical space on the page that is reserved for a particular object. Some nominal extents are calculated at run time, for example the extents for blocks of variable text in fields. Other nominal extents are presupplied.

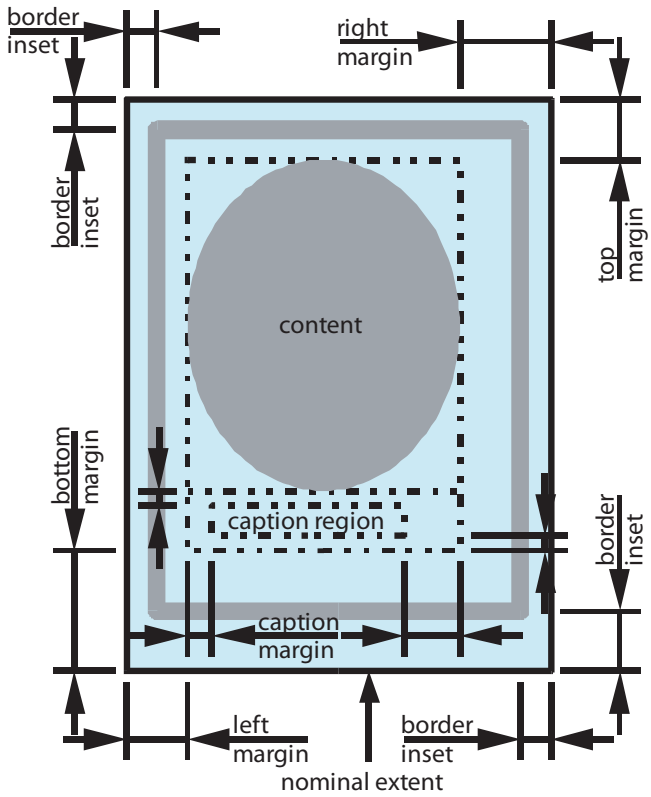
Fields in XFAF forms have nominal extents supplied explicitly via the `w` (width) and `h` (height) attributes. Each of these attributes is set to a measurement. The field's margins, if any, and its caption, if any, lie inside the nominal extent.

Fields may have borders. In XFAF forms the borders always lie inside the nominal extent of the field.

Fields are containers with box models, but so are captions which reside within fields. A caption may contain an image or text, either of which has its own box model. Hence the box model is recursive; box models may contain box models.

The relationship between the nominal extent, borders, margins, captions, and content is shown at right.The Nominal Content Region is the rectangular area left over after container's margins have been applied to its nominal extent. This is the space normally reserved for display of and interaction with the container object's content. Note that the caption may occupy part of the nominal content region.

The rules governing width and height for each type of layout object are given below:



Relationship between nominal extent and borders, margins, captions, and content

Barcode

There are two types of barcodes, one-dimensional and two-dimensional.

For some one-dimensional barcodes the width is fixed by the barcode standard. For others the width varies with the number of symbols in the data and the presence or absence of check symbol(s). In addition for some one-dimensional barcodes the width of a symbol is adjustable.

For some one-dimensional barcodes the height is fixed by the standard. For others the height is adjustable.

For two-dimensional barcodes the width varies with the number of columns and the cell size. The height varies with the number of symbols in the data, the presence or absence of check symbol(s), and the cell size.

Note that barcode standards often dictate a margin around the barcode as well. The barcode size as computed by the layout processor does *not* include any such mandated margin. It is up the creator of the template to set the appropriate margin in the container. Hence the form creator can “cheat” as desired.

Captions

Captions may have explicit heights or widths supplied by a `reserve` attribute. The reserve is a height if the caption is placed at the top or bottom of its container, but a width if the caption is placed at the left or right of its container. When the `reserve` attribute is not supplied or has a value of zero, the layout

processor calculates the minimum width or height to hold the text content of the caption. This calculation is described in "Text" on page 37.

Fields

Fields may have explicit widths supplied by `w` attributes and/or explicit heights supplied by `h` attributes. When either of these attributes is not supplied, the layout processor must calculate it.

The first step is to compute the size of the field content. If the field contains text the width and/or height of the text content is first calculated as described in "Text" on page 37. If the field contains an image the height and/or width of the image content is calculated as described in "Images" on page 38.

After this, if there is a caption, the layout processor adjusts the height and/or width to account for the caption. When the caption is on the left or right, the width is adjusted by adding the caption width plus the caption left and right margins. When the caption is above or below, the height is adjusted by adding the caption height plus the caption top and bottom margins.

Images

Depending upon the value of the `aspect` property an image may keep its original dimensions, it may grow or shrink in both dimensions while retaining its original aspect ratio, or it may grow or shrink independently in each dimension to fill the container. In all but the last case it may display outside the nominal extent. Some image formats do not specify a physical size, only horizontal and vertical pixel counts; in such cases the application assumes a pixel size to arrive at a natural width and height. The assumed pixel size is application-dependent and may vary with the graphic file type and/or display or print device. The following figure shows an example of resizing an image in a container using different `aspect` settings.

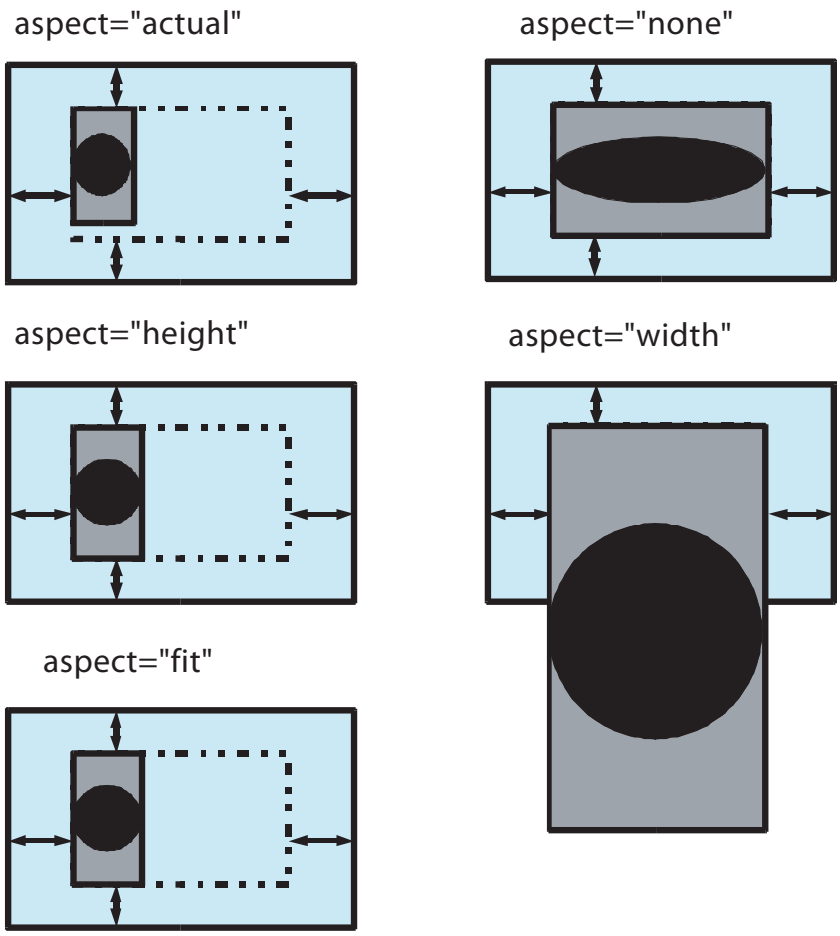


Image with different aspect settings

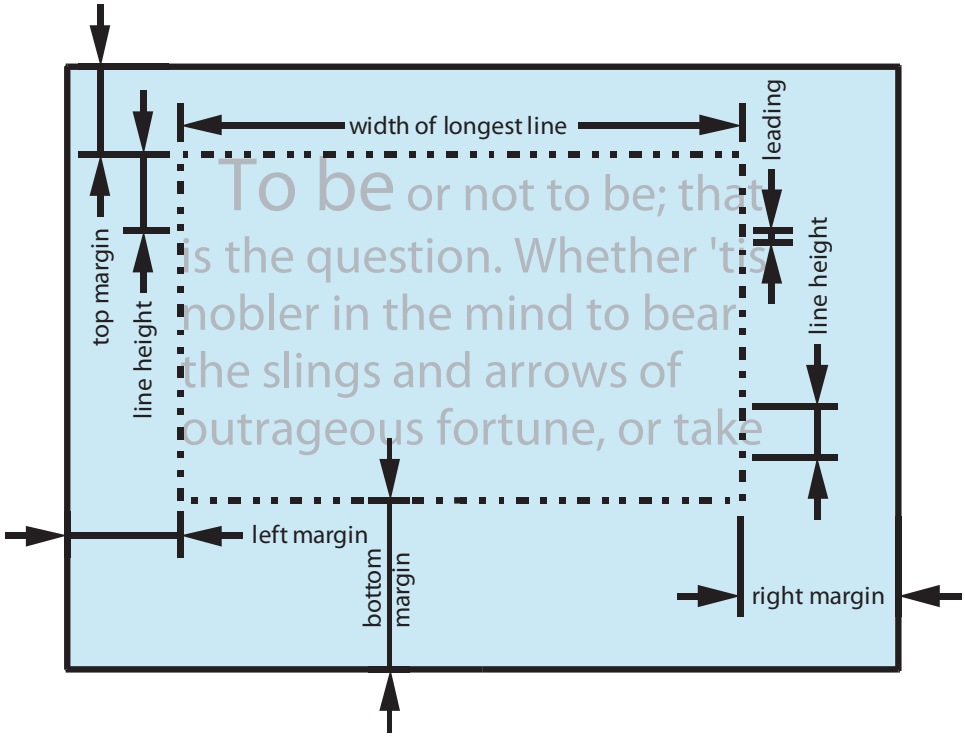
Subform

The `subform` element describes a logical grouping of its child elements. In XFAF forms subforms have no visible appearance, hence no box model.

Text

The height calculation for each line must take into account the height of each character (which depends on the type face and size), the positioning of characters above the baseline (superscript) and below the baseline (subscript), and underlining. The overall height is the sum of the individual line heights, plus the sum of the leading between lines.

The width calculation for each line must take into account the width of each character, including white space characters, and for the first line the text indent. The overall width is the largest of the line widths. The following figure shows these quantities.



Layout quantities for a text object

Size, Margins and Borders of Widgets

A widget is used here to refer to a simulated mechanism displayed by the user interface to enable the user to enter or alter data. For example, a check box displayed on a monitor, which checks or unchecks in response to a mouse click, is a widget. "Widgets" on page 406 describes the user experience with widgets.

Size

Most widgets do not have explicit size attributes. All widgets can resize themselves to fit the containing field; if the field containing the widget has a specified size then the widget's extent grows or shrinks to fit the imposed size. If no size is imposed on the widget it expresses its natural size. The natural size of a widget and the factors that control it vary with the widget type.

A widget may temporarily grow to occupy a larger area of the screen while the widget has focus (like a drop-down list). However as soon as focus is lost the widget goes back to its normal size. The table below lists sizes for widgets when they do not have focus and also when forms including images of a widgets are printed to hard copy. The size of a widget while it has focus is up to the implementation.

Natural size of widgets

Widget	Natural width	Natural height
button	Has no natural size (or a natural size of zero) because it does not display any content. A field containing a button displays only the field caption and field borders, but with their appearance changed to indicate clickability.	
check box or radio button	The size property plus left and right margins. Defaults to 10 points and no margins.	The size property plus top and bottom margins. Defaults to 10 points and no margins.

Natural size of widgets

Widget	Natural width	Natural height
choice list	Same as a text edit.	Same as text edit.
date, time, or date-time picker	May be displayed temporarily by the application during data entry while the field has focus. When the field loses focus it reverts to the same display rules as a text edit.	
image	Same as text edit.	Same as text edit.
numeric edit	Same as a text edit.	Same as text edit.
password edit	Same as a text edit, except each character of content is replaced with an asterisk ("*") character.	
signature	Implementation-defined.	
text edit	The width of the text block plus the left and right margins. Defaults to no margins.	The height of the text block plus the top and bottom margins. Defaults to no margins.

Margins and Borders

A widget, other than a button, may have its own margins and borders in addition to any margins and borders asserted by its container. When a text widget declaration has a margin element but the element asserts no attributes the margin default varies depending upon the presence of a border. Text widgets include date, time, and date-time widgets, choice lists, numeric edit, password edit and text edit widgets, and signature widgets. For these widgets when there is no border the default margin is zero on all sides. When there is a border the default margin is twice the border thickness on all sides, plus an additional amount on the top margin equal to the descent of the font (for a rich-text field, the default font.) By contrast, for all other widgets the margin default is always zero on all sides whether there is a border or not.

Size Requirement for Containers

For images the `w` and `h` attributes are mandatory. If either of these is not supplied the XFA processor should either infer a size from context or default the value to 0. Whichever strategy is taken the goal is to continue processing even if the result is unattractive.

Clipping

When the content does not fit into the container the excess content may either extend beyond the region of the container or be clipped. The permissible range of actions varies according to the type of container and the context (interactive or non-interactive).

When the container is a field and the context is interactive, the content of the field may be clipped. However some means must be provided to access the entire content. For example, the XFA application might arrange that when a field gains focus a widget pops up. The widget could be dynamically sized or it could support scrolling.

When the container is a field and the context is non-interactive (for example printing to paper) the content must not be clipped. The content may be allowed to extend beyond the field or it may be shrunk to fit the field.

When the container is a caption the behavior is implementation-defined. It is the responsibility of the form creator to ensure that the region is big enough to hold the content.

Rotating Containers

Container elements may be rotated about their anchor point. Rotation is in degrees counter-clockwise with respect to the default position. Angles are supplied as non-negative multiples of 90. In the following example, the field is rotated counter-clockwise 90 degrees.

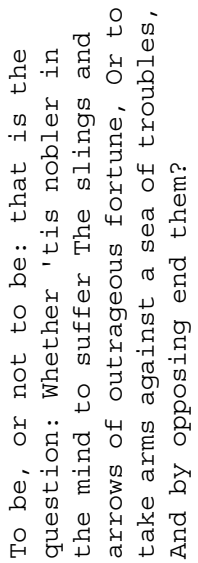
Rotating Positioned Content

The following example shows how a container is rotated.

Example 2.15 A field rotated 90 degrees rotates its contents

```
<field name="soliloquy" anchorType="topLeft" rotate="90"
   y="100.00mm" x="40.00mm" w="78.00mm" h="50.00mm">
   <value>
      <text>To be, or not to be: that is the
question: Whether 'tis nobler in
the mind to suffer The slings and
arrows of outrageous fortune, Or to
take arms against a sea of troubles,
And by opposing end them?</text>
   </value>
</field>
```

The result of rendering the above field element is shown at right.



Rotating Flowed Content

The rotation of containers in a dynamic form affects the flowed placement of containers, as described in "Effect of Container Rotation on Flowed Layout" on page 245.

Transformations

Presenting the form to the user, or printing it to paper, requires that the many objects inside the template be assembled by the processing software. During this assembly, many geometric transformations must

take place. A container object must position or flow its enclosed objects within its nominal content region, using its own coordinate space. If a container holds other containers as its content, those child containers in turn position or flow their content within their own coordinate spaces.

It is not the responsibility of this document to mandate the actual implementation of transformations. However, by describing one possible implementation, transformation calculations become more obvious. This particular implementation is provided to be illustrative of transformations.

In this implementation, there is a clear separation between what the enclosed object knows and what the enclosing container knows. Regardless of margins and the internal coordinate origin, the enclosed object adjusts the coordinates that it makes available to the container so that (0,0) is the top-left corner of the contained object's nominal extent. We refer to these as common coordinates, which are coordinates the parent can easily transform into its own coordinate space. See "Algorithms for Determining Coordinates Relative to the Page" on page 1047.

Flowing Text Within a Container

This section describes the placement of text within a fixed-size container. It does not discuss the placement of text within a growable container. Placement of text within growable containers is discussed in"Text Placement in Growable Containers" on page 240.

A container's data is flowed inside the nominal extent of the container. The flow of text within a line can be left to right or right to left or a mixture of the two. Lines always flow from top to bottom.

The enclosing container may specify the alignment of text within the object ("Alignment and Justification" on page 41) and may specify whether the container may grow in width and/or height ("Growable Containers" on page 238). The enclosing container can also control the initial direction of text flow (left to right or right to left) and rules for word parsing by specifying a locale. The enclosing container cannot specify any line-wrapping technique; however, XFA processing applications typically employ some a wrapping technique to place a long stream of flowed content into its nominal content region. Text is typically flowed over one or more lines, breaking at word boundaries.

The examples in this section assume the text stream shown below, which also serves to illustrate the organization of text data. The sequence `"&nl;"` stands for the newline indicator, which separates text records.

```
To be, or not to be: that is the question:&nl;Whether 'tis nobler in the mind
to suffer&nl;The slings and arrows of outrageous fortune,&nl;Or to take arms
against a sea of troubles,&nl;And by opposing end them?
```

When placing text into a fixed-width region the text records are interpreted as paragraphs. The layout processor starts each new paragraph on a new line. Within each paragraph the layout processor treats the text as a string of text layout units. A text layout unit may be an embedded non-text object or it may be a word. The boundaries of layout units may be delimited by the edges of embedded non-text objects or by white space characters. In many languages text layout units are words, separated by *white space*. However not all languages use white space to delimit words. Therefore, the parsing of words is language-dependant. Which languages are supported is implementation-dependant, but all implementations should support some locale that uses the Unicode Basic Latin and Latin-1 Supplement character codes (U0021 through U007E inclusive). The complete rules for parsing words are given in Unicode Standard Annex 14 [UAX-14]. Note that these rules were first adopted in XFA 2.2.

The initial flow direction depends on the locale. However a single paragraph of text can contain text from different languages that flows in different directions. For example, the language of the locale is Arabic so the initial flow direction is right to left. However the text contains a phrase in English that flows from left to

right. The flow direction of the English phrase may be explicitly indicated by codes reserved for this purpose in Unicode, or the text engine can infer it from the use of English letters in the phrase. The whole complicated set of rules is specified in Unicode Annex 9 [UAX-9]. By applying these rules the text layout engine divides the paragraph into segments, such that within each segment the flow is all in one direction.

Note: Unicode Annex 9 has been widely embraced and incorporated into many other standards. However Microsoft has its own way of doing things which is close to, but not quite in accordance with, the Unicode standard. Do not expect XFA flow direction to exactly match the behavior of non-conformant software.

If the container has fixed dimensions and the flowed text exceeds the boundaries of the container, the view of the text must be adjusted as described in "Clipping" on page 50.

Text Layout in the Horizontal Direction

Layout units within a paragraph are placed in document order in the flow direction on each line, along with any white space between them. The line stops when the text is exhausted or the next thing is a text layout unit (i.e. not white space) and it cannot be placed on the line without exceeding the width of the region. At this point trailing white space on the line is discarded.

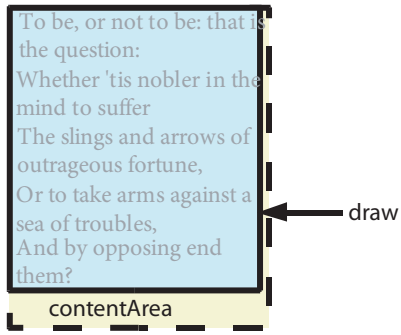
Next the layout units within the line are reordered in accordance with the direction of the segment in which each is embedded. See Unicode Annex 9 [UAX-9] for details.

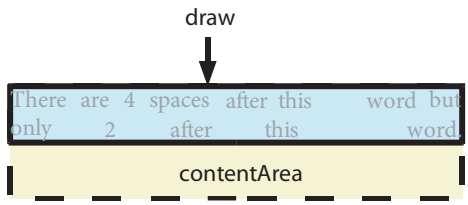
The line is now ready to be positioned within the region. The tables "Influence of hAlign on alignment of text" on page 54 further describe horizontal text alignment.

Regardless of the type of alignment being performed, if a single word is too wide to fit into the region the layout processor must break the word between characters. Furthermore, if a single character or embedded non-text object by itself is too wide to fit into the region it is allowed to extend beyond the region. In this case, if the vertical alignment is `center` or `right`, the layout processor may sacrifice vertical alignment in order to avoid assigning negative coordinates to the character or embedded object.

Influence of hAlign on alignment of text

hAlign	Description	Illustrated Effect

`left`
`right`
`center`
	If `hAlign` is set to `left` the line is positioned so that its left-most text layout unit abuts the left edge of the region. If `hAlign` is set to `right` the line is positioned so that the right-most text layout unit abuts the right edge of the region. If `hAlign` is set to `center` the line is positioned so that the middle of the line is at the middle of the region. The figure at right shows an example of justification with the `hAlign` attribute set to `left`.
	For these values the effect is the same regardless of the flow direction.
		

`justifyAll`	For each line, instead of incorporating the supplied white space characters, blank regions are inserted between layout units, one per breaking whitespace character in the original text, sized, until the line fills the region from left to right. The effect is the same regardless of the flow direction. In the example at right the template contained the following declarations: `<field …> <text>There are 4 spaces after this word but only 2 after this word.</text> </field>`	

Influence of hAlign on alignment of text (Continued)

hAlign	Description	Illustrated Effect
justify	All lines are justified except the last one in the paragraph. The last line is left aligned if the initial flow direction is left to right and right aligned if the initial flow direction is right to left.	
radix	If hAlign is set to radix, then the text is treated as a column of numbers, one per line. In this case the radixOffset property supplies the position of the radix character (" . " or " , " depending upon the locale). Each line is positioned so that the left edge of the radix character's layout extent is at the radixOffset distance from the right edge of the region. If the line of text does not contain a radix character the right edge of the line's layout extent is positioned at the same point, so that the line is treated as an integer and aligned with the place where it's radix point would have been. If a line contains more than one radix character the first one (in character order) is the one used for alignment. Radix alignment can only be used with place-value numbering systems. The same algorithm works for all known place-value numbering systems, because they all flow left to right from the most significant to the least significant digit. This is a consequence of the historical spread of place-value notation from India to Arabia and thence to Europe..	

Text Layout in the Vertical Direction

In the vertical direction, the region may not all be usable, because some portion may be reserved for a caption. Vertical alignment within the usable region is controlled by the vAlign attribute. If vAlign is set to top the first line of text is positioned at the top of the usable region, with the next line positioned below it (separated by the leading) and so on. If vAlign is set to bottom the last line of text is positioned at the bottom of the usable region, with the previous line positioned above it (separated by the leading) and so on. If vAlign is set to middle the middle of the text lines is positioned to the middle of the usable region.

Concealing Container Elements

XFA template provides several features that allow parts of a form to be concealed under various circumstances.

Concealing Containers

A container may be hidden from view by setting its presence attribute to hidden or invisible. A value of hidden prevents the container and its content from being displayed on the form. A value of

`invisible` also prevents the container and its content from being displayed, however they still take up space on the form. In static forms the space taken by a field is fixed however the location of the associated caption may shift if the field is `hidden`.

When an outer container contains inner containers, and the outer container has a `presence` of `hidden` or `invisible`, the inner containers inherit the outer container's `hidden` or `invisible` behavior regardless of their own `presence` attribute values.

Even when `hidden` or `invisible` a container is still present in the form and takes part in normal non-interactive activities. For example a `hidden` or `invisible` field may be used to calculate a value and hold the result of the calculation. Alternatively it may hold data that is loaded from an external file or database so that the data can be used in other calculations.

Concealing Containers Depending on View

A form designer may specify that containers in a form should be revealed or concealed depending on the view. This feature uses a `relevant` attribute available on most template container elements and a corresponding `relevant` element in the configuration syntax grammar ("Config Specification" on page 761).

Usually the `relevant` element in the configuration contains a single *viewname*. A viewname is an arbitrary string assigned by the template creator. For example, the template creator might use the name "print" to identify a view of the form to be used when printing a hard copy. Viewnames must not contain space characters (U+0020) or start with the minus ("-") character (U+002D).

If a template container has no `relevant` attribute, or the value of the attribute is an empty string, then the template container is included in all views. This is the default. However using the `relevant` attribute any given template container may be explicitly included in or excluded from particular views. The content of the `relevant` attribute is a space-separated list of viewnames, each of which may optionally be prefixed with a "-". If a viewname is preceded by a "-" then the container is excluded from that particular view; if not it is included in the view.

It is also possible for the `relevant` element in the configuration to specify a list of viewnames. In this case the effect is to exclude any container which is excluded from *any* of the listed views.

When a container is excluded it and its content are removed from the XFA Template DOM so that as far as the XFA processor is concerned it and its content do not exist. However if the template is subsequently saved as an XML document the excluded container is retained in that document along with its `relevant` attribute and all of its content.

The following steps determine whether a container is included in or excluded from a particular view:

1. If the configuration `relevant` element is unspecified or empty, all containers are included, regardless of their `relevant` attributes.

2. If the container's `relevant` attribute is unspecified or empty, it is included regardless of the configuration `relevant` element.

3. If the configuration specifies a non-empty `relevant` element, then every template container element having a non-empty `relevant` attribute is evaluated for inclusion, as follows.

   - If the template element specifies a `relevant` value that includes a viewname that is *not* preceded by "-" and is *not* specified in the configuration, the element is excluded.
   - If the template element specifies a `relevant` value that includes a viewname that *is* preceded by "-" and *is* specified in the configuration, the element is excluded.

- Otherwise the element is included.

The following table provides examples of the inclusion or exclusion of container elements based on the template and config `relevant` values.

Inclusion/exclusion of container elements based on correlation between "relevant" value in template container element and config

Template relevant attribute	Config relevant element				
	`print`	`manager`	`manager print`	`summary`	`""` **(null)**
`print`	included	excluded	included	excluded	included
`manager`	excluded	included	included	excluded	included
`-print`	excluded	excluded	excluded	excluded	included
`-manager`	‘included	excluded	excluded	excluded	included
`print -manager`	included	excluded	excluded	excluded	included
`-print manager`	excluded	included	excluded	excluded	included
`-print -manager`	excluded	excluded	excluded	excluded	included
`""` (null)	included	included	included	included	included

Exclusion of Hidden Containers in the Data Binding Process

If a container is hidden as a result of settings in the `presence` attribute, the container included in the XFA Form DOM and is considered in the data binding process. Calculations may provide values to such hidden containers; however, the person filling out the form cannot.

If a container is hidden as a result of the `relevant` attribute, it is excluded from both the XFA Template DOM and the XFA Form DOM. Hence it plays no part in the data binding process.

Appearance Order (Z-Order)

While the form coordinate space is two-dimensional, it should be recognized that container elements appear to exist at different depths. Objects at shallower depths (closer to the user) may visibly obscure parts of objects at deeper depths. This notion of a depth-based order is the Z-order of the form. One can think of objects that come early in the Z-order as being placed on the presentation media earlier in the construction of the form. Objects that come later are placed over top of the earlier objects.

Each subform encloses other container objects. The subform imposes an implicit Z-order for those containers, which is simply the order in which the children occur; the first object exists at a deeper Z-order depth than the last object. This is convenient for many drawing APIs, where objects drawn later appear on top of objects drawn earlier.

Within an container's box model, there is also a Z-ordering. Content will always appear on top of the border when the two overlap.

GUI-based template design applications may provide the ability to move an object in front of or behind another object. Such movement is accomplished by changing the order in which the objects appear in the document.

In the following example, `TextField2` obscures `TextField1`. `TextField1` could be placed on top of `TextField2` by simply reversing the order in which they appear in the document.

Example 2.16 "TextField2" obscures "TextField1"

```
<field name="TextField1" y="70.00mm" x="60.00mm" h="15.00mm" w="30.00mm">
   <ui>
      <textEdit>
      </textEdit>
   </ui>
</field>
<field name="TextField2" y="70.00mm" x="60.00mm" h="15.00mm" w="30.00mm">
   <ui>
      <textEdit>
      </textEdit>
   </ui>
</field>
```

Note: XFA processors are not expected to do anything to prevent fields and their associated widgets on the same page from overlapping. It is up to the creator of the form to ensure that the extents assigned to the fields do not overlap, if that is what is desired.

Extending XFA Templates

The XFA template grammar defines the `extras` and `desc` elements, which can be used to add human-readable or machine-readable data to a template. These elements provide the same properties with the following exceptions: the `extras` element may be named and may contain child `extras` elements. These elements differ in their intended use; however, they can both be used to extend XFA templates.

Use of custom namespaces in the template grammar is not permitted because the XFA template grammar is a standard for interactive forms. That is, an XFA document should cause the same behavior, regardless of the XFA processing application that processes it.

XFA processor may also copy custom data into the XMP packet of a document to make it quickly accessible without the burden of parsing the entire template. This is a new feature in XFA 2.5.

Adding Custom Named and/or Nested Data to a Template (extras)

The `extras` element is used to add comments, metadata, or datatyped values to template subelements. An `extras` element is named and may contain multiple child `extras` elements. The `extras` element is intended for use by XFA processing applications to support custom features.

The data contained in `extras` elements is loaded into the Template and Form DOMs, which makes it available to scripts. Because the type of the data is indicated in the markup the data is available to scripts in its native format, not only as a string. (Unless it is enclosed in <text> in which case string is its native format.)

See also the "Template Specification" description of the `extras` element.

Adding Metadata or Comments to a Template (desc)

The `desc` element is used to add comments, metadata, or other datatyped values to its parent element. It may also be used to house information that supports custom features, as with the `extras` element.

Element comments represented in a `desc` element remain with the template throughout its life. In contrast, XML comments in an XFA template are lost when the template is represented as a Data Object Model. Such information may be used for debugging or may be used to store custom information.

Example 2.17 Metadata in a template subform

```
<subform …>
   …
   <desc>
      <text name="contact">Adobe Systems Incorporated</text>
      <text name="description">Adobe Designer Sample</text>
      <text name="title">Tax Receipt</text>
      <text name="version">1.0</text>
   </desc>
</subform>
```

See also the "Template Specification" description of the `desc` element.

Connecting the PDF to the XFA Template

An XFAF form of necessity includes (or is embedded in) a PDF file. There are several things that must be set up in the PDF to make it operate with the XFA side of the form.

Flags

The PDF specification [PDF] defines a NeedsRendering flag in the the catalog dictionary (document catalog) which controls whether PDF viewers attempt to regenerate the page content when the document is opened. Only dynamic templates contain the information required to regenerate the full page content including the boilerplate. XFAF templates do not contain this information so for XFAF forms the flag must be false.

Resources

An XFAF form may be contained in an XDP document. The XDP may be a standalone document which also contains the associated PDF. However it is more common for the XFAF form to be embedded inside a PDF file. When the PDF contains the XFAF form, the XFA template and other XFA packets are contained in an entry in the interactive form dictionary with the key `XFA`. The interactive form dictionary is referenced from the `AcroForm` entry in the document catalog. The presence of these entries tells the PDF processor that the form is an XFAF form. For more information see the PDF specification [PDF].

Field Names

PDF has long included support for interactive fields. Each interactive field has a *partial name*, as described in the PDF specification [PDF]. When the interactive field is global (i. e. not inside another field) its partial name is also its fully qualified name. In an XFAF form each PDF field is global and its name is an expression which points to the corresponding `field` element in the XFA template.

The expressions used to link PDF fields to XFA fields employ a grammar known as the XFA Scripting Object Model (XFA-SOM). This grammar is used throughout XFA and is described in the chapter "Object Models in XFA" on page 62. The grammar is non-trivial, but fortunately the whole grammar is not required for this particular purpose. For this purpose it is enough to know that the expression consists of `form` followed by the names of the objects (starting with the outermost subform) that must be entered in turn to reach the field. The names are separated by dot ('.') (U+002E) characters. Usually XFAF forms have a subform per page plus a root subform holding all of the page subforms, so this works out to `form.`*`rootname`*`.`*`pagename`*`.`*`fieldname`*.

Note: For normal AcroForm fields the partial field name is not allowed to contain the dot character. This restriction does not apply to fields which are simply pointing to an XFA field.

Example 2.18 PDF field names for a typical XFAF form

```
<template ...>
   <subform name="PurchaseOrder" ...>
      <subform name="CustomerID" ...>
         <field name="FirstName" ...>
         ...
         </field>
         <field name="LastName" ...>
         ...
         </field>
      </subform>
   </subform>
</template>
```

In this example the PDF field linked to the first field in the template would be named `form.PurchaseOrder.CustomerID.FirstName`. The PDF field linked to the second field would be named `form.PurchaseOrder.CustomerID.LastName`.

It is also possible for there to be multiple subforms and/or multiple fields with the same name and the same enclosing element. These are distinguished using a numeric index, with 0 as the index of the first same-named element.

Example 2.19 PDF field names for an XFAF form with duplicate names

```
<template ...>
   <subform name="PurchaseOrder" ...>
      <subform name="Items" ...>
         <!-- The first row in a table -->
         <field name="PartNo" ...> ... </field>
         <field name="Description" ...> ... </field>
         <field name="Quantity" ...> ... </field>
         <field name="UnitPrice" ...> ... </field>
         <field name="Price" ...> ... </field>
         <!-- The second row in the table -->
         <field name="PartNo" ...> ... </field>
         <field name="Description" ...> ... </field>
         <field name="Quantity" ...> ... </field>
         <field name="UnitPrice" ...> ... </field>
         <field name="Price" ...> ... </field>
         <!-- The third row in the table -->
         <field name="PartNo" ...> ... </field>
         <field name="Description" ...> ... </field>
         <field name="Quantity" ...> ... </field>
```

```
        <field name="UnitPrice" ...> ... </field>
        <field name="Price" ...> ... </field>
        <!-- More rows in the table -->
        ...
      </subform>
    </subform>
</template>
```

In this example the PDF field linked to the part number field in the first row of the table would be named `form.PurchaseOrder.Items.PartNo[0]`. The PDF field linked to the part number field in the second row would be named `form.PurchaseOrder.Items.PartNo[1]`, and so on.

Field location

Each XFA field corresponds to a PDF interactive field (AcroForm field) as described in "Field Names" on page 59. The PDF field name locates its corresponding XFA `field` element on the particular page of the PDF that contains the PDF field. However the `field` element's own `x` and `y` attributes determine its location on the page, regardless of the position of the PDF field on the page.

3 Object Models in XFA

This chapter describes the manner in which data objects are named, stored, manipulated, and referenced by XFA processing applications.

This chapter contains the following sections:

- "XFA Names" explains the syntax of names that can be assigned to individual template elements. Such names are important in the subsequent sections in this chapter.
- "Document Object Models" introduces the various Document Object Models (DOMs) used in XFA and discusses the general rules governing their relationship to XML. Then, it gives an overview of how the various DOMs interact with each other and with XFA processors.
- "Scripting Object Model" introduces the expression syntax used to refer to objects in the DOMs. This expression syntax is known as the XFA Scripting Object Model (XFA-SOM).

XFA Names

An XFA name is a valid XML name, as defined in the XML specification version 1.0 [XML], with the additional restriction that it must not contain a colon (:) character. XFA element names are used in the following ways:

- Explicitly identifying an object in an XFA DOM, using the XFA Scripting Object Model syntax ("Scripting Object Model" on page 73)
- Associating data with template fields, as part of data binding

The XML Specification version 1.0 defines name as follows:

> Name is a token beginning with a letter or one of a few punctuation characters, and continuing with letters, digits, hyphens, underscores, colons, or full stops, together known as name characters. Names beginning with the string "xml", or with any string which would match (`('X'|'x') ('M'|'m') ('L'|'l')`), are reserved for standardization in this or future versions of this specification.
>
> Note: The Namespaces in XML Recommendation [XMLNAMES] assigns a meaning to names containing colon characters. Therefore, authors should not use the colon in XML names except for namespace purposes, but XML processors must accept the colon as a name character.
>
> An Nmtoken (name token) is any mixture of name characters.

```
NameChar   ::=  Letter | Digit | '.' | '-' | '_' | ':' | CombiningChar
                  | Extender
```

[This specification precludes the use of ':' in a name.]

```
Name       ::=  (Letter | '_' | ':') (NameChar)*
```

[This specification precludes the use of ':' in a name.]

Document Object Models

General Information

A Document Object Model (DOM) is a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of a document. DOMs are commonly used with data expressed in XML. Indeed the preceding definition of a DOM is paraphrased from the W3C XML DOM standard [XMLDOM2].

In this specification many operations are described in terms of their effects upon the DOMs. This permits the specification to be detailed enough for implementation without imposing any requirement upon the language used for the implementation. Although the contents of a DOM are sometimes referred to as objects, this terminology does not mean that implementations must be written in an object-oriented or even object-based language.

However XFA does include scripting facilities. Both of the scripting languages currently defined for use with XFA are object-based. Hence, in order to make scripts interoperate across different XFA implementations a common object models must be presented. Furthermore, almost the entirety of the various DOMs are exposed to scripts. Hence, although an implementation need not be object-based internally it must present to scripts an object-based API covering most of its functionality.

All of the DOMs used in XFA share the following characteristics:

- They are strictly tree-structured.
- A node may have mandatory children. In such cases the mandatory child nodes are created at the same time as their parent.
- The non-mandatory children of each node in the tree are ordered by age. That is, the DOM is aware of the order in which the non-mandatory child nodes were added.

There are many DOMs in XFA. The large number arises because XFA exposes almost all processing. For each step in processing there is a DOM holding the data structures for that stage. Scripts can examine and modify each DOM. DOMs are responsible for maintaining internal consistency but not external consistency. For instance, when a script turns on a radio button by assigning to the corresponding field, all the other buttons coupled to that one are automatically turned off. This is a matter of internal consistency so it is managed by the Form DOM itself. By contrast the XFA Data DOM does nothing to prevent a script violating the rules of XML, for instance by giving an element two attributes with the same name. This is a matter of external consistency so it is the responsibility of the script author, not the DOM.

Hierarchy of DOMs

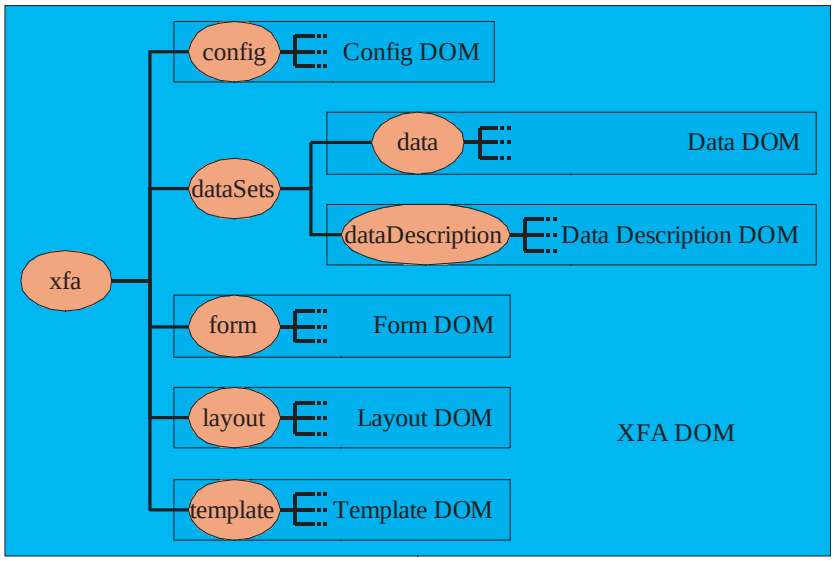
The XFA DOM encapsulates all but one of the other DOMs. The root nodes of most of the other DOMs are children of the root node of the XFA DOM. All but one of the remaining root nodes are grandchildren of the root node of the XFA DOM. The only DOM used in XFA that is not inside the XFA DOM is the XML Data DOM.

Note: If an application has more than one form open simultaneously, each open form has its own separate XFA DOM. There is no way for an expression or script in one form to refer to another form.

This encapsulation is convenient for scripting. Scripts refer to nodes in the DOM using XFA Scripting Object Model (XFA-SOM) expressions. All of the DOMs contained within the XFA DOM can be referenced using a uniform syntax. The XML Data DOM, because it is not inside the XFA DOM, is not directly accessible to

scripts. However the XFA Data DOM is tightly coupled to the XML Data DOM and exposes some of the XML Data DOM's content to scripts.

The following illustration shows the hierarchy of the principal DOMs in XFA.



Hierarchy of the DOMs

DOMs and XML

Most of the DOMs used in XFA can be loaded from or written out as XML documents. One such document is an XDP, which like the XFA DOM is a container for subtrees representing other DOMs.

The XML DOM specification [XMLDOM2] defines a standard way in which a DOM loads from XML and is stored in XML. The various XFA DOMs differ slightly from the norm in their relationship to XML documents. First, they treat white space differently. Second, they distinguish between properties and children, a distinction that is not made in [XMLDOM2].

Note that a similar hierarchy is used in XDP documents to represent a collection of DOMs in a single document. However an XDP may contain subtrees that are not represented by a DOM (for example application-specific data). At the same time some DOMs (such as the Layout DOM) are never represented in an XDP. For this reason in an XDP the outermost element tag is `xdp` rather than `xfa`.

Grouping Elements and Whitespace

In XML whitespace is significant by default. However, many schemas allow the free inclusion of whitespace within *grouping elements*. A grouping element is one that directly contains only other elements, not literal content. Freedom to include whitespace allows indentation which makes XML more readable.

Within XFA schemas all grouping elements may include whitespace. When loading an XFA DOM whitespace inside grouping elements is discarded. When unloading an XFA DOM whitespace may legally be inserted; whether it is, how much, and where are application-defined.

Properties vs. Children

The W3C XML DOM specification [XMLDOM2] distinguishes elements, attributes, and content. In other words a DOM constructed in accordance with [XMLDOM2] simply echoes the XML infoset. By contrast XFA DOMs distinguish between *properties* and *children*.

Properties

Properties are nodes that are automatically present in the DOM even if they are missing from the XML. For example, a `subform` node in the Template DOM always has beneath it a `bind` property node, whether or not the XML from which it was loaded included a `bind` element. If there is no `bind` element in the XML then at load time the `bind` property node is created with default values.

Some properties are represented by elements, others by attributes. Property attributes are restricted by the rules of XML to be singly-occurring and placed within their respective start tags. Property elements can physically occur more than once per parent, but in every case the number of occurrences is restricted by the DOM. For example, there can only be one `bind` property node per `subform` parent node, hence only one `bind` element per enclosing `subform` element. Most properties are restricted to a single occurrence. In addition, properties represented by elements may themselves have subproperties. For example the `bind` property has `match`, and `ref` subproperties. However there is no logical distinction between element and attribute properties at the DOM level.

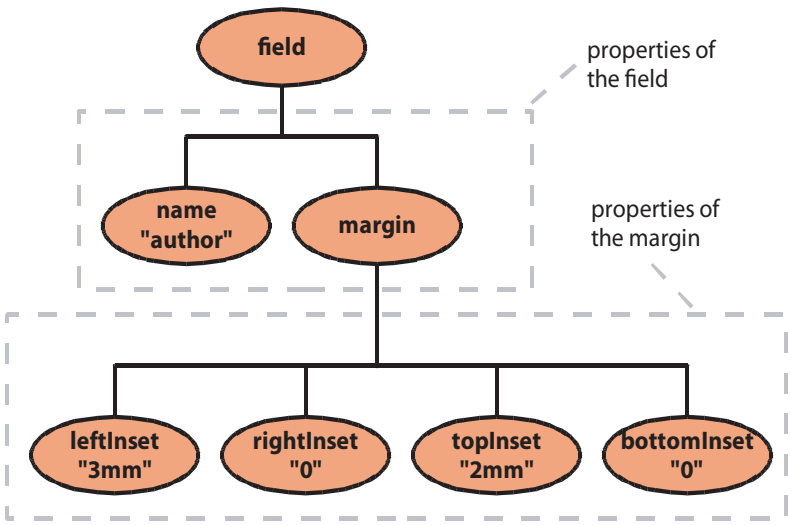
The order of occurrence of different properties within the same enclosing element is not significant. For those few properties that are multiply-occurring, the order of occurrence with respect to each other is significant, but not the order of occurrence with respect to other properties. In the terminology used by the RELAX schema language [RELAX-NG], properties can *interleave*.

The following example shows a fragment from the XML representation of an XFA template.

Example 3.1 Fragment of a template in XML

```
<field name="author" …>
  <margin leftInset="3mm" topInset="2mm"/>
</field>
```

When the fragment is loaded into the Template DOM the result is as follows:



Properties in a DOM

Note that this does not show all of the properties of a field. There are many.

The following XML fragment produces exactly the same logical result as the previous example.

Example 3.2 Order of attributes is not significant

```
<field name="author" …>
  <margin topInset="2mm" leftInset="3mm"/>
</field>
```

In this fragment the order of the `topInset` and `leftInset` properties has been changed but this is not significant and is not *necessarily* reflected in the DOM.

Children

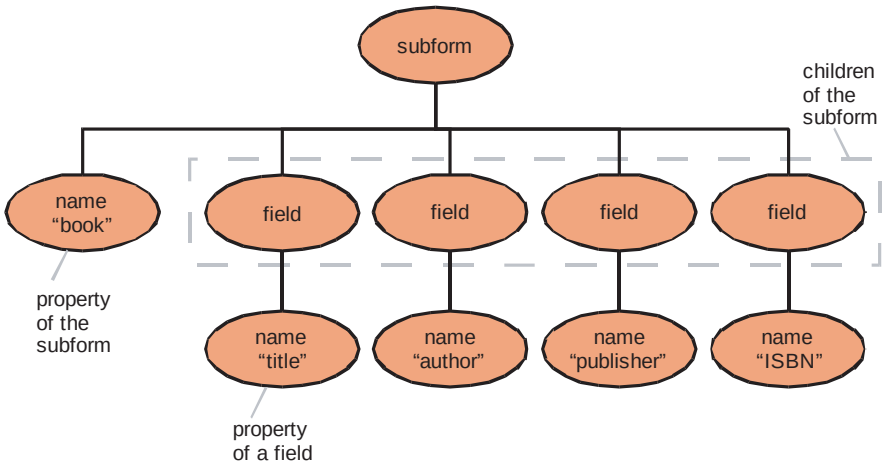
Children are nodes that can occur any number of times (including zero). When expressed in XML child elements can freely interleave with property elements and such interleaving has no significance. However the order of child nodes with respect to each other is significant. When a DOM is expressed in XML document order corresponds to a depth-first traversal of the tree. For child nodes document order corresponds to oldest-to-youngest. By convention oldest-to-youngest is also referred to as left-to-right.

For example, consider the following fragment of XML, again representing part of a template.

Example 3.3 Parent node with children

```
<subform name="book">
  <field name="title" …> … </field>
  <field name="author" …> … </field>
  <field name="publisher" …> … </field>
  <field name="ISBN" …> … </field>
</subform>
```

When the fragment is loaded into the Template DOM the result is as follows:



Children in a DOM

The subform object has five nodes below it, one property node and four child nodes. Each of the child nodes is a field object. There are exactly as many child nodes as are present in the XML. The document

order of the children is exactly as presented in the XML.

Note that a similar hierarchy is used in XDP documents to represent a collection of DOMs in a single document. However an XDP may contain subtrees that are not represented by a DOM (for example application-specific data). At the same time some DOMs (such as the Form and Layout DOMs) are never represented in an XDP. For this reason in an XDP the outermost element tag is `xdp` rather than `xfa`.

The DOMs Used in XFA

This section introduces the individual DOMs and briefly describes each.

The Configuration DOM

This DOM holds option settings that modify the behavior of all the other DOMs. It may also hold private (custom) options.

Because it affects so many other things the Configuration DOM is the first one loaded by an XFA processor. If a script modifies a value in the Configuration DOM there is often no immediate effect because the value applies to processing which has already occurred.

A major part of the content of the Configuration DOM is a set of data transformation options used by the XFA Data DOM. It is common to include the Configuration DOM in an XDP in order to ensure that necessary transformations are carried out.

The Connection Set DOM

This DOM holds information concerning web services using WSDL, such as the URL at which the service is located.

The Connection Set DOM names each transaction it describes. The transaction names are used to associate the transactions with events that activate the transactions. Hence this DOM is mandatory if the form uses web services. It is otherwise optional.

This DOM is commonly loaded from and/or written to an XDP.

The Connection Set Data DOM

This DOM is a temporary buffer holding data going to or coming from a WSDL service, or data going to an HTTP server for a POST operation.

When used with WSDL this DOM is loaded from and/or written to XML documents, but those documents are transient messages exchanged with the web service. This DOM is never loaded from or written out to an XDP or other persistent document.

The Data Description DOM

This DOM holds a schema for the data (as stored in the XFA Data DOM), and/or documents to be exchanged with web services.

The data description is mandatory if web services are being used, because it supplies the structure of the documents to be exchanged with the web service. Otherwise it is optional.

Even when present and describing data, its effects are limited. When loading data, XFA does not check whether data conforms to the data description. Once loaded, scripts may modify the data in ways that are not consistent with the data description. The data description only affects the data binding operation. If a data description is present and the data conforms to it then the XFA processor ensures that data inserted by the data binding operation (see "Interaction of the DOMs" on page 70) is also in conformance. Without the data description, the XFA processor would insert data that conformed to the structure of the template. Thus the data description enhances the independence of data and template.

The Form DOM

This DOM holds the result of merging the data with the template. All nodes in the Form DOM are tied to nodes representing form entities (such as fields or boilerplate) in the Template DOM. Some nodes in the Form DOM are simultaneously tied to nodes representing data in the XFA Data DOM. Operations that modify the value of a node in the Form DOM are passed through to the corresponding node in the XFA Data DOM and vice-versa.

The Form DOM is not necessarily loaded from or written to XML. However some XFA applications may save a copy of all or part of the Form DOM in order to preserve the context of a session. This allows the application to preserve:

- data values that are not bound to any field.
- the fact that the user has overridden the calculation of a field, so it should not be calculated upon reloading.
- modifications made to the Form DOM by scripts, for example changing the border color of a field.

This specification does not define a syntax for saving the Form DOM in XML. One could save it using the same schema used for XML representations of the Template DOM, however this would unnecessarily duplicate a lot of information from the Template DOM.

The Form DOM is the natural territory of scripts because it is where logical entities assume values. For example, when the user tabs into a field, the resulting field enter event is associated with a field object in the Form DOM, not the Template DOM. When the script runs, the `"$"` or `this` variable points to the field object in the Form DOM.

The Layout DOM

This internal DOM holds the result of laying out the form, including data, onto a page or pages. Each node in the Layout DOM represents the placing of an object, or portion of an object, from the Form DOM into a particular position on a particular page.

The Layout DOM is neither loaded from nor written to XML. It can always be reconstructed from the template and the Form DOM so there is no need for it to persist.

The Layout DOM connects to the UI in interactive contexts but that connection is not defined in this specification. except in very general terms. The Layout DOM also connects to the printer driver in non-interactive contexts.

The Locale Set DOM

A locale is a cultural context (usually language and country). The Locale Set DOM provides resources for each locale. For example, it provides the currency symbol and the month and day names. These are grouped by locale so that the form can automatically adjust to the locale in which it is used.

The Source Set DOM

This DOM holds information about the connection between a form and external data sources and/or sinks. It is only used for connections employing ActiveX® Data Objects (ADO). This interface is record-oriented and is usually used for connections to data bases within a corporate LAN or WAN.

The Template DOM

This DOM holds the fixed components of the form. It controls the structure and organization of the form. It supplies all the boilerplate. And it contains all the form's intelligence, including scripts, defaults, calculations, and validations.

The Template DOM is commonly loaded from and written to XML. No useful XFA processing can take place without a template.

The XFA DOM

As described in "Hierarchy of DOMs" on page 63, the XFA DOM is a wrapper for the other DOMs. In addition it has a few nodes of its own, for example the `host` object (`$host`) which holds a number of convenience methods for scripts.

The XFA DOM is not directly loaded from or written to XML, but some of the DOMs within it are. Within an XFA application the XFA DOM always exists, even when none of the other DOMs do (for example, at the start of processing).

The XFA Data DOM

The XFA Data DOM presents an abstract view of the XML data document. In this abstract view the document contents are represented by two types of nodes, data group nodes and data value nodes. Data group nodes represent grouping elements. Data value nodes represent non-grouping elements and attributes.

There are property nodes in the XFA Data DOM. However usually these do not represent elements or attributes in the XML data document. Rather they hold properties associated with an element or attribute. For example, each node has a `name` property which holds the start tag of the element or the name of the attribute. Similarly there is `namespace` property which holds the full namespace of the element or attribute, regardless of whether it was inherited or declared explicitly. Sometimes a property node does correspond to a particular attribute but in such cases the attribute is in a reserved namespace and is treated as out-of-band information. For example, the `xsi:nil` attribute defined by [XML-Schema] in the namespace `http://www.w3.org/2001/XMLSchema-instance` is *not* represented in the XFA Data DOM as a data value node. Instead it modifies the `isNull` property of the data value node corresponding to the element which holds it.

There are a large number of options available to transform the data on its way in from the XML Data DOM to the XFA Data DOM. In most cases the reverse transform is applied on the way out. Hence scripts which operate on the XFA Data DOM can be isolated from details of the XML representation. For example, it is possible to rename particular elements or attributes on the way in. When the data is subsequently written out to a new XML document the original element tags appear in the new document.

The XFA Data DOM is not directly loaded from or written to XML. However its content is echoed in the XML Data DOM which is always loaded from and/or written to XML. Unlike most other DOMs, the XFA Data DOM may operate in record mode. In this mode only global data plus a window of record data is resident in memory at any moment.

Note: Throughout this specification when the term Data DOM is used without specifying the XML Data DOM or the XFA Data DOM, the XFA Data DOM is implied.

The XML Data DOM

The XML Data DOM is a standard XML DOM that is used to hold the content of the XML data document. In accordance with the W3C XML DOM specification [XMLDOM2] it has different types of nodes for elements, attributes, and content. It does not distinguish properties from children nor does it ignore whitespace in grouping elements. XFA processors do not in general operate directly on the XML Data DOM. Instead they operate on the XFA Data DOM which presents and manipulates the same data in a more abstract way.

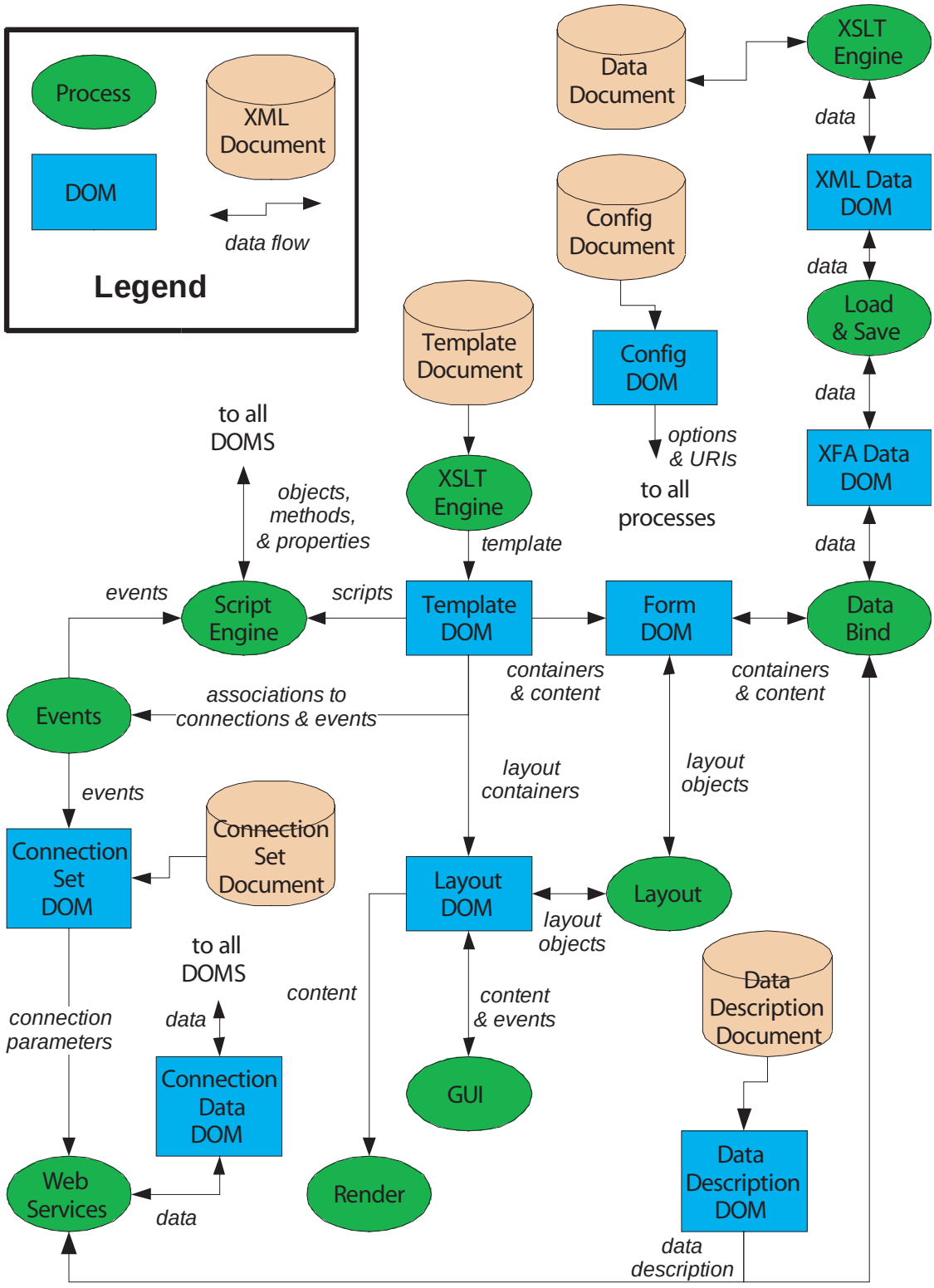
The XML Data DOM is always loaded from and/or written to XML. It has no other purpose. Unlike most other DOMs, the XML Data DOM may operate in record mode. In this mode only global data plus a window of record data is resident in memory at any moment.

Note: Throughout this specification when the term Data DOM is used without specifying the XML Data DOM or the XFA Data DOM, the XFA Data DOM is implied, *not* the XML Data DOM.

Interaction of the DOMs

Much of this specification is detailed description of how the DOMs interact. This section gives an overview of those interactions.

The following diagram depicts the interactions schematically:



Interactions Between the DOMs

Templating versus Binding

There are two general ways in which XFA DOMs interact, through templating and through binding. *Templating* means using one DOM as a pattern in the construction of another DOM. For example, the merge process uses the XFA Template DOM as a template for constructing the XFA Form DOM. *Binding* means tying elements of two DOMs together. The merge process binds individual data values from the XFA Data DOM to nodes representing fields in the XFA Form DOM.

Binding results in a tighter connection than templating. Templating is a one-time process. If the XFA Template DOM is modified after the XFA Form DOM has already been constructed, the template modification has no effect on the XFA Form DOM. By contrast if a data value in the XFA Data DOM is modified after it has been bound to the `value` property of a field in the XFA Form DOM, the change propagates into the field node automatically. Similarly if the data value is updated the change propagates automatically the other direction into the field node's `value` property.

Either templating or binding can be a one-to-one or one-to-many proposition. When binding is one-to-many there is one data value node bound to multiple form nodes. In this case a change in any one of the bound form nodes propagates to the data node and from there to all the other bound form nodes. This allows for global fields that reappear on different pages of the form, with edits in any instance propagating to all other instances.

XML Data DOM and XFA Data DOM

The relationship between the XML Data DOM and the XFA Data DOM was alluded to earlier. Data is loaded from the XML data document into the XML Data DOM. From there the data flows to the XFA Data DOM. The XFA Data DOM provides a view of the data that has much of the XML-related detail abstracted away. When changes are made to data in the XFA Data DOM, the changes are passed back to the XML Data DOM to keep the two views of the data synchronized. When the data is unloaded (saved) it is written from the XML Data DOM to a new XML data document.

During loading the data may be transformed in accordance with settings in the Configuration DOM. Most of the transformations affect only the XFA Data DOM, not the XML Data DOM, so that they affect how the data is viewed internally but not how it is represented in XML when it is unloaded. However some of the transformations affect the XML Data DOM and therefore alter the representation of the data in XML when it is unloaded.

The XML Data DOM and the XFA Data DOM may be loaded and unloaded all at once or a record at a time. When operating in record mode there is a distinction between record data and global data. Record data is the data within a record. Global data is data that is outside all records. Record data is loaded into and purged from the two data DOMs in synchronization. By contrast global data is handled differently in the two data DOMs. Global data is loaded into the XML Data DOM as it is encountered in the XML data document and purged when that part of the document is past. But, once loaded, global data is retained in the XFA Data DOM for the life of the DOM.

The processes of loading data into the Data DOMs and saving data out from the Data DOMs are described in "Creating, Updating, and Unloading a Basic XFA Data DOM" on page 108.

Template DOM, XFA Data DOM, and Form DOM

The Form DOM is the place where the data from the XFA Data DOM is bound to logical structure from the Template DOM. The result is objects copied from the Template DOM into the Form DOM, with some objects in the Form DOM bound to data in the XFA Data DOM. Note that objects in the Form DOM do not have assigned physical locations on the page.

For static forms, similar to a traditional pre-printed paper form, the structure is entirely supplied by the Template DOM. In this case the Form DOM is a duplicate of the subtree under the root subform in the Template DOM, except that some objects in the Form DOM are bound to nodes in the Data DOM.

For dynamic forms the form structure varies in response to the data. For example a subform and its fields can be copied into the Form DOM once for each record of data. In this case the objects in the Form DOM are still copied from the Template DOM but the number of copies and/or the arrangement of objects is dictated by the data.

The degree and nature of dynamisms is controlled at the level of individual objects in the Template DOM. Hence a form can be partly static and partly dynamic. For example, a subform may be included in the Form DOM conditionally upon the presence of a particular data item, yet the conditional subform may itself have a fixed appearance and content.

Data binding is controlled by properties of the objects in the Template DOM and by the names and hierarchy of the objects in both the Template DOM and the XFA Data DOM. The process of binding data to logical structure is described in "Basic Data Binding to Produce the XFA Form DOM" on page 155.

Template DOM, Form DOM, and Layout DOM

The Layout DOM is the place where objects from the Form DOM, or parts of objects, are placed upon one or more pages. The result is objects copied from the Template DOM and the Form DOM to the Layout DOM, with each object in the Layout DOM assigned a place upon a particular page.

The Template DOM supplies objects representing sets of pages, pages, and regions of pages. These are copied into the highest levels of the Layout DOM hierarchy. Objects from the Form DOM are copied into the lower levels, in the place they occupy when displayed or printed. An object from the Form DOM can split across multiple locations (for example, text can flow from one column to the next). Hence a single object in the Form DOM may be copied multiple times into the Layout DOM, with each copy in the Layout DOM representing a different fraction of its content.

Objects representing sets of pages, pages, and regions of pages may be allowed to repeat and/or vary in number. In this way the physical representation of the form can vary with the data.

The layout process can automatically insert headers and footers, leaders and trailers. When these are inserted they are copied from subform objects in the Template DOM.

In interactive applications the GUI is downstream from the Layout DOM. However the GUI also emits data (keyed in) and events (such as mouse clicks). This data and these events are passed upstream by the Layout DOM to the Form DOM. When data is entered it is forwarded to the appropriate field or exclusion group in the Form DOM and updates the data there. When an GUI event causes a script to be activated the script's `"$"` or `this` object is set to the corresponding object in the Form DOM. Thus the Layout DOM is transparent to scripts and user actions.

Layout operations are controlled by properties of the objects being laid out, properties which are copied from the Form DOM but in turn originate from the Template DOM. The process of laying out the form upon one or more pages is described in "Layout for Dynamic Forms" on page 310.

Scripting Object Model

This section explains the conventions for referencing the properties and methods in the object models used by an XFA processing application.

About SOM Expressions

The XFA Scripting Object Model (SOM) is a model for referencing values, properties and methods within a particular Document Object Model (DOM). A DOM structures objects and properties as a tree hierarchy. XFA SOM expressions provide easy access to these objects and properties through a straightforward object reference syntax. This syntax is described in detail later in this specification.

XFA SOM, in combination with a scripting environment, allows individuals to quickly and easily perform a variety of functions without requiring extensive coding. Through the use of various notations, accessors, and operating rules, XFA SOM defines ways to reference specific objects, groups of objects, or even objects whose name is unknown but whose position within the tree is known.

XFA SOM interacts with any XFA-DOM, and may appear in a form template, XML data, configuration information, or on the command line. It is the responsibility of a scripting environment to expose the appropriate objects and properties to XFA SOM. As such, referencing unexposed objects or properties is not possible.

The SOM examples used throughout this document (*XFA Specification*) reflect the properties and methods described in *Adobe XML Form Object Model Reference* [FOM]. It is recommended that XFA processing applications adopt the same names and behaviors as those described in that reference, to ensure a consistent behavior in forms used across a variety of XFA processing applications.

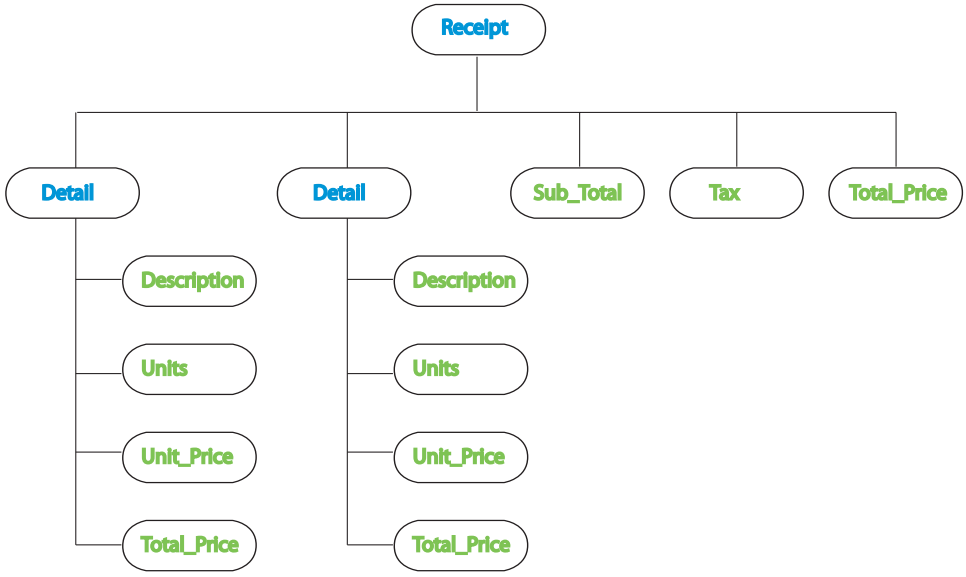
The Receipt Example

An XFA-DOM is structurally represented as a tree hierarchy with a single root object (or node) having a potentially unlimited number of descendant objects (or nodes). For example, the data for a receipt has the following form when expressed as an XML document.

Example 3.4 Data document for receipt example

```
<?xml version="1.0" encoding="UTF-8" ?>
<Receipt>
   <Detail>
      <Description>Giant Slingshot</Description>
      <Units>1</Units>
      <Unit_Price>250.00</Unit_Price>
      <Total_Price>250.00</Total_Price>
   </Detail>
   <Detail>
      <Description>Road Runner Bait, large bag</Description>
      <Units>5</Units>
      <Unit_Price>12.00</Unit_Price>
      <Total_Price>60.00</Total_Price>
   </Detail>
   <Sub_Total>310.00</Sub_Total>
   <Tax>24.80</Tax>
   <Total_Price>334.80</Total_Price>
</Receipt>
```

The following diagram shows the tree for the receipt data as it is stored in the XFA Data DOM. Although the distinction is not important here, data group nodes are shown in blue while data value nodes are shown in green.



Receipt Form Tree

It may be helpful to see the outline of a template that could be used with the receipt data shown above. This is not the only possible template but it shows the basic features.

Example 3.5 Skeleton template for receipt example

```
<xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/">
   <template xmlns="http://www.xfa.org/schema/xfa-template/2.5/">
      <subform name="Receipt" …>
      <pageSet name="ReceiptPageSet" …> … </pageSet>
      <subform name="Detail" …>
         <field name="Description" …> … </field>
         <field name="Units" …> … </field>
         <field name="Unit_Price" …> … </field>
         <field name="Total_Price" …> … </field>
      </subform>
      <subform name="Detail" …>
         <field name="Description" …> … </field>
         <field name="Units" …> … </field>
         <field name="Unit_Price" …> … </field>
         <field name="Total_Price" …> … </field>
      </subform>
         <field name="Sub_Total" …> … </field>
         <field name="Tax" …> … </field>
         <field name="Total_Price" …> … </field>
         …
      </subform>
   </template>
</xdp:xdp>
```

SOM Conventions

One use for XFA SOM expressions is to specify an explicit binding between a field in the template and a node in the Data DOM.

Example 3.6 Field using an XFA SOM expression to explicitly bind to data

```
<field name="Final_Price" …>
   <bind match="dataRef" ref="$data.Receipt.Total_Price"/>
</field>
```

The expression `$data.Receipt.Total_Price` refers to a single node by naming the nodes which must be traversed from the root of the Data DOM down to the desired node. Hence it refers to the `Total_Price` node which corresponds to the last `Total_Price` element in the receipt document. (This is the node containing the value 334.80.) The result of this data reference is to force the XFA application to associate the template field named `Final_Price` with that particular data node, even though the template and data nodes have different names.

XFA SOM expressions may also be used in scripting to reference nodes in an XFA-DOM. For example, this FormCalc expression contains an XFA SOM expression (highlighted in **bold**):

```
Sum(Detail[*].Total_Price)
```

This expression takes advantage of `"[*]"` notation, which is described below under "Selecting All Child Nodes" on page 87, and scoping, which is described in "Relative References". For now it is sufficient to understand that the expression `Detail[*].Total_Price` resolves as a list of all of the `Total_Price` data within `Detail` data groups. With the data given above this becomes 250.00 60.00. The FormCalc function `sum()` simply adds the list of numbers passed to it, yielding in this case 310.00. This expression would be embedded in the template of an intelligent form that added up the totals and taxes automatically, rather than relying on the data file to supply them pre-calculated. For the receipt example, the data file would be as follows.

Example 3.7 Data document for receipt template that performs calculations

```
<?xml version="1.0" encoding="UTF-8" ?>
<Receipt>
   <Detail>
      <Description>Giant Slingshot</Description>
      <Units>1</Units>
      <Unit_Price>250.00</Unit_Price>
   </Detail>
   <Detail>
      <Description>Road Runner Bait, large bag</Description>
      <Units>5</Units>
      <Unit_Price>12.00</Unit_Price>
   </Detail>
</Receipt>
```

The following template uses XFA SOM expressions to perform the calculations automatically. XFA SOM expressions embedded in the template are highlighted in **bold**.

Example 3.8 Receipt template that performs calculations

```
<xdp:xdp …>
   <template …>
      <subform name="Receipt" …>
         <pageSet name="ReceiptPageSet" …> … </pageSet>
         <subform name="Detail" …>
            <field name="Description" …> … </field>
            <field name="Units" …> … </field>
            <field name="Unit_Price" …> … </field>
```

```
            <field name="Total_Price" …>
               <calculate>
                  <script>Units * Unit_Price</script>
               </calculate>
               …
            </field>
         </subform>
         <subform name="Detail" …>
            <field name="Description" …> … </field>
            <field name="Units" …> … </field>
            <field name="Unit_Price" …> … </field>
            <field name="Total_Price" …>
               <calculate>
                  <script>Units * Unit_Price</script>
               </calculate>
               …
            </field>
         </subform>
         <field name="Sub_Total" …>
            <calculate>
               <script>Sum(Detail[*].Total_Price)</script>
            </calculate>
            …
         </field>
         <field name="Tax" …>
            <calculate>
               <script>Sub_Total * .08</script>
            </calculate>
            …
         </field>
         <field name="Total_Price" …>
            <calculate>
               <script>Sub_Total + Tax</script>
            </calculate>
            …
         </field>
         …
      </subform>
   </template>
</xdp:xdp>
```

Basic Object References

XFA SOM expressions provide the means to reference objects within a DOM.

Compound Object Names

Compound object names are a way of navigating down through the hierarchy of objects; each level of the hierarchy is represented by a name and the names are separated by dot (`"."`) characters. The simplest XFA SOM expressions begin with the name of the root node (highest object in the hierarchy) which is named `xfa`. To reference an object within `xfa`, add a dot (`"."`) to the right of `"xfa"` and then append the name of the node you want. Repeat to whatever depth is required.

The template is placed in the hierarchy under the node `xfa.template`. For example, in the receipt template the Tax field is identified in SOM expressions as:

```
xfa.template.Receipt.Tax
```

The data that fills the form is placed in the hierarchy under the node `xfa.datasets.data`. For example, using the receipt example, the node corresponding to the sub-total is identified in SOM as:

```
xfa.datasets.data.Receipt.Sub_Total
```

While the node corresponding to the grand total at the end of the document is identified as:

```
xfa.datasets.data.Receipt.Total_Price
```

Note: As usual when data is expressed in XML, case is significant. The following expressions do *not* match the sub-total node in the receipt example, because the **bold** letters are in the wrong case:

```
xfa.datasets.data.receipt.sub_total
Xfa.datasets.Data.Receipt.Sub_Total
xfa.datasets.data.Receipt.Sub_total
```

Shortcuts

It would be tedious typing in `xfa.datasets.data` over and over again. For convenience a set of predefined shortcuts is available. The complete list is described in the following table:

Short and long forms	***Example 3.9 Short form examples***	Comments
`$data` `xfa.datasets.data`	`$data.Receipt.Tax`	Data that fills the form (Data DOM)
`$template` `xfa.template`	`$template.Receipt.layout`	Template for the form (Template DOM)
`$connectionSet` `xfa.connectionSet`	`$connectionSet.ShoppingCart.` `soapAction`	Schema(s) or interfaces to host(s) (Connection Set DOM)
`$form` `xfa.form`	`$form.Receipt.Tax`	Joined template and data after a merge operation (Form DOM)
`$layout` `xfa.layout`	`$layout.ready`	Methods and properties belonging to the layout process (pseudo-DOM)
`$host` `xfa.host`	`$host.setFocus` `("$form.Receipt.Tax")`	Methods and properties that do not belong anywhere else (pseudo-DOM)
`$record` varies (see note below)	`$record.Tax`	Current data record (subtree within the Data DOM)
`$dataWindow` `xfa.dataWindow`	`$dataWindow.isRecordGroup(` `ref(xfa.datasets.data))`	Object controlling loading and unloading of data records (pseudo-DOM)
`$event` `xfa.event`	`$event.name`	Properties of the current event (pseudo-DOM)

Short and long forms	***Example 3.9 Short form examples***	Comments
`!` `xfa.datasets.`	`!data.Receipt.Tax`	Does not require a `"."` before the next name in the expression
`$xfa` `xfa`	`$xfa.resolveNode("Tax")`	Not really shorter but provided for symmetry

More about $record

The meaning of `$record` varies depending whether record processing is enabled or not:

- *Record processing enabled*. If record processing is enabled, only a portion of the data is loaded into memory at any one time. This portion is a window containing several consecutive records. Each record is a subtree of the data corresponding to one element and its contents. In this mode of operation `$record` points to the node representing the outer element for the current record. In the receipt example `$record` would initially be set to point to the node representing the first `Detail` element. After some processing `$record` would advance to the node representing the next `Detail` element. The receipt example contains only two records, but large documents may contain thousands or millions of records.
- *Record processing not enabled*. In non-record mode `$record` points to the node representing the outermost element of the data document, that is the node which is the only child of `$data`. In the receipt example `$record` would be set to `$data.Receipt`. Hence in non-record mode the entire document is treated as one big record.

See the "Creating, Updating, and Unloading a Basic XFA Data DOM" on page 108 for more information on record processing.

Repeated Elements

When multiple nodes with the same name occur as children of the same parent node, a reference to the shared name is taken to refer to the first matching child, in document order. (In tree diagrams, document order corresponds to starting at the root node and making a depth-first left-to-right traversal of the tree.) The receipt example includes two sets of data describing purchased items, each in a `Detail` element. The following expression refers only to the node representing the first `Detail` element in document order (that is, the one for a giant sling shot):

```
$data.Receipt.Detail
```

To access the other `Detail` nodes, given that they have the same name, it is necessary to use an array-subscript notation. The syntax [*nnn*], where *nnn* represents a number, is used to select one particular element out of a group of siblings with the same names. The number zero represents the first sibling. Hence the following two expressions are equivalent:

```
$data.Receipt.Detail $data.Receipt.Detail[0]
```

The next `Detail` node is referenced as

```
$data.Receipt.Detail[1]
```

Note: It would not make any difference if there had been other nodes in between, as long as they were not named `Detail`. For example, the data document could be changed as follows.

Example 3.10 Receipt data document with additional interleaved data

```
<?xml version="1.0" encoding="UTF-8" ?>
```

```
<Receipt>
   <Page>1</Page>
   <Detail>
      <Description>Giant Slingshot</Description>
      <Units>1</Units>
      <Unit_Price>250.00</Unit_Price>
      <Total_Price>250.00</Total_Price>
   </Detail>
   <Page>2</Page>
   <Detail>
      <Description>Road Runner Bait, large bag</Description>
      <Units>5</Units>
      <Unit_Price>12.00</Unit_Price>
      <Total_Price>60.00</Total_Price>
   </Detail>
   <Sub_Total>310.00</Sub_Total>
   <Tax>24.80</Tax>
   <Total_Price>334.80</Total_Price>
</Receipt>
```

Even with this change to the data document, no change is required to the SOM expression referencing either Detail node. This is an important feature of SOM expressions; they are not invalidated by the insertion or removal of other nodes with different names and hence presumably containing unrelated information. Readers familiar with the RELAX NG schema language will recognize this as equivalent to saying that XFA SOM supports *interleaved* elements.

XFA does not impose any built-in limit to how many sibling nodes can share the same name.

Explicitly Named Objects

In XFA SOM, an explicitly nameable object represents an element that takes a `name` attribute, rather than relying on the element tag to supply it with a name. The following example shows an XDP file containing a connection set which contains the named objects `ShoppingCart`, `Catalogue`, `Shipping` and `TsAndCs`.

Example 3.11 XDP containing a connection set

```
<xdp:xdp …>
   <connectionSet xmlns="http://www.xfa.org/schema/xfa-connection-set/2.4/">
      <wsdlConnection name="ShoppingCart" … > … </wsdlConnection>
      <wsdlConnection name="Catalogue" … > … </wsdlConnection>
      <wsdlConnection name="Shipping" … > … </wsdlConnection>
      <xmlConnection name="TsAndCs" … > … </xmlConnection>
   </connectionSet>
</xdp:xdp>
```

The above-mentioned objects can be referenced in SOM expressions as follows:

```
$connectionSet.ShoppingCart
$connectionSet.Cataloge
$connectionSet.Shipping
$connectionSet.TsAndCs
```

Objects are either nameable or not nameable. For nameable objects, the name specified by the `name` attribute is the only name for the object. If the `name` attribute is omitted the object has no name.

The most common reason for XFA objects being nameable, as for the `wsdlConnection` elements here, is to make it easier to pick a particular item out of a list of items. Naming also enhances modularity by separating the SOM expression that refers to an object from the type of the object. Here, if the `xmlConnection` is changed to an `xsdConnection` (because a schema has been published for it), it can still be referenced using the name `TsAndCs`.

Most nameable objects are not required to have unique names. The children of a `connectionSet` are exceptions in that they are required to have unique names. Consult the individual syntax reference for the DOM to determine whether or not names are required to be unique.

Though it is not shown in these examples, the template element can take a `name` attribute. Despite this the template element is not nameable, because it is a top-level packet wrapper. The `name` attribute *in this one case only* merely holds a human-readable description of the template. The template object must always be referenced using `xfa.template` or `$template`.

Transparent Nodes

When an explicitly nameable object is left unnamed, it is invisible to the normal SOM syntax. Such unnamed objects are called *transparent*.

In the following example, the receipt template is changed to omit the `name` attribute from the detail subforms.

Example 3.12 "name" omitted from subforms

```
<xdp:xdp …>
   <template …>
      <subform name="Receipt">
         <subform>
            <field name="Description" …> … </field>
            <field name="Units"> … </field>
            <field name="Unit_Price" …> … </field>
            <field name="Total_Price" …> … </field>
         </subform>
         <subform>
            <field name="Description" …> … </field>
            <field name="Units"> … </field>
            <field name="Unit_Price" …> … </field>
            <field name="Total_Price" …> … </field>
         </subform>
         <field name="Sub_Total" …> … </field>
         <field name="Tax" …> … </field>
         <field name="Total_Price" …> … </field>
         …
      </subform>
   </template>
</xdp:xdp>
```

In the above example, the `Description` field for the first detail subform is referenced in the Form DOM as:

```
$form.Receipt.Description[0]
```

while the `Description` field in the second detail subform is referenced as:

```
$form.Receipt.Description[1]
```

Similarly in the Template DOM the references would be:

```
$template.Receipt.Description[0]
```

and

```
$template.Receipt.Description[1]
```

It is as though the nameless subform was removed and its children adopted by the nameless subform's parent. This has the side-effect of making fields into siblings that would not otherwise be siblings. For the `Total_Price` fields all three become siblings for purposes of SOM expressions even though they are physically at different levels in the tree.

Nameless template and form objects cannot partake in the full set of functions that named objects partake in. Rather, nameless subforms are usually inserted simply to wrap around another object in order to lend the subform's richer capabilities to the enclosed object. For example, fields do not have `occur` properties but subforms do. As a result, it is normal to wrap a field inside a nameless subform in order to place the field under the influence of an `occur` property. Similarly, nameless exclusion groups are mere wrappers around sets of fields; the actual data values belong to the fields, not to the nameless exclusion group. In the same way, all nameless template and form objects are second-class objects, of interest to the form creator but not corresponding to data or to anything visible. The SOM expression resolver makes them transparent because it is convenient to be able to insert or remove such second-class objects without being forced to modify scripts.

Area Objects Are Always Transparent

Within the Template and Form DOMs there may be `area` objects. An area object is an object which groups together other template objects when the form is being created or modified. It has no consequences at run time, either for the server or client. For this reason areas are always transparent to SOM expressions *even if the areas have names*.

Variables Objects Are Always Transparent

Also within the Template and Form DOMs, subform objects may have a child `variables` object. A `variables` object holds document variable objects, each of which is explicitly nameable. (See "Document Variables" on page 325.) The `variables` object itself is transparent, so that each document variable appears in SOM expressions as though it was directly a child of the root subform. For example, the following template defines a document variable called `CompanyName` (shown in **bold**).

Example 3.13 Document variables are transparent

```
<xdp:xdp …>
   <template …>
      <subform name="Receipt">
         <variables>
            <text name="CompanyName">AnyCo</text>
            <float name="TaxRate">0.0725</float>
         </variables>
      </subform>
   </template>
</xdp:xdp>
```

Within a SOM expression the document variable `CompanyName` is referred to using the SOM expression:

```
$template.Receipt.CompanyName
```

The transparency of the `variables` object makes document variables easy to refer to within scripts when using a more advanced type of SOM expression, as explained below in "Relative References" on page 94

Other Transparent Objects

Transparency is limited to nameable objects in the Form and Template DOMs. In other DOMs all objects are opaque, whether named or not.

Note: Version 2.4 and earlier versions of this specification said that `wsdlConnection`, `xsdConnection` and `xmlConnection` in the Connection Set DOM were transparent when unnamed. This statement was an error. A `wsdlConnection` is not useable *unless* it is named. An `xsdConnection` or `xmlConnection` may be useful when unnamed (as a default connection) but even then it is not transparent.

Reference by Class

There is a special syntax which can be used to reference all objects, whether they are transparent or not. The syntax is `"#`*`class`*`"`, where *class* is the name of the object class. In most cases for objects which can be expressed in XML the name of the object class is the same as the tag for the associated element. For example, the second detail subform object in the template on page 81 can be referenced as

```
$template.Receipt.#subform[1]
```

Note: When an index is used with the `"#`*`class`*`"` syntax, the index refers to all occurrences of true siblings of that class, whether they are transparent or not.

Explicit naming is available as an option in the Data DOM. However, in the Data DOM, the element tag is taken as the name by default, but may be overridden by the content of an attribute. Thus, nodes in the Data DOM always have names, one way or the other. See the "XFA Names" on page 62 for a description of the explicit naming option and how to invoke it. Consequently the `"#"` syntax is not usually needed for nodes in the Data DOM. One case in which it is needed is when the element tag contains characters that are not allowed in the names of objects by the scripting language. For example, FormCalc does not support object names containing a minus (`"-"`) character. If such an element is loaded into the Data DOM without mapping the name to something else (another load option), the resulting `dataGroup` or `dataValue` object cannot be referenced using the usual syntax. In such a case, it is necessary to use `#dataGroup` or `#dataValue`, respectively.

The `"#`*`class`*`"` syntax can also be used for objects that cannot be explicitly named, although it is redundant. For example, consider the following configuration information.

Example 3.14 Configuration packet specifying number of copies to print

```
<xdp:xdp …>
   <config …>
      <present>
         <copies>4</copies>
         …
      </present>
   </config>
</xdp:xdp>
```

In this example the SOM expression `$config.present.copies` is equivalent to `$config.present.#copies`.

Attributes

Attributes are accessed using the same syntax as elements. Instead of the element tag/object class use the attribute name.

Example 3.15 Connection set containing an attribute of interest

```
<xdp:xdp …>
   <connectionSet xmlns="http://www.xfa.org/schema/xfa-connection-set/2.4/">
      <wsdlConnection name="ShoppingCart" dataDescription="cartDD">
         …
      </wsdlConnection>
      <wsdlConnection name="Catalogue" … >
      …
      </wsdlConnection>
      <wsdlConnection name="Shipping" … > … </wsdlConnection>
      <xmlConnection name="TsAndCs" … > … </xmlConnection>
   </connectionSet>
</xdp:xdp>
```

In the example the `dataDescription` attribute of the `wsdlConnection` named `ShoppingCart` can be referenced using the SOM expression:

```
$connectionSet.ShoppingCart.dataDescription
```

XML forbids more than one occurrence of a particular attribute per element, so it is never necessary to use array-subscripting when referring to attributes.

The Data DOM does not by default load attributes, but there is an option to load attributes. See "Creating, Updating, and Unloading a Basic XFA Data DOM" on page 108 for more information about loading attributes into the Data DOM.

Using the same syntax for child references and attributes raises the spectre of name clashes. See "Name clashes" on page 85 for more information.

Internal Properties and Methods

Scripting objects may have internal properties and methods that do not correspond to any XML element or attribute. These are known as *transient* objects. For example, the `$event` object is created at run time to hold the properties of whatever event is currently active. It can not be serialized to XML. Properties and methods of such objects are referenced using the same `"."` notation used for attributes and classes. For example, the following template fragment contains a script, activated by a mouse click on the field, that checks a property of `$event` to determine whether the shift key was held down while the mouse button was clicked.

Example 3.16 Script that responds only to shift-clicks

```
<subform name="Receipt">
   <field name="Sub_Total" …>
      <event action="click">
         <script>if ($event.shift) then … endif</script>
      </event>
   </field>
</subform>
```

`$host` is another object that is purely internal. The following template fragment extends the above script using a method of `$host` to set the keyboard focus when the shift-click event occurs.

Example 3.17 Script that sets focus

```
<subform name="Receipt">
```

```
   <field name="Sub_Total" …>
      <event action="click">
         <script>
            if ($event.shift) then
               $host.setFocus("$form.Receipt.Tax")
            endif
         </script>
      </event>
   </field>
   …
</field>
```

Some nodes have properties that may or may not correspond to an XML element or attribute. For example, every subform and field has a locale property. When expressed in XML the corresponding element may not have a locale declaration because it may inherit the locale of its parent subform. It is expected that when an XFA application writes out data as XML it will eliminate redundant locale declarations where possible. Nonetheless, to make scripting easier, every node in the Data DOM presents a locale property. Hence the locale for the Tax element in the receipt example can be referenced as:

```
$form.Receipt.Tax.locale
```

It is beyond the scope of this specification to describe the properties possessed by different nodes in different DOMs. For that information consult the scripting reference and the individual reference for each DOM.

Name clashes

Name clashes can occur between names explicitly assigned via a `name` attribute and names automatically generated from element tags, attributes, or internal properties. The `".#"` syntax can be used to resolve such name clashes.

This example contains name clashes between two attribute names and the names explicitly assigned to child elements.

Example 3.18 Name clashes caused by choice of field names

```
<subform name="Detail" x="7.76mm" y="6.17mm">
   <field name="x" …> … </field>
   <field name="y" …> … </field>
</subform>
```

The expression

```
   $template.Detail.x
```

returns the content of the field named `x`. By contrast the expression

```
   $template.Detail.#x
```

returns the attribute `x` on Detail, which has a value of `7.76mm`.

In the next example, the subform has a `name` attribute which is set to `Detail`. However it also contains a field element which is explicitly named `name`.

Example 3.19 Name clash caused by the name "name"

```
<subform name="Detail">
```

```
   <field name="name">
      <value>
         <text>Ernest</text>
      </value>
   </field>
</subform>
```

The XFA SOM expression

```
$template.Detail.name
```

returns the value of the field named `name`, which is `Ernest`, because XFA SOM resolves the name clash in favor of the explicit naming of the field, rather than the automatic naming of the subform's attribute.

To access the `name` attribute of Detail, use `".#name"`. For example,

```
$template.Detail.#name
```

returns the value of the property `name` on the `Detail` subform which is the string `Detail`.

More usefully, the same trick works with an object pointer. Suppose the script contains a variable `mycontainer` which points a container in the Form DOM. The value of the container's `name` property can reliably be determined using the SOM expression

```
mycontainer.#name
```

whereas the expression

```
mycontainer.name
```

could return a pointer to a node which is a child of `mycontainer` and itself has a `name` property of `name`.

Note that there is no way to disambiguate clashes between attribute names and child element tags or internal properties. XFA schemas, such as the template schema, are constructed in such a way as to prevent such clashes. User data cannot be so constrained. Instead, in the Data DOM the situation is handled by treating attribute values as just another type of content, so that array-subscripting can be used to select the desired node. For example, assume attributes are being loaded into the Data DOM and the data is as follows.

Example 3.20 Modified receipt data yielding a name clash

```
<Receipt Detail="Acme">
   <Detail> … </Detail>
   <Detail> … </Detail>
</Receipt>
```

In the Data DOM this is handled by creating three separate nodes which are siblings. The first node (eldest sibling) represents the Detail attribute with the value Acme. The second node (middle sibling) represents the first Detail element. The third node (youngest sibling) represents the second Detail element. Hence either of the expressions

```
$data.Receipt.Detail
$data.Receipt.Detail[0]
```

resolves to Acme, whereas the expression

```
$data.Receipt.Detail[1]
```

resolves to the node representing the first of the two Detail elements, and the expression

```
$data.Receipt.Detail[2]
```

resolves to the node representing the second Detail element. This behavior is unique to the Data DOM. For more information, see "Exchanging Data Between an External Application and a Basic XFA Form" on page 108.

Selecting All Child Nodes

The syntax ".*" can be used to select all child nodes, regardless of their names, which match the subsequent portions of the expression. Example 3.21 is used to illustrate selection of multiple nodes.

Example 3.21 Data used to illustrate selection of multiple nodes

```
<?xml version="1.0" encoding="UTF-8" ?>
<Receipt>
   <Page>1</Page>
   <Detail PartNo="GS001">
      <Description>Giant Slingshot</Description>
      <Units>1</Units>
      <Unit_Price>250.00</Unit_Price>
      <Total_Price>250.00</Total_Price>
   </Detail>
   <Page>2</Page>
   <Detail PartNo="RRB-LB">
      <Description>Road Runner Bait, large bag</Description>
      <Units>5</Units>
      <Unit_Price>12.00</Unit_Price>
      <Total_Price>60.00</Total_Price>
   </Detail>
   <Sub_Total>310.00</Sub_Total>
   <Tax>24.80</Tax>
   <Total_Price>334.80</Total_Price>
</Receipt>
```

When the above data (Example 3.21) is loaded into the Data DOM, the expression

```
$data.Receipt.*
```

by default yields seven nodes corresponding to all of the elements which are direct children of the `Receipt` element.

With the same data, the expression

```
$data.Receipt.*.Total_Price
```

yields two nodes corresponding to the `Total_Price` elements contained within `Detail` elements. The `Total_Price` element that is a direct child of `Receipt` is excluded because it there is no node in between it and `Receipt`, hence nothing that matches ".*".

Again with the same data, the expression

```
$data.Receipt.Detail[0].*
```

by default yields four nodes corresponding the elements enclosed within the first `Detail` element. The default behavior is that attributes are not loaded into the Data DOM. However if attributes had been loaded there would have been an additional node, representing the `PartNo` attributes on the `Detail`

element, included in the set. See "Basic Data Binding to Produce the XFA Form DOM" on page 155 for more information about loading of attributes into the Data DOM.

Selecting All Sibling Nodes

The syntax `"[*]"` can be used to select all sibling nodes that share a name. For example, given the same data from Example 3.21, the expression

```
$data.Receipt.Detail[*]
```

yields the two `Detail` nodes which are children of the `Receipt` node. The set does not include their sibling `Page`, `Sub_Total`, `Tax`, and `Total_Price` nodes.

Selecting a Subset of Sibling Nodes

XFA-SOM expressions can include *predicates*. A predicate is an expression which, when evaluated, yields either True or False. A predicate is used in a context where more than one node may exist and it is desired to select only a subset. The XFA processor evaluates the predicate in the context of each candidate node in turn and adds the node to the subset only when the predicate yields True.

The syntax `" .[formcalc_expression] "` can be used to select all sibling nodes that match the given expression. The contained expression must yield a Boolean value.

For XFA processors that support ECMAScript, the same set selection can be expressed using the form `" .(ecmascript_expression)"`. This format is compliant with section 11.2.4 of the ECMAScript-357 standard [ECMAScript357]. Again the contained expression must yield a Boolean value.

Note: The language of the expression is determined only by the type of brackets, not by the language of the script hosting the expression. This allows for the use of predicates using either FormCalc or ECMAScript in contexts where there is no hosting script, such as the `ref` sub-property of a field's `bind` property.

For example, given the same data from Example 3.21, the SOM expression

```
$data.Receipt.Detail.(Total_Price.rawValue > 200)
```

yields the the single `Detail` node which its the `Total_Price` property set to `250.00`. An exactly equivalent SOM expression can be written using FormCalc syntax:

```
$data.Receipt.Detail.[Total_Price > 200]
```

By contrast either

```
$data.Receipt.Detail.(Total_Price.rawValue < 200)
```

or

```
$data.Receipt.Detail.[Total_Price < 200]
```

yields the single `Detail` node which has its `Total_Price` property set to `60.00`.

Finally, either

```
$data.Receipt.Detail.(Total_Price.rawValue < 1000)
```

or

```
$data.Receipt.Detail.[Total_Price < 1000]
```

yields both `Detail` nodes.

If an error is encountered while evaluating the expression the error condition is processed immediately without evaluating the expression against any more siblings.

Predicates are often used in the `ref` subproperty of a field's `bind` property. This makes it possible to make a field bind to data conditionally upon the value of the data. Note, however, that once the binding has been made the predicate is no longer consulted. Hence this is not suitable for interactive applications, where the user may change the value, unless a validation script is also provided to force the data to remain within the predicate range. Of course this is not a problem in non-interactive applications such as report generation.

For example, the receipt form could be modified to generate a report that differentiates between items above a certain price (which require a manager's approval) and items at or below that price (which are discretionary). The predicate for items below the threshold price includes a < character, which is written `<` in accordance with the rules of XML.

Example 3.22 Binding controlled by predicates

```
<subform name="Receipt">
   <subform name="Detail_Approval_Reqd">
      <bind ref="$data.Receipt.Detail.[Total_Price > 200]"/>
      <field name="Description" …>…</field>
      <field name="Units" …>…</field>
      <field name="Unit_Price" …>…</field>
      <field name="Total_Price" …>…</field>
   </subform>
   …
   <subform name="Detail_Discretionary">
      <bind ref="$data.Receipt.Detail.[Total_Price &lt;= 200]"/>
      <field name="Description" …>…</field>
      <field name="Units" …>…</field>
      <field name="Unit_Price" …>…</field>
      <field name="Total_Price" …>…</field>
   </subform>
   …
```

You cannot use a predicate directly in the body of a script as part of a normal object reference. Instead you must pass the complete SOM expression as a string to either a `resolveNode()` or `resolveNodes()` method. For example, a script could calculate the sum of items above a certain price as follows.

Example 3.23 Calculation using a predicate

```
<subform name="Receipt">
   <subform name="Detail">
      <field name="Total_Price" …>…</field>
   </subform>
      <subform name="Detail">
      <field name="Total_Price" …>…</field>
   </subform>
   …
   <field name="Big_Items_Total_Price" …>
      <calculate>
         <script>
            sum($.resolveNodes("Detail.[Total_Price > 200]"))
         </script>
```

```
    </calculate>
</field>
```

Caution: Attempting to use a predicate as part of a normal object reference will result in a syntax error. As a rule of thumb, SOM expressions containing predicates must always be within quotation marks.

The Parent Property

Every object except the `xfa` object has a property called parent that points to the object's parent node. When parent is used in a SOM expression, it has the effect of forcing the expression resolution to go back up the tree one level. This is analogous in function to the `"/..."` or `"\..."` constructs often used in file names. However, it should be emphasized the similar syntax `"..."` has quite a different meaning in XFA SOM expressions, as described in "Selecting Descendants At Any Level" on page 90. Instead the function of going up one level is performed by `".parent"`. For example, given the receipt data, the expression

```
$data.Receipt.Detail[1].parent.Tax
```

is equivalent to

```
$data.Receipt.Tax
```

This facility works in any DOM but is much more useful when used with unqualified references in the Form DOM.

The `xfa` object also has a `parent` property, but its value is null.

Selecting Descendants At Any Level

The syntax `"..."` can be used to select the first descendant in document order at any level which matches the subsequent portions of the expression. The resolution of such expression requires additional computation, so their use may adversely impact performance.

To understand this syntax consider the original receipt data. For convenience this data is repeated below.

Example 3.24 Receipt data

```
<?xml version="1.0" encoding="UTF-8" ?>
<Receipt>
   <Detail>
      <Description>Giant Slingshot</Description>
      <Units>1</Units>
      <Unit_Price>250.00</Unit_Price>
      <Total_Price>250.00</Total_Price>
   </Detail>
   <Detail>
      <Description>Road Runner Bait, large bag</Description>
      <Units>5</Units>
      <Unit_Price>12.00</Unit_Price>
      <Total_Price>60.00</Total_Price>
   </Detail>
   <Sub_Total>310.00</Sub_Total>
   <Tax>24.80</Tax>
   <Total_Price>334.80</Total_Price>
</Receipt>
```

In the above example, both of the following expressions

```
$data..Total_Price
$data..Total_Price[0]
```

resolve to `$data.Receipt.Detail[0].Total_Price`, the first matching node in document order. The value of this node is 250.00. Note that once this first match is found, the SOM expression resolver does not look at any other branches of the SOM. In particular, the expression `$data..Total_Price[1]` does not match any node, because there is no node corresponding to `$data.Receipt.Detail[0].Total_Price[1]`.

SOM Expressions That Include Periods and Dashes

An XFA name may include the dash and/or period characters ("XFA Names" on page 62), which have dual meanings in SOM expressions, especially when such expressions are interpreted by a script interpreter, such as FormCalc and ECMAScript. Whether such special characters can be included in a SOM expression depends on whether the expression is expected to be a SOM only or may be either a SOM or a script.

Example 3.25 Object names that include periods and dashes

```
<template>
   <subform>
      <subform name="Toys">
         <field name="My-toy" …/>
         <field name="My.childs-toy" …/>
      </subform>
   </subform>
</template>
```

SOM only

A SOM-only expression is used as the argument for the `resolveNode()` function or method or as the value of the `ref` property of a `bind` element.

- Dash. The dash character is interpreted as part of the field name, rather than as an operator. This is true for both FormCalc and ECMAScript.

Example 3.26 Dash character in a SOM-only expression

```
<script>
   $xfa.resolveNode("$form..My-toy") // '-' included with XFA name
</script>
```

- Period. One or more escape characters must be used to distinguish a period used in a name from a period used as an object separator.

Example 3.27 Period embedded in a name in a SOM-only expression

```
<script>
   // FormCalc environment
   $xfa.resolveNode("form..My\.childs-toy")
</script>
```

or

```
<script contentType="application/x-javascript">
   // ECMAScript environment
   xfa.resolveNode("form..My\\.childs-toy").rawValue
</script>
```

ECMAScript strips one of the escape characters before passing the remaining string to the SOM resolver, but FormCalc does not.

Without the `rawValue` property in the ECMAScript expression, the object `My.childs-toy` would be returned rather than the value of the object, as explained in "Using SOM Expressions in ECMAScript" on page 93

Mixed SOM/script context

A mixed SOM/script context exists when the expression can be interpreted as a SOM expression or as a script expression. Whether the escape sequence `'\'` can be used to prevent the script environment from handling dashes and periods is application dependent. If it is not supported, the `resolveNode()` method/function may be used, as described in "SOM only" on page 91.

Example 3.28 Dot and dash in name are mis-interpreted

```
<calculate>
   <script contentType="application/x-javascript">
      // '.' interpreted as object separator and '-' as subtraction
      xfa.form.subform.My.childs-toy
   </script>
</calculate>
```

Using SOM Expressions in FormCalc

For each script in a template, the script language is encoded in the `script` element by the `contentType` attribute. If this attribute is not specified the language defaults to `application/x-formcalc`, which signifies FormCalc.

SOM expressions are native to FormCalc. Hence, most valid SOM expression can be used anywhere FormCalc requires you to specify a DOM node. The sole exception is a SOM expression using a predicate. SOM expressions using predicates must be passed as strings to a `resolveNode()` or `resolveNodes()` method.

The FormCalc scripting language is, in general, oriented to strings rather than objects. It does not store all variables internally as strings, but it does try to store data in forms that can be converted to and from strings. In keeping with this, when it encounters a SOM expression for a node it looks for a property of that node called `value`. If the property exists, it takes the string value of that property as the resolved value for the SOM expression. Thus for example, given the data for the receipt example, the FormCalc expression

```
$data.Receipt.Detail[1].Units
```

yields not the `Units` node but rather its `value`, which is 5. Similarly the expression

```
sum($data.Receipt.Detail.Total_Price[*])
```

specifies a list of field nodes, but when the `sum()` function processes each node in the list it looks up the `value` property of the node and adds together these numbers.

Runtime resolution of object names

Sometimes you may want to build the name of an object in a string at run time. You can use the `resolveNode()` method of `$xfa` to translate the string into an object reference. For example,

```
$xfa.resolveNode(My_String)
```

There is also a `resolveNodes()` method that returns a list of zero or more object references. For example:

```
sum($xfa.resolveNodes(My_String))
```

All nodes in all XFA DOMs have `resolveNode()` and `resolveNodes()` methods. Furthermore, for the types of SOM expressions described under "Basic Object References" on page 77, the `resolveNode()` or `resolveNodes()` method of any node can be used. However for the advanced expressions described in "Relative References" on page 94 you must use the methods of the `$xfa` object or the `"$"` object.

Using SOM Expressions in ECMAScript

For each script in a template, the script language is encoded in the script element by the `contentType` attribute. If this attribute is specified as `application/x-javascript` it signifies that the language is ECMAScript. ECMAscript was formerly known as JavaScript.

Obtaining the value of an expression

ECMAScript, unlike FormCalc, is not aware when the context calls for a string rather than an object. Hence it never automatically resolves a node reference into a value. To make reference to the data associated with a field or exclusion group in ECMAScript you must explicitly invoke the `rawValue` property of the node. For example, to reference the value of the `Tax` field in the receipt example on page 90, you must use the SOM expression

```
$form.Receipt.Detail.Tax.rawValue
```

Within an XFA processor the same field content may be represented in several different formats. The above expression using `rawValue` returns the field content in canonical format. This is the format that is suitable for calculations. The same data may be presented to the user differently through the lens of a picture clause and/or localization. For more information about the different representations of the data see "Localization and Canonicalization" on page 138.

The above example can be modified to return the formatted value, which is the format seen by the user, as follows:

```
$form.Receipt.Detail.Tax.formattedValue
```

In contrast, when FormCalc evaluates the expression `$data.Receipt.Detail.Tax` it returns the data in canonical form without having to be told to.

Note: Versions of this specification prior to version 2.5 erroneously stated that the `value` property was equivalent to the `rawValue` property. In fact the `value` property corresponds to the `value` element which is a child of `field` and in ECMAScript a SOM expression using this name evaluates to a node.

SOM expressions that use special characters

ECMAScript is rather restrictive in its handling of object names. In particular, expressions for object names may not include any of `"["`, `"]"`, `"*"`, and `". ."`. Consequently many valid SOM expressions cannot be used directly in ECMAScript expressions. For example, the following expressions result in an error when interpreted by ECMAScript:

```
$data..Total_Price // ECMAScript does not support ".."
$data.Receipt.Detail.Total_Price[0] // ECMAScript does not support "[" or "]".
```

To address this limitation, ECMAScript scripts must pass such SOM expressions as strings to a `resolveNode()` method. `resolveNode()` returns the object specified in the SOM expression.

Every node in any of the XFA DOMs has a `resolveNode()` method. Furthermore, for the types of SOM expressions described under "Basic Object References" on page 77, the `resolveNode()` method of any node can be used. However for the advanced expressions described in "Relative References" on page 94 you must use the methods of the `$xfa` object or the current container (which is accessible as the `this` object).

For example, the following line of code is valid:

```
$data.Receipt.Tax.rawValue = 11.34;   // this is valid ECMAScript
```

Whereas the following line of code is *not* valid because the square brackets are not allowed:

```
$data.Receipt.Detail[1].Units.rawValue = 3;  // this is NOT valid ECMAScript
```

Instead you must pass the SOM expression to `resolveNode()`:

```
// A valid ECMAScript expression
$xfa.resolveNode("$data.Receipt.Detail[1].Units").rawValue = 3;
```

Sometimes an operation expects or requires a list of objects, rather than a single object. For these cases the script must use the `resolveNodes()` method instead of the `resolveNode()` method. The `resolveNodes()` method returns a list of zero or more objects,sorted in document order. For example, the following expression creates a variable containing a list of zero or more `dataValues` corresponding to `Units` elements of receipt detail records.

```
// A valid ECMAScript expression
var aList = $xfa.resolveNodes("$data.Receipt.Detail[*].Units");
```

As with the `resolveNode()` method, there is a `resolveNodes()` method on every node in any XFA SOM, but only the methods on `$xfa` and the current container handle relative SOM expressions.

Using SOM Expressions in Bind References

An XFA SOM expression used as a bind reference is not in a scripting context, so it is not FormCalc or any other scripting language. It is evaluated as a raw SOM expression. Any valid SOM expression constructed according to the rules in this section ("Basic Object References" on page 77) can be used, however it must always resolve to a node or list of nodes of the appropriate type in the Data DOM. When the expression resolves to a list of data nodes the XFA processor binds the form object to the first unbound data object in the list. See "Basic Data Binding to Produce the XFA Form DOM" on page 155 for more information about data binding.

Relative References

Whenever a script is activated it resides somewhere in the Form DOM. It originated in the Template DOM, from which it was copied, but the copy in the Template DOM is never activated. Scripts do not reside in any other DOMs. The node containing the script provides a context for the script. Scripts can employ SOM expressions that reference nodes in the Form DOM relative to the node which contains the script. This facility is extended with scoping rules which allow the relative reference to succeed even if it does not exactly match the hierarchy of nodes in the Form DOM.

When data is merged with a template to create the Form DOM, some parts of the Template DOM (including scripts) may be replicated more than once in the Form DOM. This allows a template to dynamically adapt to the number and arrangement of records in the data. But this imposes upon scripting

the requirement that a script be able to work unchanged even when it is not in the same position in the Form DOM that it was originally in the Template DOM. In other words, it must be possible to write scripts that are relocatable. This can be accomplished using relative references and scoping.

The Current Container

Within the Form DOM there is a concept of a container. A container is an object that holds data or values. Simple containers include field (interactive element on the form), draw (static) and contentArea (layout region) elements. All other containers are capable of containing other containers as well as other non-container objects. For more information about containers see "Container Elements" on page 30.

The following objects can directly contain scripts:

- `field`
- `exclGroup`
- `subform`
- `subformSet`

In XFA SOM, the default current object for a script is the container that is the most immediate ancestor of the `script` element. Most often such containers are `field` objects. In addition `exclGroup`, `subform`, and `subformSet` objects can be the current object for scripts. The other containers cannot contain scripts except inside contained `field`, `exclGroup`, `subform`, or `subformSet` objects.

When a SOM expression contains a predicate, the predicate is effectively a small script. Within this script the current container is the node selected by the part of the SOM expression to the left of the predicate.

The current object can be explicitly referenced using the dollar sign, `"$"`. This serves the same purpose as `this` in ECMAScript or `Me` in VBScript. In the following example of an XFA SOM expression embedded in a script, the current object is the `Receipt` subform, the most immediate ancestor that is a container. This script performs a subform-level validation when the user tabs out of the `Receipt` subform. The validation script uses `"$"` to make a relative reference to the value of the `Tax` field, highlighted in bold.

Example 3.29 Script using a reference relative to "$"

```
<xdp:xdp …>
   <template …>
      <subform>
         <subform name="Receipt"…>
            <validate>
               <script>$.Tax >= 0</script>
            </validate>
            <field name="Tax"…> … </field>
            …
         </subform>
         <field …> … </field>
         …
      </subform>
   </template>
</xdp:xdp>
```

In the example above, the full name of the referenced object is `$form.Receipt.Tax` (the root subform is transparent because it is nameless).

For scripts written in ECMAScript, the name of the current container is `this` in native ECMAScript expressions but `"$"` in SOM expressions. The following shows the same validation as Example 3.29, modified to use ECMAScript.

Example 3.30 Script in ECMAScript using "this"

```
<xdp:xdp …>
   <template …>
      <subform>
         <subform name="Receipt"…>
            <validate>
               <script contentType="application/x-javascript">
                  this.Tax.rawValue >= 0  // ECMAScript
               </script>
            </validate>
            <field name="Tax"…> … </field>
            …
         </subform>
         <field …> … </field>
         …
      </subform>
   </template>
</xdp:xdp>
```

In the example the script uses `this` inside a native ECMAScript expression to refer to the current container. Instead of `this` it could have named the current container explicitly, but it must name the correct container! For example, the example could have used the expression:

```
$form.Receipt.Tax.rawValue >= 0  // ECMAScript
```

Or, the script could have used the `resolveNode()` method on the current container or on the `$xfa` object.

Note: The resolveNode() method always uses `"$"`, not `this`, regardless of the scripting language. Hence if this example is changed to use `resolveNode()` on `$xfa` it employs the following syntax:

```
$xfa.resolveNode("$.Tax").rawValue >= 0  // ECMAScript
```

All nodes in all XFA DOMs have `resolveNode()` and `resolveNodes()` methods. Furthermore, for the types of SOM expressions described under "Basic Object References" on page 77, the `resolveNode()` or `resolveNodes()` method of any node can be used. However for relative expressions you must use the methods of either the `$xfa` object or of the script's current container.

Unqualified References to Children of the Container

It is possible to refer directly to a child of the current container by name. In the following example of an XFA SOM expression embedded in a script, the current object is the `Receipt` subform, the most immediate ancestor that is a container. This script uses a relative reference to the value of the `Tax` field, highlighted in bold.

Example 3.31 Script using an unqualified reference to a child of the container

```
<xdp:xdp …>
   <template …>
      <subform name="Receipt"…>
         <field name="Tax"…> … </field>
```

```
        …
        <validate>
           <script>Tax > 0</script>
        </validate>
        …
     </subform>
  </template>
</xdp:xdp>
```

The SOM expression Tax does not start with `"xfa"` or any of the shortcut strings so it is taken to be the name of a child of the current object. The full name of the referenced object is `$form.Receipt.Tax`.

In the example above, the following SOM expressions are equivalent:

```
Tax
$.Tax
$form.Receipt.Tax
$xfa.resolveNode("Tax")
$xfa.resolveNode("$.Tax")
```

The equivalents in ECMAScript are:

```
Tax.rawValue // ECMAScript native expression
this.Tax.rawValue  // ECMAScript native expression
$form.Receipt.Tax.rawValue  // ECMAScript native expression
$xfa.resolveNode("Tax").rawValue  // ECMAScript SOM expression
$xfa.resolveNode("$.Tax").rawValue  // ECMAScript SOM expression
```

Unqualified References to Siblings of the Container

A SOM expression can also refer directly to siblings of its container node. For example, the calculation script for the `$form.Receipt.Total_Price` field can refer to the `Tax` and `Sub_Total` fields, using unqualified names.

Example 3.32 Script using an unqualified reference to a sibling of the container

```
<xdp:xdp …>
  <template …>
     <subform name="Receipt" …>
        …
        <field name="Sub_Total" …> … </field>
        <field name="Tax" …> … </field>
        <field name="Total" …>
           <calculate>
              <script>Sub_Total + Tax</script>
           </calculate>
        </field>
     </subform>
  </template>
</xdp:xdp>
```

The equivalent in ECMAScript is:

```
$xfa.resolveNode("Sub_Total").rawValue +
  $xfa.resolveNode("Tax").rawValue // ECMAScript
```

The ability to refer to siblings with unqualified SOM expressions makes it possible to write relocatable SOM expressions. In the following example the same script is used for calculations in both of the Detail subforms.

Example 3.33 Calculation using a relocatable SOM expression

```
<xdp:xdp …>
   <template …>
      <subform name="Receipt" …>
         <subform name="Detail" …>
            <field name="Description" …> … </field>
            <field name="Units" …> … </field>
            <field name="Unit_Price" …> … </field>
            <field name="Sub_Total" …>
               <calculate>
                  <script>Units * Unit_Price</script>
               </calculate>
            </field>
         </subform>
         <subform name="Detail" …>
            <field name="Description" …> … </field>
            <field name="Units" …> … </field>
            <field name="Unit_Price" …> … </field>
            <field name="Sub_Total" …>
               <calculate>
                  <script>Units * Unit_Price</script>
               </calculate>
            </field>
         </subform>
         …
      </subform>
   </template>
</xdp:xdp>
```

This in turn makes it possible to eliminate the redundant subform declaration in the template. The two subforms can be coalesced into a single subform with an occurrence number of 2. The resulting template is as follows.

Example 3.34 Relocatable calculation allows consolidation of subforms

```
<xdp:xdp …>
   <template …>
      <pageArea> … </pageArea>
```

```
      <subform name="Receipt" layout="tb" …>
         <subform name="Detail" …>
            <occur min="2" max="2" />
            <field name="Description" …> … </field>
            <field name="Units" …> … </field>
            <field name="Unit_Price" …> … </field>
            <field name="Sub_Total" …>
               <calculate>
                  <script>Units * Unit_Price</script>
               </calculate>
            </field>
         </subform>
         …
      </subform>
   </template>
</xdp:xdp>
```

When data is merged into the form, the XFA application automatically incorporates two copies of the `Detail` subform into the Form DOM. See "Dynamic Forms" on page 286 for more information about templates for dynamic forms, and "Basic Data Binding to Produce the XFA Form DOM" on page 155 for more information about how occurrence numbers affect the merge process. The `Receipt` subform uses a top-to-bottom flowing layout strategy so that successive instances of the `Detail` subform are placed into successive content regions. See "Layout for Growable Objects" on page 237 for more information about the layout process for dynamic forms.

Unqualified References to Ancestors of the Container

One more type of unqualified reference is possible. A SOM expression can refer with an unqualified name to an ancestor of its container or to a sibling of an ancestor. This makes it possible to modify a template by wrapping portions of it inside a subform without having to change any of the enclosed scripts. For example, suppose that we are starting with the template from Example 3.33. Later the template is modified as follows.

Example 3.35 Modified template relies on relocatable calculation

```
<xdp:xdp …>
   <template …>
      <subform name="Receipt" …>
         <subform name="Detail" …>
            <field name="Description" …> … </field>
            <field name="Units" …> … </field>
            <field name="Unit_Price" …> … </field>
            <subform name="New_Subform" …>
               <field name="Sub_Total" …>
                  <calculate>
                     <script>Units * Unit_Price</script>
                  </calculate>
               </field>
            </subform>
         </subform>
         …
      </subform>
   </template>
</xdp:xdp>
```

The same script still works because `Units` and `Unit_Price` are both siblings of `New_Subform`, which is an ancestor of `Sub_Total`, which is the container for the script.

Note that this does not work in the other direction. Ancestors can be referred to directly but not descendants beyond immediate children. Starting again with the template from Example 3.33, if a new subform is wrapped around `Units` and `Unit_Price`, it is necessary to modify the script that calculates `Sub_Total` as follows.

Example 3.36 Modification that can not take advantage of relocation

```
<xdp:xdp …>
   <template …>
      <subform name="Receipt" …>
         <subform name="Detail" …>
            <occur min="2" max="2" />
            <field name="Description" …> … </field>
            <subform name="New_Subform" …>
               <field name="Units" …> … </field>
               <field name="Unit_Price" …> … </field>
            </subform>
            <field name="Sub_Total" …>
               <calculate>
                  <script>New_Subform.Units * New_Subform.Unit_Price</script>
               </calculate>
            </field>
         </subform>
         …
      </subform>
   </template>
</xdp:xdp>
```

Differences Between Qualified and Unqualified References

A SOM expression is qualified if the first character is `"$"` or `"!"` or if the first term in the expression is `xfa`. It is also qualified when used to identify an object in ECMAScript and it starts with `this`. Otherwise it is unqualified. Unqualified references search for matching nodes in the following order:

1. Children of the container
2. The container and siblings of the container
3. The parent of the container and siblings of the parent (aunts or uncles) of the container
4. The grandparent of the container and siblings of the grandparent (great-aunts or great-uncles) of the container
5. The above steps repeat recursively up to the root. The unqualified reference fails in either of two cases. It fails if the search reaches the root without finding a match. And it fails if it finds a match for the first term in the expression but fails to find a match for some subsequent term.

"$." Versus Unqualified SOM Expressions

Sometimes because of name conflicts an unqualified SOM expression matches more nodes than you want, or a different node than the one you wanted. In these cases an expression starting with `"$."` may be more suitable. A SOM expression starting with `"$."` is syntactically a qualified expression, yet it is relative to the

script container. Thus it escapes scope-matching without giving up relocation. For example, consider the following template.

Example 3.37 Use of "$" as an alternative to an unqualified SOM expression

```
<xdp:xdp …>
   <template>
      <subform name="Receipt">
         <subform name="Detail">
            <validate>
               <script>$.Total_Price >= 0</script>
            </validate>
            <field name="Total_Price"> … </field>
            …
         </subform>
         <field name="Total_Price"> … </field>
         …
      </subform>
   </template>
</xdp:xdp>
```

the expression `$.Total_Price` resolves unambiguously to `$form.Receipt.Detail.Total_Price`. Scope-matching does not apply hence the expression does *not* resolve to the same-named field `$form.Receipt.Total_Price`.

`"$"` can also be used for expressions pointing to nodes that are higher up in the hierarchy than the script's container. Use `"$.parent"`, `"$.parent.parent"`, and so on to climb levels in the tree. It is possible to climb all the way to the root. The equivalent syntax for native ECMAScript expressions is `"this.parent"`, `"this.parent.parent"`, and so on.

Inferred Index

The previous sections have used as examples a template that is divided up into individual subforms for each detail record. Conceptually such a template is arranged in a tree structure. However it is also possible to create templates that are notionally arranged in a matrix, like a spreadsheet. For example, consider the following receipt template with room for multiple items.

Example 3.38 Receipt template using spreadsheet-like organization

```
<xdp:xdp …>
   <template …>
      <subform name="Receipt" …>
         <field name="Description" …> … </field>
         <field name="Units" …> … </field>
         <field name="Unit_Price" …> … </field>
         <field name="Total_Price" …> … </field>
         <field name="Description" …> … </field>
         <field name="Units" …> … </field>
         <field name="Unit_Price" …> … </field>
         <field name="Total_Price" …> … </field>
         <field name="Description" …> … </field>
         <field name="Units" …> … </field>
         <field name="Unit_Price" …> … </field>
         <field name="Total_Price" …> … </field>
         <field name="Description" …> … </field>
```

```
        <field name="Units" …> … </field>
        <field name="Unit_Price" …> … </field>
        <field name="Total_Price" …> … </field>
        <field name="Description" …> … </field>
        <field name="Units" …> … </field>
        <field name="Unit_Price" …> … </field>
        <field name="Total_Price" …> … </field>
      </subform>
   </template>
</xdp:xdp>
```

Instead of grouping the fields by subform, this static template simply repeats each of the `Description`, `Units`, `Unit_Price`, and `Total_Price` fields five times. Most likely these are arranged on the page as a matrix four fields wide and five lines high, in imitation of a traditional pre-printed paper form.

SOM expressions provide a mechanism to deal conveniently with such arrangements. When scope-matching, if an unqualified reference is made without specifying an index, the index of the container is also used for the unqualified reference. For example, the above template can be modified by adding scripts as follows.

Example 3.39 Using relocatable calculations in a spreadsheet-like organization

```
<xdp:xdp …>
   <template …>
      <subform name="Receipt" …>
        <field name="Description" …> … </field>
        <field name="Units" …> … </field>
        <field name="Unit_Price" …> … </field>
        <field name="Total_Price" …>
          <calculate>
            <script>Units * Unit_Price</script>
          </calculate>
        </field>
        <field name="Description" …> … </field>
        <field name="Units" …> … </field>
        <field name="Unit_Price" …> … </field>
        <field name="Total_Price" …>
          <calculate>
            <script>Units * Unit_Price</script>
          </calculate>
        </field>
        …
      </subform>
   </template>
</xdp:xdp>
```

When each script is activated, the index used for `Units` and for `Unit_Price` are inferred from the `Total_Price` that contains the script. Therefore `Total_Price[0]` is calculated as `Units[0] * Unit_Price[0]`, `Total_Price[1]` is calculated as `Units[1] * Unit_Prices[1]`, and so on. This way the same script can be replicated in different cells without having to edit it for each cell.

To take advantage of inferred indexing in ECMAScript you must use the `resolveNodes()` method. The equivalent of the above scripts in ECMAScript is:

```
$xfa.resolveNode("Units").rawValue *
   $xfa.resolveNode("Unit_Price").rawValue       // ECMAScript
```

It is possible to design a form where the size of the array of referencing fields is not the same as the size of the array of referenced fields. In such a case matching is still attempted by index number. So, if the reference falls within the range of referenced fields, a match is found. If it falls outside, it is an error. For example, if three of the `Units` fields were deleted from the above example, so that it had five `Total_Price` fields but only two `Units` fields, the calculations for `Total_Price[2]`, `Total_Price[3]`, and `Total_Price[4]` would fail. The same calculations fail regardless of which three of the `Units` fields were deleted, because SOM expression indexes refer to occurrence count rather than position on the page. It is generally not a good idea to use this sort of construction unless the fields, subforms, and/or exclusion groups involved form congruent arrays.

There is one exception to this rule. If a script in a container with multiple same-named siblings makes reference to a singly-occurring node with no explicit occurrence indication, that single occurrence is always found. For example, all instances of the `Total_Price` field here refer to a singly-occurring `Discount` field.

Example 3.40 Singly-occuring node is found by an unqualified reference

```
<xdp:xdp …>
   <template …>
      <subform name="Receipt" …>
         <field name="Discount"…> … </field>
         <field name="Description" …> … </field>
         <field name="Units" …> … </field>
         <field name="Unit_Price" …> … </field>
         <field name="Total_Price" …>
            <calculate>
               <script>(Units * Unit_Price) * (1 - (Discount/100.0))</script>
            </calculate>
         </field>
         <field name="Description" …> … </field>
         <field name="Units" …> … </field>
         <field name="Unit_Price" …> … </field>
         <field name="Total_Price" …>
            <calculate>
               <script>(Units * Unit_Price) * (1 - (Discount/100.0))</script>
            </calculate>
         </field>
         …
      </subform>
   </template>
</xdp:xdp>
```

Inferred Index for Ancestors of the Container

The same logic that is used at the level of the script's container also applies to ancestors or siblings of ancestors which match an unqualified SOM expression. In each case, the target's index defaults to the index of the ancestor or ancestor's sibling at the same level of the hierarchy. The result of this rule is that if a cell in a table contains a SOM expression that references another table, the SOM expression defaults to referencing the corresponding cell in the other table. For example, a form has been created to calculate trip times. In one field the user enters his estimated average driving speed in kilometres per hour. The form also displays two 5 by 5 tables. The first table shows distances between cities in kilometres and the second

shows estimated travel times between the same cities in hours. The content of each cell in the second table is calculated based upon the corresponding cell in the first table. The template is as follows.

Example 3.41 Template manipulating tables via inferred indexing

```
<xdp:xdp …>
   <template …>
      <subform name="Trip">
         <field name="Speed" …> … </field>
         <subform name="Distance">
            <subform name="Distances" …>
               <field name="Cell"…> … </field>
               <field name="Cell"…> … </field>
               …
            </subform>
            <subform name="Distances" …>
               <field name="Cell"…> … </field>
               <field name="Cell"…> … </field>
               …
            </subform>
         </subform>
         …
         <subform name="Time" …>
            <subform name="Times" …>
               <field name="Cell"…>
                  <calculate>
                     <script>Distance.Distances.Cell / Speed</script>
                  </calculate>
               </field>
               <field name="Cell"…>
                  <calculate>
                     <script>Distance.Distances.Cell / Speed</script>
                  </calculate>
               </field>
               …
            </subform>
            <subform name="Times" …>
               <field name="Cell"…>
                  <calculate>
                     <script>Distance.Distances.Cell / Speed</script>
                  </calculate>
               </field>
               <field name="Cell"…>
                  <calculate>
                     <script>Distance.Distances.Cell / Speed</script>
                  </calculate>
               </field>
               …
            </subform>
            …
         </subform>
      </subform>
   </template>
</xdp:xdp>
```

Each cell in the Timetable looks up the corresponding distance in the Distance table using the SOM expression `Distance.Distances.Cell`. Consider the case of the calculate script for the field `$template.Trip.Time.Times[3].Cell[2]`. The expression `Distance.Distances.Cell` is resolved as follows:

1. The current container is a field named `Cell`, specifically `$template.Trip.Time.Times[3].Cell[2]`. The `Cell` field does not have a property or child named `Distance`.

2. The `Cell` field's parent is a subform called `Times`, specifically `$template.Trip.Time.Times[3]`. The `Times` subform is not named `Distance` nor does it have a sibling named `Distance`.

3. The parent of `Times` is a subform called `Time`, specifically `$template.Trip.Time`. `Time` has a sibling called `Distance`. Hence `Distance` is resolved to `$template.Trip.Distance`.

4. `$template.Grid.Distance` has multiple children called `Distances`. The SOM expression does not supply an index. Hence an index must be inferred. Inferring is possible because the corresponding node on the way up the tree, `$template.Trip.Times.Times[3]`, has an index. Its index is borrowed and `$template.Trip.Distance.Distances[3]` is selected.

5. `$template.Trip.Distance.Distances[3]` has multiple children called `Cell`. The SOM expression does not supply an index. Hence, an index must be inferred. Inferring is possible because the corresponding node on the way up the tree, `$template.Trip.Times.Times[3].Cell[2]`, has an index. Its index is borrowed and `$template.Trip.Distance.Distances[3].Cell[2]` is selected.

6. Because the script language is FormCalc, and a string is required by the context, and `$template.Trip.Distance.Distances[3].Cell[2]` has an associated value, the expression is resolved to `$template.Trip.Distance.Distances[3].Cell[2].rawValue`.

Note that when it comes to inferring an index it makes no difference whether or not a particular node on the way up has the same name as the corresponding node on the way down. Hence the tables do not have to match by name, they only have to be congruent (i.e. have the same dimensions).

The SOM expression `Speed` is easier to resolve because it does not need an index.

7. By the usual scoping logic `Speed` is resolved to the field `$template.Trip.Speed`. Because that field has no siblings with the same name, no index is required.

8. Because the script language is FormCalc, and a string is required by the context, and `$template.Trip.Speed` has an associated value, this expression is further resolved to `$template.Trip.Distance.Distances[3].Cell[2].rawValue`.

Finally the calculation can be done.

9. The number in `$template.Trip.Distance.Distances[3].Cell[2].rawValue` is divided by the number in `$template.Trip.Speed.rawValue` and the quotient assigned to `$template.Trip.Time.Times[3].Cell[2].rawValue`.

It is possible for the SOM expression to reach down to a lower level than the level of the script's container. In that case, when the SOM expression does not specify an index and an index is required, `[0]` is assumed. The same thing happens when the script's container does not have siblings of the same name. In short, when an index is needed but none is supplied and there is no way to infer an index, `[0]` is used.

Index inferral must not be combined with the use of ".parent" in the same SOM expression. The SOM expression resolver is not required to correctly handle the inferral when the SOM expression contains ".parent".

Index inferral may be used with references by class. For example consider the following template.

Example 3.42 Index inferral used with a reference by class

```
<xdp:xdp …>
   <template …>
      <subform name="root">
         <subform name="A">
            <subform> … </subform>
            <subform name="P">
               <field name="X">
                  <calculate>
                     <script>B.#subform.Y</script>
                  </calculate>
               </field>
            </subform>
         </subform>
         <subform name="B">
            <subform> … </subform>
            <subform name="Q">
               <field name="Y"> … </field>
            </subform>
         </subform>
      </subform>
   </template>
</xdp:xdp>
```

the expression B.#subform.Y resolves to $form.root.B.#subform[1].Y, which is to say the field named Y within the second child of subform B. The index "[1]" is copied from the corresponding level in the current object's full name, which is expressed as $form.root.A.#subform[1].X. Note that the current subform and/or the referenced subform may be named, as shown, but even when they are the names have no effect on the resolution of the expression.

Relative Index

Sometimes it is necessary for a script associated with one cell in an array to refer to another cell on another line above or below. A special syntax is provided for this. Within an unqualified reference an index of the form "[-*nnn*]" is interpreted as the current container's index minus *nnn*, while an index of the form "[+*nnn*]" is interpreted as the current container's index plus *nnn*. For example, in the following example an item number field has been added to each detail record. This example is based upon the spreadsheet-style template of Example 3.38 so all of the Item_No fields are siblings. The first Item_No field defaults to 1. Each remaining Item_No field automatically calculates its value as one more than the previous Item_No field. Furthermore, if the user manually enters a value into the first Item_No field all subsequent item numbers are automatically updated.

Example 3.43 Relative indexing

```
<xdp:xdp …>
   <template …>
      <subform name="Receipt" …>
         <field name="Item_No" …>
```

```
            <value>
               <integer>1</integer>
            </value>
         </field>
         <field name="Description" …> … </field>
         <field name="Units" …> … </field>
         <field name="Unit_Price" …> … </field>
         <field name="Total_Price" …> … </field>
         <field name="Item_No" …>
            <calculate>
               <script>Item_No[-1] + 1</script>
            </calculate>
         </field>
         …
         <field name="Description" …> … </field>
         <field name="Units" …> … </field>
         <field name="Unit_Price" …> … </field>
         <field name="Total_Price" …> … </field>
         …
      </subform>
   </template>
</xdp:xdp>
```

Relative indexing can also be used with inferred indexes. Relative indexes are defined as relative to the unadorned reference. Hence the full meaning of "`[-nnn]`" is "indexed by *nnn* less than what it would have been if `[-nnn]` had not been specified". Similarly the full meaning of "`[+nnn]`" is "indexed by *nnn* more than it would have been if `[+nnn]` had not been specified".

Relative indexing *cannot* be used in fully-qualified SOM expressions because such expressions cannot infer indexes.

SOM Expressions That Reference Variables Properties

SOM expressions may reference values or named script objects specified in `variables` properties. Variables may be used to hold boilerplate or image references or to define script object references. Because SOM expressions that reference variable properties usually appear in calculation or validation scripts, their use is described in "Document Variables" on page 325.

4 Exchanging Data Between an External Application and a Basic XFA Form

This chapter explains the basic steps involved with exchanging data between an external application and an XFA processing application hosting an XFA form.

This chapter assumes the format of the data provided by the external application is compatible with the XFA default data loading rules. "Dealing with Data in Different XML Formats" on page 423 explains how to translate data into a representation that is compatible with the XFA default data loading.

This chapter contains the following sections:

- "Creating, Updating, and Unloading a Basic XFA Data DOM" explains how mapping rules influence where data in the XML Data DOM is placed in an XFA Data DOM, how the XML Data DOM is updated when data in the XFA Data DOM changes, and how data in the XML Data DOM is unloaded.
- "Localization and Canonicalization" explains the role locale plays when data is being loaded into or out of an XFA Data DOM or Form DOM.
- "Basic Data Binding to Produce the XFA Form DOM" explains how containers in the Form DOM are bound with data in the XFA Data DOM.

Creating, Updating, and Unloading a Basic XFA Data DOM

This section explains how generic well-formed XML documents are processed by XFA processing applications and how they present an object and processing model for interpreting and manipulating the structured data contained within the XML document. This section refers to such XML documents as *XML data documents*.

Background and Goals

The reader of this specification will learn how XML data documents are handled by XFA processing applications, how the data is mapped to an object model known as the XFA Data DOM, and how the data is mapped back out again during the creation of a new XML data document. This information is valuable to authors of form templates who will be processing existing XML data, and to people or systems producing data that will be serviced by XFA processing applications.

This section primarily focuses on the processing of XML data documents; however, it was a design goal that the same object model could also be used to represent data from non-XML sources such as a database or other data formats such as comma-separated values. For this reason the XFA Data DOM does not interact directly with XML but instead goes through an XML Data DOM. Custom processors can replace the XML Data DOM with a DOM appropriate for their non-XML data formats.

It should be noted that the concepts and guidelines presented by this specification are also valuable to other types of XML processing applications outside the realm of XFA, such as applications concerned with utilizing XML for data exchange and mapping the data into structured documents types other than forms. A reduction in implementation costs and a gain in flexibility can be obtained by taking a simplified view of how XML expresses data. The object model described by this specification represents such a simplified view.

An important philosophy underlying this specification is to place as few requirements as possible on the data. One of the strengths of XML is that it provides a comprehensible, and often self-evident

representation of data. Data expressed in XML can be manipulated via off-the-shelf, and often freely available, processing tools; in the worst case the data can be manipulated using a common text editor. Hence, it would be contrary to this inherent property of XML to require that data conform to a specific grammar or schema before it can be used within an XFA processing application.

XFA Data DOM Basic Concepts

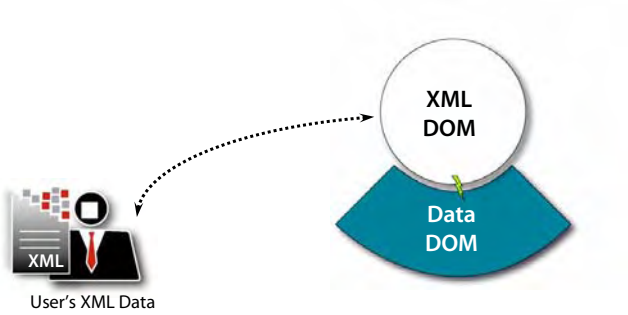
This section introduces basic concepts behind the XFA Data Document Object Model. It introduces the following concepts:

- How data is represented in an XFA data document object model (DOM), how the tree structure of such a DOM aids data organization, how data within the Data DOM is identified
- Relationship between the XFA Data DOM and the XML Data DOM, and how this can simplify your implementation
- Notation used to represent data

About the XFA Data DOM

The interpretation of an XML data document is a mapping operation of the data into an object model known as the "XFA Data Document Object Model", commonly referred to as the XFA Data DOM. The behavior of the mapping is governed by rules described by this specification. The software component responsible for performing the mapping of the XML data document into the XFA Data DOM is referred to in this specification as the *data loader*. There is in addition a reverse function that maps from the XFA Data DOM to a new XML data document. The software component responsible for this reverse mapping is referred to in this specification as the data unloader.

Data loader creating an XFA Data DOM



The XFA Data DOM provides a set of software interfaces to the data mapped from an XML data document. In principle, the same model could also be used to represent data from non-XML sources such as a database or other data formats such as comma-separated values. The XFA Data DOM provides interfaces that are simpler than a generic XML Document Object Model [XMLDOM2]. There are fewer interfaces in the XFA Data DOM as compared to the XML DOM, and many of the physical structures of XML are abstracted away. Notwithstanding the wider applicability of the XFA Data DOM, this specification assumes that the XML data document is first loaded into an XML Data DOM and from there into the XFA Data DOM.

The XFA Data DOM encloses a tree structure in which each node is either a `dataValue` object or a `dataGroup` object. In most cases nodes correspond to individual XML elements and are peered with individual nodes in the XML Data DOM. Parent-child relationships correspond to element nesting relationships, that is, the element corresponding to the parent of any given node contains the element corresponding to the given node. In addition the children of any node are ordered by age, that is the first child acquired by the node is the eldest, the next child acquired is the second-eldest and so on. In XFA

specifications when a tree is drawn pictorially sibling nodes are shown in order by age, with the oldest at the left and the youngest at the right. In the case of the XFA Data DOM the result of this ordering is that a tree walk that goes in depth-first order, and left-to-right at any level, traverses the data in the same order as it was present in the original XML data document.

The objects in the XFA Data DOM and their properties are exposed via the XFA Data DOM interfaces. Every object in the XFA Data DOM has the following exposed property:

Property	Description
`name`	A string of any length (including zero-length, that is empty) which is a non-unique identifier for the object. This corresponds to the local part of either the element type or the attribute name in the XML data document.

In addition, each object created by the data loader has an internal pointer to the node in the XML Data DOM with which it is peered. Furthermore, some objects have additional properties appropriate to their object types, as described below.

For example, consider the following fragment of XML.

Example 4.1 Data fragment with "tree" element

```
<abc:tree xmlns:abc="http://www.example.org/orchard/">apple</abc:tree>
```

When loaded into the XFA Data DOM using default mapping rules, the `dataValue` node representing this data has a property called `name` with a value of `tree`.

After the XFA Data DOM has been loaded the XFA application may update it. Updates may include adding, deleting, moving, and changing the properties of nodes. These changes are passed through to the XML Data DOM by the XFA Data DOM so that the two data DOMs stay synchronized. When the data unloader runs it creates the new XML data document based upon the contents of the XML Data DOM, as updated by the application.

Note that the exposed properties may be set by an XFA application to any Unicode string, including the empty string `""`. This allows XFA applications to construct arbitrary data structures. However the XML 1.0 Specification [XML] imposes additional restrictions upon element types, namespace prefixes, and URIs. Hence when the XFA Data DOM is unloaded to an XML data document the result may be malformed XML. It is up to the application to ensure, if desired, that the restrictions of XML with regard to element types, namespace prefixes and URIs are respected.

The XFA Data DOM is part of a larger tree that holds all exposed XFA nodes. The single large tree makes it possible to refer to XFA nodes using a unified format known as a Scripting Object Model (SOM) expression. The grammar of SOM expressions is described in "Scripting Object Model" on page 73. Briefly, an expression consists of a sequence of node names separated by periods (`"."` characters). Starting from some point in the XFA tree, each name identifies which child of the current node to descend to. The Data DOM descends from a node named `data` which is a child of `datasets`, which is a child of `xfa`, which is the root. The XML data document does not supply the `xfa`, `datasets`, or `data` nodes; instead the data loader creates them automatically and makes the node mapped to the outermost element of the data the child of `data`. Consider the following XML data document.

Example 4.2 Complete data document

```
<?xml version="1.0" encoding="UTF-8"?>
<book>
   <ISBN>15536455</ISBN>
```

```
   <title>Introduction to XML</title>
</book>
```

When loaded into the XFA Data DOM, the node representing the `book` element would be referenced by the SOM expression `"xfa.datasets.data.book"`. The `ISBN` element would be referenced by `"xfa.datasets.data.book.ISBN"` and the `title` element by `"xfa.datasets.data.book.title"`.

Also, SOM expressions recognize the short-form `"!"` as equivalent to `"xfa.datasets."` and `"$data"` as equivalent to `"xfa.datasets.data"`. Thus for example the `title` element above could also be referenced as either `"!data.book.title"` or `"$data.book.title"`.

dataValue Nodes

A `dataValue` node is an object in the XFA Data DOM that corresponds to an element holding character data (and possibly other elements) in an XML data document. Within the XFA Data DOM leaf nodes are usually `dataValue` nodes. A `dataValue` node can have other `dataValue` nodes descended from it but it can not have any `dataGroup` nodes descended from it.

`dataValue` nodes have the following properties:

Property	Description
`contains`	A string identifying the source of the data. The string is set to `metadata` if the `value` property originated from an XML attribute, but to `data` if it did not.
`contentType`	A string identifying the type of the data. By default this is set to the empty string (`""`). The empty string is interpreted as equivalent to `text/plain`. Note however that the treatment of `text/plain` in XFA is more relaxed than that specified in [RFC2046] for the MIMEtype `text/plain`. The difference is that in XFA the data loader may recognize a line break signified by a newline character (U000A) without an accompanying carriage-return character (U000D).
`isNull`	A Boolean flag which is true if and only if the value of the data is null. Note that there is no simple way to detect null values other than by inspecting the `isNull` property, because the syntax defined by [XML Schema] allows an element to be explicitly declared null using the `xsi:nil` attribute even though the element contains data. Hence, a data node may contain data yet `isNull` may be `1`. When this occurs the correct behavior is to treat the value of the data node as null, on the grounds that explicit markup should override an implicit property. However when such a data node is unloaded to a new XML data document the value should be written out along with the `xsi:nil` attribute so that round-tripping preserves the original document as far as possible.
`value`	A string of Unicode characters holding the data associated with the node. The string may be of any length (including zero-length, that is empty). The string must not include the character code NUL (U0000). Hence, NUL may be used as the string terminator by XFA applications.

A `dataValue` node has the standard properties such as `name` plus the properties listed in the above table. These properties are exposed to scripts and SOM expressions.

In addition, for illustrative purposes in this chapter, we will think of a `dataValue` object as having a fictional property called `nullType`. This fictional property records the manner in which a null value was

represented in the XML data document from which it was loaded, and controls how it will be represented in the new XML data document when it is unloaded. This property takes one of three values. The value `xsi` means that a null value is represented by an `xsi:nil` attribute as defined in [XML Schema]. The value `empty` means that an empty value is represented by an empty element. The value `exclude` means that an empty value is represented by a missing element or attribute. Note that the first two values (`xsi` and `empty`) are only available for data nodes representing elements. For data nodes representing attributes the value of this property is always `exclude`.

In the following fragment of XML, the elements `ISBN`, `title`, `desc` and `keyword` all represent data values.

Example 4.3 Fragment showing simple content

```
<book>
   <ISBN>15536455</ISBN>
   <title>Introduction to XML</title>
</book>
```

In the above example, the element `ISBN` is represented by a `dataValue` node with a `name` property of `ISBN`, a `value` property of `"15536455"`, and a `contains` property of `data`. The element `title` is represented by a `dataValue` node with a `name` property of `title`, a `value` property of `"Introduction to XML"`, and a `contains` property of `data`.

When loading the `value` property, the data loader removes XML escaping, so that for example if the XML data contains the XML entity `<` this is represented in the `value` string as `"<"` rather than by the XML entity. The XML data document must not contain character data containing either a NUL character (U0000) or any escape sequence that evaluates to zero, for example `"&#0;"`.

`dataValue` nodes are permitted to be descended from other `dataValue` nodes in order to represent XML mixed content. The following example shows mixed content.

Example 4.4 Fragment showing mixed content

```
<book>
   <ISBN>15536455</ISBN>
   <title>Introduction to XML</title>
   <desc>Basic primer on <keyword>XML</keyword> technology.</desc>
</book>
```

The element `keyword` in the above example is represented in the XFA Data DOM by a `dataValue` node which is the child of the `dataValue` node representing the element `desc`. Note that the value of a `dataValue` node is the concatenation of the values of its child `dataValue` nodes. For instance, in the case of the `desc` data value above, the value is `"Basic primer on XML technology."`; the `XML` portion of the value is contributed by the `keyword` data value to its parent `desc` data value. The resulting data values have the properties:

Name	Value	Contains
ISBN	"15536455"	data
title	"Introduction to XML"	data
desc	"Basic primer on XML technology."	data
keyword	"XML"	data

This process of the parent incorporating the child's value is recursive to any number of levels. When a lower-level value changes all of the higher-level values incorporating it automatically change as well.

XML attributes are also by default treated as data values. Those `dataValue` nodes that result from mapping of XML attributes are marked as a different flavor of `dataValue` node. Such nodes resulting from attributes for the purpose of this specification are said to represent *metadata*. One benefit of this approach is the potential to place the values of these nodes back into attributes when unloading the XFA Data DOM into an XML data document.

Attributes may be prevented from loading by a configuration option as described in "The attributes Element" on page 427.

`dataValue` nodes that are considered to contain metadata are excluded from the value of any ancestor `dataValue` node. Consider the following XML data fragment that extends the previous example with the addition of a `language` attribute on the `desc` element.

Example 4.5 Data with an attribute

```
<book>
   <ISBN>15536455</ISBN>
   <title>Introduction to XML</title>
   <desc language="english"
      >Basic primer on <keyword>XML</keyword> technology.</desc>
</book>
```

In the above example the `value` property of the `desc` `dataValue` node is `"Basic primer on XML technology."` regardless of the presence of the `language` attribute and its corresponding `dataValue` node. The content of the data value element `keyword` contributes to the value of `desc`, but the content of the attribute `language` does not contribute. Hence the `dataValue` nodes resulting from this example have the properties listed in the following table. The only difference from the preceding example is the addition of a new `dataValue` node representing the attribute:

Name	Value	Contains
`ISBN`	`"15536455"`	`data`
`title`	`"Introduction to XML"`	`data`
`desc`	`"Basic primer on XML technology."`	`data`
`keyword`	`"XML"`	`data`
`language`	`"english"`	`metadata`

In many cases it is useful to distinguish between a data value which is empty (zero-length) and a data value which was never entered. For example, suppose a form has ten fields into which the user may enter numbers. The form is required to calculate and display the average of the numbers in the fields. However, the user enters only six numbers, leaving four fields null. If the calculation simply adds all ten fields together and divides the result by ten, it will treat the four null fields as zeroes and get the wrong answer. Instead it needs to count and sum just the non-null fields. The easiest way to do this is to use the `isNull` property of the `dataValue` node associated with each field.

dataGroup Nodes

A dataGroup node is an object in the XFA Data DOM that corresponds to an element holding other elements (as opposed to character data) in an XML data document. Within the XFA Data DOM interior nodes are usually dataGroup nodes. A dataGroup node may have other dataGroup nodes and/or dataValue nodes descended from it. Note however that while a dataValue node can descend from a dataGroup node, a dataGroup node can never descend from a dataValue node.

Some XML data documents enclose repeating sets of data, with the same or similar structure but with different content. This specification refers to these repeating sets of data as *records*. The outermost element of a record must map to a dataGroup node.

dataGroup objects have the common properties such as name. These properties are exposed to scripts and SOM expressions. In addition, for illustrative purposes in this chapter, we will pretend that they have some fictional properties.

A dataGroup object can be thought of as possessing a fictional property called nullType. This fictional property controls the manner in which null values are represented in XML representations for data value nodes descended from this node. dataGroup nodes themselves do not have any value, null or non-null, so this property has no effect on the dataGroup node itself. This property is present purely so it can be inherited by the nullType properties of those dataValue nodes which are children of the dataGroup node.

One can also think of a dataGroup object as possessing a fictional property called isRecord. In reality scripts determine whether a dataGroup is or is not a record by calling the isRecordGroup(NODE) method of the $dataWindow object. The isRecord fictional property represents the value that would be returned by this method if passed this dataGroup.

In the following fragment of XML the element book represents a data group containing data values ISBN, title, desc, and keyword.

Example 4.6 Data fragment showing an element that maps to a dataGroup

```
<book>
   <ISBN>15536455</ISBN>
   <title>Introduction to XML</title>
   <desc>Basic primer on <keyword>XML</keyword> technology.</desc>
</book>
```

In the above example the element book is represented in the XFA Data DOM by a dataGroup node with a name property of book. The dataGroup node may be a record or not depending on the context within the XML data document and depending on the configuration option described in "The record Element" on page 445. If the above text was the entire XML data document and default mapping rules were used, it would be a record.

As described in "dataValue Nodes" on page 111 the ISBN, title and desc elements are represented by dataValue nodes in the Data DOM. Those nodes are all children of the dataGroup node representing the book element. The keyword element is also represented by a dataValue node but it is a child of the dataValue representing the desc element, hence a grandchild of the dataGroup node.

Relationship Between the XFA Data DOM and the XML Data DOM

After the XFA Data DOM has been loaded the XFA application may update it. Updates may include adding, deleting, moving, and changing the properties of nodes. These changes are passed through to the XML

Data DOM by the XFA Data DOM so that the two data DOMs stay synchronized. When the data unloader runs it creates the new XML data document based upon the contents of the XML Data DOM, as updated by the application.

Note that the exposed properties may be set by an XFA application to any Unicode string, including the empty string `""`. This allows XFA applications to construct arbitrary data structures. However the XML 1.0 Specification [XML] imposes additional restrictions upon element types, namespace prefixes, and URIs. Hence when the XFA Data DOM is unloaded to an XML data document the result may be malformed XML. It is up to the application to ensure, if desired, that the restrictions of XML with regard to element types, namespace prefixes and URIs are respected.

Some data loader options cause the data loader to load only a subset of data from the XML Data DOM into the XFA Data DOM. The result is that the ignored data is still in the XML Data DOM but is not accessible through the XFA Data DOM. Consequently the XFA application is unable to alter the ignored data. When the data unloader writes out a new XML data document, since the ignored data has been kept untouched in the XML Data DOM, it is written out in the new document without significant[1] changes. This applies to data which is always excluded from the document range as described in "Document Range" on page 117. It applies to data which is excluded using the extended mapping rules described in "The excludeNS Element" on page 428, "The startNode Element" on page 452, and "The range Element" on page 444. Finally, it applies to all extended mapping rules invoked by the `ignore` keyword, as described in "The attributes Element" on page 427, "The ifEmpty Element" on page 436, and "The presence Element" on page 442.

The `name` property of a node in the XFA Data DOM may be altered by the application or by the data loader, but doing so does not affect the `name` property of the peered node in the XML Data DOM. Consequently the node in the XFA Data DOM is accessible to scripts under its new name, but when the peered node is written out from the XML Data DOM to a new XML data document it retains its original name. This applies to nodes renamed via the data-loading options described in "The nameAttr Element" on page 440 and "The rename Element" on page 451. On the other hand when the application creates a new node in the XFA Data DOM, there is no existing peer in the XML Data DOM, so one is created and given the same `name` property.

The remaining data loader options cause the data loader to alter the content of the XML Data DOM along with the XFA Data DOM. When the data is unloaded the alteration is reflected in the new XML data document. This applies to extended mapping rules described in "XSLT Preprocessing" on page 458, "The whitespace Element" on page 454 and "XSLT Postprocessing" on page 458. It also applies to the rules invoked via the `remove`, `dissolve` or `dissolveStructure` keywords as described in"The presence Element" on page 442, and "The ifEmpty Element" on page 436.

Other updates by the XFA application carry through to the XML Data DOM. When updating any property other than `name`, and when deleting, inserting, or moving a node in response to a request from the XFA application, the XFA Data DOM propagates the update to the XML Data DOM.

Moving a node can cause a name conflict if, for example, a data value containing an attribute (`contains` set to `metadata`) is moved to a location where there is already a peer representing an attribute with the same name. A similar situation can arise from a request to create a new node. Carrying out such a request would result in a structure that violates the linkage rules of the XML Data DOM. The XML Data DOM refuses to carry out such a request and the XFA Data DOM in response returns an error code to the XFA application and leaves the XFA Data DOM in a state consistent with the XML Data DOM.

1. The XML standard [XML1.0] defines some contexts in which certain content is not significant, for example whitespace preceding the closing `">"` of a tag.

Tree Notation

This specification illustrates the contents of an XFA Data DOM, using the DOM notation described "Document Object Model Notation" on page xi. This section provides examples that apply this convention to the XFA Data DOM.

Data groups are expressed in the following form:

```
[dataGroup (name)]
```

where *name* represents the `name` property of the data group.

Data values are expressed in the following form:

```
[dataValue (name) = "value"]
```

where *name* represents the `name` property of the data value, and *value* represents the `value` property.

The `contains` property of a data value has a value of `data` unless some other value is explicitly shown. That is, it will only be expressed in this notation when it has a value of `metadata`, as described by the following two examples:

```
[dataValue (ISBN) = "15536455"]
```

In the above example the data value ISBN has a `value` property of `"15536455"` and, although it isn't explicitly stated by the notation, it also has a `contains` property of `data`.

```
[dataValue (status) = "stocked" contains="metadata"]
```

In the above example, the data value `status` has a `value` property of `"stocked"` and a `contains` property of `metadata`.

Similarly, within this specification the `contentType` property of a data value has a value of `the empty string ("")` when not explicitly shown. Likewise the `isRecord` property of a data group has a value of false when not explicitly shown.

Indenting is used to show parent-child relationships between nodes. In the following example a data group named `book` is the parent of a data value named ISBN:

```
[dataGroup (book)]
   [dataValue (ISBN) = "15536455"]
```

Within a group of sibling nodes, the age relationship is shown by the vertical ordering. The eldest child is at the top, the youngest at the bottom. In the following example `ISBN` is the eldest child of `book`, `title` is the middle child, and `author` is the youngest. Between the children of `author`, `firstname` is the older and `lastname` the younger.

```
[dataGroup (book)]
   [dataValue (ISBN)  = "15536455"]
   [dataValue (title) = "Introduction to XML"]
   [dataGroup (author)]
      [dataValue (firstname) = "Charles"]
      [dataValue (lastname)  = "Porter"]
```

In some complex XML data documents the elements that correspond to data groups or data values may be annotated with an XML namespace [XMLNAMES]. In these cases, the namespace and, if present, the namespace prefix are shown in the notation used here even though they are not present in the XFA Data DOM itself. The notation is as follows:

```
[dataGroup (prefix:book) xmlns="uri"]
   [dataValue (prefix:ISBN) = "15536455" xmlns="uri"]
```

In order to not clutter each example with namespace information, only examples that depend upon namespace information will include this form of the notation. The *prefix* refers to a namespace prefix as described by [XMLNAMES]. The *uri* refers to the corresponding Uniform Resource Identifier as described by *RFC2396: Uniform Resource Identifiers (URI): Generic Syntax* [URI].

The `isNull` property and the fictional `nullType` property are not illustrated in the above examples. To avoid clutter they are only shown if the value of `isNull` is true. The value `1` is used to represent true.

Similarly the fictional `isRecord` property is only illustrated where it is of interest and true. Again the value `1` is used to represent true.

It is worth repeating that the above notation is provided only as means within this specification to depict the structure and content within an XFA Data DOM.

Default Data Mapping Rules

There is a set of rules that govern, by default, how an XML data document is mapped to an XFA Data DOM and vice-versa. These rules have the effect of directing the data loader to usefully interpret the XML document content by mapping the physical structures of an XML data document into data values and data groups. In addition they direct the data unloader to map the data values and data groups into a new XML data document.

The default mapping rules may be overridden by options in the configuration document (i.e. the `config` section of the XDP). Those options and their effects are described in "Extended Mapping Rules" on page 423.

Document Range

The term *document range* refers to the portion of the XML data document that is processed by the data loader, such as the whole XML data document or a fragment. Assuming the XML data document starts as a file containing serialized XML, the first step in processing is to load the entire content of the XML data document into an XML Data DOM. The portion of the XML data document corresponding to the document range is then loaded into and accessible from the XFA Data DOM, and any portions of the data that are outside of the document range are not accessible via the XFA Data DOM. When the data unloader creates a new XML data document it stitches together the data within the XFA Data DOM and the data excluded from the XFA Data DOM (but still in the XML Data DOM) to create a new serialized XML file.

The document range can be influenced via the use of a number of extended mapping rules, however there is a by default document range which is described in more detail in the following subsections.

The document range is the portion of content corresponding to the intersecting result of applying the rules described by the following sections, in order. In other words, the mechanisms described by this section (and in the corresponding sections within the ""Extended Mapping Rules" on page 423") each excludes more content from the document range. The order that the data loader applies the rules is as follows:

1. "XML Logical Structures": Constrain the document range to a limited set of XML logical structures.
2. "Start Element": Additionally constrain the document range to the document content within a specified element.

3. "Namespaces": Additionally constrain the document range to a set of content belonging to a set of XML namespaces.

4. "Record Elements": Partition the document range into records enclosed within specified elements.

XML Logical Structures

An XML data document is comprised of a number of physical and logical structures (see the XML specification [XML] for more information on logical and physical structures).

The document range is constrained to a range of document content not greater than the content that corresponds to the following logical structures:

- elements with character data
- elements with element content
- elements with mixed content
- empty elements
- attributes

This means that XML processing instructions, for example, are not included in the document range.

Start Element

The notion of a *start element* represents the element where the XFA data handling processing begins, and consequently the end of the element determines where processing ends; the start element is a way to specify that a particular element within the XML data document actually represents the root of the document.

By default the start element corresponds to the root element (also known as the document element) of the XML data document (see section 2.1 "Well-Formed XML Documents" of the XML specification [XML] for more information about the root element). Therefore, the data loader by default maps the content of the document beginning with the root element inclusively.

The document range is constrained to a range of document content not greater than that specified by the start element.

Consider the following XML data document, which holds data pertaining to a single order from a book store.

Example 4.7 Book order example data

```
<?xml version="1.0" contentType="UTF-8"?>
<order>
   <number>1</number>
   <shipto>
      <reference><customer>c001</customer></reference>
   </shipto>
   <contact>Tim Bell</contact>
   <date><day>14</day><month>11</month><year>1998</year></date>
   <item>
      <book>
         <ISBN>15536455</ISBN>
         <title>Introduction to XML</title>
         <author>
            <firstname>Charles</firstname>
```

```
            <lastname>Porter</lastname>
          </author>
          <quantity>1</quantity>
          <unitprice>25.00</unitprice>
          <discount>.40</discount>
        </book>
      </item>
      <item>
        <book>
          <ISBN>15536456</ISBN>
          <title>XML Power</title>
          <author>
            <firstname>John</firstname>
            <lastname>Smith</lastname>
          </author>
          <quantity>2</quantity>
          <unitprice>30.00</unitprice>
          <discount>.40</discount>
        </book>
      </item>
      <notes>You owe $85.00, please pay up!</notes>
</order>
```

By default, the start element corresponds to the root element, which in the above example is the `order` element. The data loader by default maps the entire document, which results in the following mapping:

```
[dataGroup (order) isRecord="true"]
   [dataValue (number) = "1"]
   [dataGroup (shipTo)]
      [dataGroup (reference)]
         [dataValue (customer) = "c001"]
   [dataValue (contact) = "Tim Bell"]
   [dataGroup (date)]
      [dataValue (day)   = "14"]
      [dataValue (month) = "11"]
      [dataValue (year)  = "1998"]
   [dataGroup (item)]
      [dataGroup (book)]
         [dataValue (ISBN)  = "15536455"]
         [dataValue (title) = "Introduction to XML"]
         [dataGroup (author)]
            [dataValue (firstname) = "Charles"]
            [dataValue (lastname)  = "Porter"]
         [dataValue (quantity)  = "1"]
         [dataValue (unitprice) = "25.00"]
         [dataValue (discount)  = ".40"]
   [dataGroup (item)]
      [dataGroup (book)]
         [dataValue (ISBN)  = "15536456"]
         [dataValue (title) = "XML Power"]
         [dataGroup (author)]
            [dataValue (firstname) = "John"]
            [dataValue (lastname)  = "Smith"]
         [dataValue (quantity)  = "2"]
         [dataValue (unitprice) = "30.00"]
         [dataValue (discount)  = ".40"]
```

```
[dataValue (notes) = "You owe $85.00, please pay up!"]
```

In the above example the document range is the range of document content enclosed by the `order` data group element.

Namespaces

It is common for XML data documents to be comprised of content belonging to more than one namespace. The formal specification of XML namespaces is provided by the "Namespaces for XML" [XMLNAMES] specification.

Namespace inheritance is described fully in the [XMLNAMES] specification. Briefly, each element or attribute that does not explicitly declare a namespace inherits the namespace declared by its enclosing element. A namespace is declared using a namespace prefix. The namespace prefix must be associated with a URI either in the element using the namespace prefix or in an enclosing element. The same rules apply to the XFA Data DOM except for namespaces that are reserved for XFA directives, as described below.

The following example illustrates an XML document containing information about a purchase from a bookstore. The information is partitioned into two namespaces. The default namespace represents the order information needed by the bookstore for inventory and shipping purposes. The other namespace represents accounting information pertaining to the e-commerce transaction.

Example 4.8 Data document using multiple namespaces

```
<?xml version="1.0" encoding="UTF-8"?>
<invoice xmlns:trn="http://www.example.com/transaction/">
   <item>
      <book>
         <ISBN>15536455</ISBN>
         <title xml:lang="en">Introduction to XML</title>
         <quantity>1</quantity>
         <unitprice currency="us">25.00</unitprice>
         <discount>.40</discount>
         <trn:identifier>27342712</trn:identifier>
      </book>
   </item>
</invoice>
```

The use of namespaces within an XML data document assists in the determination of data values and data groups, and can also exclude document content from data loader processing. The by default handling of namespaces is described in this section, and additional control over namespace processing is described in the section "The excludeNS Element" on page 428.

The document range always excludes any document content as follows:

- content belonging to the namespace "`http://www.xfa.com/schema/xfa-package/`"
- content belonging to the namespace "`http://www.xfa.org/schema/xfa-package/`"
- content belonging to the namespace "`http://www.w3.org/2001/XMLSchema-instance`", including `nil` attributes
- attributes with a namespace prefix of `xml`, such as `"xml:space"` and `"xml:lang"`
- attributes belonging to the namespace "`http://www.xfa.org/schema/xfa-data/1.0/`"
- namespace declaration attributes, including the default namespace attribute named `xmlns` and specific namespace attributes with the `xmlns` namespace prefix

The data loader may have additional namespaces that it considers to be excluded from the document range: this behavior is implementation-defined.

Note that some attributes, although excluded from processing as ordinary metadata, are nonetheless processed separately. Each data value node has an `isNull` property determining whether it holds a null value, which can be thought of as out-of-band processing of `nil` attributes in the "`http://www.w3.org/2001/XMLSchema-instance`" namespace. Furthermore, attributes belonging to the "`http://www.xfa.org/schema/xfa-data/1.0/`" namespace may affect processing by the data loader at load time but, in accordance with the rule above, are not included as metadata in the XFA Data DOM.

Consider again the preceding example, Example 4.8. There is an `invoice` element that belongs to the default namespace and also includes an `identifier` element that belongs to the namespace `"http://www.example.com/transaction/"`. The `invoice` element has an attribute which makes the `"trn"` prefix a synonym for the `"http://www.example.com/transaction/"` namespace. This declaration uses an attribute named `"xmlns:trn"`, which according to the rule above is excluded from the document range. Although the declaration has its intended effect (associating subsequent data with the appropriate namespace) it is itself excluded from the document range. Similarly the `xml:lang` attribute of the `title` element is excluded by another of the rules above. To illustrate this, the following is the same example with the document range represented in bold type.

Example 4.9 Same data document showing document range in bold

```
<?xml version="1.0" encoding="UTF-8"?>
<invoice xmlns:trn="http://www.example.com/transaction/">
   <item>
      <book>
         <ISBN>15536455</ISBN>
         <title xml:lang="en">Introduction to XML</title>
         <quantity>1</quantity>
         <unitprice currency="us">25.00</unitprice>
         <discount>.40</discount>
         <trn:identifier>27342712</trn:identifier>
      </book>
   </item>
</invoice>
```

The first element is also excluded from the document range, not on the basis of namespace but because it is an XML processing instruction, as described above in "XML Logical Structures" on page 118.

The result of mapping this XML data document using the default mapping is as follows:

```
[dataGroup (invoice) isRecord="true"]
   [dataGroup (item)]
      [dataGroup (book)]
         [dataValue (ISBN) = "15536455"]
         [dataValue (title) = "Introduction to XML"]
         [dataValue (quantity) = "1"]
         [dataValue (unitprice) = "25.00"]
            [dataValue (currency) = "us" content="metadata"]
         [dataValue (discount) = ".40"]
         [dataValue (trn:identifier) = "27342712"
           xlmns="http://www.example.com/transaction/"]
```

The above example demonstrates the automatic exclusion of attributes having the `xmlns` prefix and of attributes having the `xml` namespace prefix.

Record Elements

Some XML data documents enclose repeating groups of data, each with different content. This specification refers to these repeating groups of data as *records*. Typically a record holds the data from a single form instance.

By default, the data loader considers the document range to enclose one record of data represented by the first (outermost) data group within the document range.

Consider once again the book order shown in Example 4.7. The result of mapping this XML data document with default mapping rules is as follows:

```
[dataGroup (order) isRecord="true"]
   [dataValue (number) = "1"]
   [dataGroup (shipTo)]
      [dataGroup (reference)]
         [dataValue (customer) = "c001"]
   [dataValue (contact) = "Tim Bell"]
   [dataGroup (date)]
      [dataValue (day)   = "14"]
      [dataValue (month) = "11"]
      [dataValue (year)  = "1998"]
   [dataGroup (item)]
      [dataGroup (book)]
         [dataValue (ISBN)  = "15536455"]
         [dataValue (title) = "Introduction to XML"]
         [dataGroup (author)]
            [dataValue (firstname) = "Charles"]
            [dataValue (lastname)  = "Porter"]
         [dataValue (quantity)  = "1"]
         [dataValue (unitprice) = "25.00"]
         [dataValue (discount)  = ".40"]
   [dataGroup (item)]
      [dataGroup (book)]
         [dataValue (ISBN)  = "15536456"]
         [dataValue (title) = "XML Power"]
         [dataGroup (author)]
            [dataValue (firstname) = "John"]
            [dataValue (lastname)  = "Smith"]
         [dataValue (quantity)  = "2"]
         [dataValue (unitprice) = "30.00"]
         [dataValue (discount)  = ".40"]
   [dataValue (notes) = "You owe $85.00, please pay up!"]
```

In the above example the document range is the range of document content enclosed by the `order` data group element, and by default, is partitioned into a single record.

Data Value Elements

Consider once again the following XML data fragment, which repeats Example 4.3.

Example 4.10 Fragment showing simple content

```
<book>
   <ISBN>15536455</ISBN>
   <title>Introduction to XML</title>
```

```
</book>
```

In the above example the elements ISBN and title enclose character data. All such elements within the document range map to dataValue nodes with a name property corresponding to the local part of the element type (the name given in the element's start and end tags), and a rawValue property corresponding to the element content. The result of the mapping is as follows:

```
[dataGroup (book)]
   [dataValue (ISBN)  = "15536455"]
   [dataValue (title) = "Introduction to XML"]
```

The rules for mapping content of the XML data document into dataValue nodes are defined in the following sections.

Data Values Containing Character Data

For elements within the document range enclosing purely character data, the data loader maps each element into a dataValue node as specified:

- the name property of the dataValue node is set to the local part of the element type (tag name) of the element
- the value property of the dataValue node is set to the character data of the element
- the contains property of the dataValue node is set to data
- the isNull property of the dataValue node is set to 0

This behavior is illustrated by the previous example.

Data Values Containing Mixed Content

For elements within the document range enclosing mixed content, the data loader maps each element into dataValue nodes as specified:

- the name property of the dataValue node is set to the local part of the element type (tag name) of the element
- the value property of the dataValue node is set to the ordered concatenation of all of its child dataValue node's value properties, excluding children that contain metadata (see "Attributes" on page 128 for more information about the contains property)
- the isNull property of the dataValue node is set to 0

In addition, each enclosed unit of character data within the element maps to a dataValue node with a name property of an empty string, a value property corresponding to the unit of character data, a contains property of data, and an isNull property of 0. dataValue nodes created according to this rule are the children of the dataValue node mapped from the enclosing element. The child dataValue nodes are ordered in document order.

Consider the following example where the element desc has mixed content, repeating Example 4.4.

Example 4.11 Data value with mixed content

```
<book>
   <ISBN>15536455</ISBN>
   <title>Introduction to XML</title>
   <desc>Basic primer on <keyword>XML</keyword> technology.</desc>
</book>
```

The result of mapping this XML data document is as follows:

```
[dataGroup (book)]
   [dataValue (ISBN)      = "15536455"]
   [dataValue (title)     = "Introduction to XML"]
   [dataValue (desc)      = "Basic primer on XML technology."]
      [dataValue ()       = "Basic primer on "]
      [dataValue (keyword) = "XML"]
      [dataValue ()       = " technology."]
```

In the above example the element `desc` maps to a `dataValue` node where

- `"Basic primer on "` is an enclosed unit of character data so it maps to a `dataValue` node named `""`
- `"XML"` is the content of an enclosed element with the element tag `keyword` so it maps to a `dataValue` node named `"keyword"`
- `" technology."` is another enclosed unit of character data so it maps to another `dataValue` node named `""`
- each of these three `dataValue` nodes is a child of the `desc dataValue` node
- the children of `desc` are in the same order that they occur in the XML data document
- the value of `desc` is formed by concatenating `"Basic primer on "`, `"XML"`, and `" technology"` in that order

Data Values Containing Empty Elements

For each empty element within the document range, the data loader by default maps the element into a null `dataValue` node as specified:

- the `name` property of the `dataValue` node is set to the local part of the element type (tag name) of the element
- the `value` property of the `dataValue` node is set to an empty string
- the `contains` property of the `dataValue` node is set to `data`
- the `isNull` property of the `dataValue` node is set to `1`
- the `nullType` property of the `dataValue` node is set to `empty`

This specification provides an extended mapping rule described in section "The presence Element" on page 442 for overriding this by default behavior by explicitly forcing an empty element to be mapped to a data group. In addition, a data description may be used to specify a different type of null data handling for the element as described in "dd:nullType Attribute" on page 841.

Consider the following example where the element `desc` is empty.

Example 4.12 Data containing an empty element

```
<book>
   <ISBN>15536455</ISBN>
   <title>Introduction to XML</title>
   <desc></desc>
</book>
```

As defined by the [XML] specification, the empty element can be expressed in a more compact format as follows.

Example 4.13 Data containing an empty-element tag

```
<book>
   <ISBN>15536455</ISBN>
   <title>Introduction to XML</title>
   <desc/>
</book>
```

Whichever way the empty element is expressed, the result of the mapping is as follows:

```
[dataGroup (book)]
   [dataValue (ISBN)     = "15536455"]
   [dataValue (title)    = "Introduction to XML"]
   [dataValue (desc)     = "" isNull="1" nullType="empty"]
```

Data Values Representing Null Data

It is desirable to represent null values in XFA data documents in an unambiguous way so that data can make round trips without introducing artifacts. The XML 1.0 specification [XML1.0]does not provide such a mechanism. Instead a mechanism was defined in [XML Schema]. XFA adopts the `xsi:nil` attribute defined in XML Schema but extends it for greater flexibility. In XML Schema when the value of `xsi:nil` is `true` the element must be empty. In XFA the element may be non-empty even when the value of `xsi:nil` is `true`. XML Schema does not define any circumstance in which the value of `xsi:nil` may be `false`. XFA defines a value of `false` for `xsi:nil` to mean that the associated `dataValue` node is non-null, even if the element is empty. Thus in XFA any element, empty or not, may have an attribute `xsi:nil` which explicitly states whether or not it is null. Note that `xsi:` here and throughout this document stands for any namespace prefix which maps to the namespace `http://www.w3.org/2000/10/XMLSchema-instance`.

The default treatment of null values can be modified by a data description. A data description is a separate XML document that carries information about the schema of the data, as described in "Data Description Specification" on page 834. The data description may supply a non-default value for the `nullType` property of a particular element. The following table shows how loading is affected by `nullType` for various combinations of `nullType` value and context.

Example 4.14 How loading is affected by nullType

Input XML	nullType	value in the XFA Data DOM	isNull
`<title>A Book</title>`	`empty exclude xsi`	`"A Book"`	`0`
`<title /> <title></title>`	`empty`	`""`	`1`
`<title /> <title></title>`	`exclude xsi`	`""`	`0`
`<title xsi:nil="false">A Book</title>`	`empty exclude xsi`	`"A Book"`	`0`
`<title xsi:nil="false" /> <title xsi:nil="false"></title>`	`empty exclude xsi`	`""`	`0`

Input XML	nullType	value in the XFA Data DOM	isNull
`<title xsi:nil="true">A Book</title>`	`empty exclude xsi`	`""`[a]	`1`
`<title xsi:nil="true" />` `<title xsi:nil="true"></title>`	`empty exclude xsi`	`""`	`1`
The data description indicates that the element is required but the element is not present in the XML data document.	`empty`	`""`	`1`

a.INote that prior to version 2.5 of this specification this table did not distinguish which value was being shown, the value in the XFA Data DOM or the value in the XML Data DOM. In this case the XFA Data DOM contains a null value, however the original value (`"A Book"`) is preserved in the XML Data DOM. In effect the value is excluded from the document range. As with other excluded material, it is written to the output XML data document when the data is unloaded.

Note that the correct behavior is only defined for `xsi:nil` attributes with a value of `true` or `false`. The data document must not contain `xsi:nil` attributes with other values.

By default the XFA processor loads metadata (attribute values) into the Data DOM. However there is an option to exclude them as described in "The attributes Element" on page 427. When attributes are loaded they may also be represented by `dataValue` nodes containing null values. As with elements, the data description may specify `nullType` for an attribute. However the value of `nullType` can not be `xsi` for metadata. It must be one of `empty` or `exclude`. The following table shows how loading of metadata is affected by `nullType` for various values of `nullType` and contexts.

Example 4.15 How loading of metadata is affected by nullType

Input XML	nullType	value	isNull
`title="A Book"`	`empty exclude`	`"A Book"`	`0`
`title=""`	`empty`	`""`	`1`
`title=""`	`exclude`	`""`	`0`
The data description indicates that the attribute is required but the attribute is not present in the XML data document.	`empty`	`""`	`1`

Although XFA is tolerant of null values, external constraints may bar them in particular contexts. For example, within rich text (which is expressed as a fragment of XHTML) empty elements are legal but not null values. XHTML, as defined in [XHTML], does not comprehend null values.

It is important to understand that null values are in almost all ways treated the same as non-null values. For example, suppose that as a result of data binding a field in the Form DOM is bound to a node representing a null value in the Data DOM. This binding has the same effect as a binding to a non-null value. If a value is assigned to the field, that value propagates to the bound node in the Data DOM, and as a result the node in the Data DOM is no longer marked as null. Or if a value is assigned to the node in the Data DOM, that value propagates to the bound field in the Form DOM.

Null Data in Mixed Content

A data value may enclose a data value which in turn contains null data. For example, consider the following data.

Example 4.16 Null data in a data value enclosed by another data value

```
<book xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance">
   <ISBN>15536455</ISBN>
   <title>Introduction to XML</title>
   <desc>Primer on <keyword xsi:nil="true"></keyword>XML technology.</desc>
</book>
```

The result of mapping this XML data document is as follows:

```
[dataGroup (book)]
   [dataValue (ISBN)      = "15536455"]
   [dataValue (title)     = "Introduction to XML"]
   [dataValue (desc)      = "Primer on XML technology. "]
      [dataValue ()       = "Primer on "]
      [dataValue (keyword) = "" isNull="1" nullType="empty"]
      [dataValue ()       = "XML technology."]
```

Data Values Containing Element Content

A `dataValue` node cannot be the ancestor of a data group, and therefore once an element is mapped to a `dataValue` node, the descendent elements must also be mapped to `dataValue` nodes. For each element enclosing element content within the document range, where the enclosing element is mapped to a `dataValue` node, the data loader by default maps the element into a `dataValue` node as specified:

- the `name` property of the data group is set to the local part of the element type (tag name) of the element
- the `value` property of the `dataValue` node is set to the ordered concatenation of all of its child `dataValue` node's `value` properties, excluding children that contain metadata (see "Attributes" on page 128 for more information about the `contains` property)
- the `isNull` property of the `dataValue` node is set to 0 (even if all of its children have `isNull` set to 1)

Consider the following example with the element `stamp`.

Example 4.17 A dataValue contains an element that would otherwise map to a dataGroup

```
<book>
   <ISBN>15536455</ISBN>
   <title>Introduction to XML</title>
   <stamp
      >Ordered on <date><yr>2000</yr><mo>06</mo><day>23</day></date></stamp>
</book>
```

In the above example the element `date` encloses purely element content and would be mapped to a data group based upon the rules for data groups (described in the next section); however, the enclosing element `stamp` maps to a `dataValue` node and therefore the data loader maps the element `date` to a `dataValue` node so that the `stamp` `dataValue` node does not become the ancestor to a data group.

The result of mapping this XML data document is as follows:

```
[dataGroup (book)]
```

```
[dataValue (ISBN)      = "15536455"]
[dataValue (title)     = "Introduction to XML"]
[dataValue (stamp)     = "Ordered on 20000623"]
   [dataValue ()       = "Ordered on "]
   [dataValue (date)    = "20000623"]
      [dataValue (yr)    = "2000"]
      [dataValue (MO)    = "06"]
      [dataValue (day)   = "23"]
```

Data Group Elements

An element that encloses only other elements and whitespace, and is not itself enclosed by a data value element, is treated as a data group. For each such element the data loader by default maps the element into a `dataGroup` node as follows:

- the `name` property of the `dataGroup` node is set to the local part of the element type (tag name) of the element

Consider the following XML data document.

Example 4.18 Data containing an element that maps to a dataGroup

```
<book>
   <ISBN>15536455</ISBN>
   <title>Introduction to XML</title>
   <author>
      <firstname>Charles</firstname>
      <lastname>Porter</lastname>
   </author>
</book>
```

The result of the mapping is as follows:

```
[dataGroup (book)]
   [dataValue (ISBN)  = "15536455"]
   [dataValue (title) = "Introduction to XML"]
   [dataGroup (author)]
      [dataValue (firstname) = "Charles"]
      [dataValue (lastname)  = "Porter"]
```

As specified above, an element is not a candidate for mapping to a `dataGroup` node if it is enclosed within an element mapped to a `dataValue` node; this because dataValue nodes can not be ancestors to `dataGroup` nodes. An example illustrating this case is presented in "Data Values Containing Element Content" on page 127.

Attributes

The data loader by default loads attributes into the XFA Data DOM. This applies to attributes of elements associated with both data values and data groups. Each attribute is represented by a `dataValue` node with a `contains` property of `metadata`.

The data loader processes attributes of an element prior to processing any content of the element. The effect is that any `dataValue` nodes resulting from attributes appear as children of the parent node before (left of) any children resulting from processing content of the element. This is in keeping with the general

structure of the XFA Data DOM as described in "About the XFA Data DOM" on page 109, whereby a top-down left-to-right traversal reproduces document order.

The set of `dataValue` nodes resulting from processing attributes are ordered in an implementation-defined order. The XML specification [XML] states that by definition the order of attributes is not meaningful. Hence there is no justification for imposing any attribute ordering restriction upon the data unloader. On the other hand applications using the XFA Data DOM, if they are truly compliant to the XML specification, do not rely on any particular ordering of attributes apart from the previously stated condition that attribute nodes precede content nodes.

Consider the following XML data document.

Example 4.19 Fragment with attributes

```
<book status="stocked">
   <ISBN>15536455</ISBN>
   <title language="en"
      alternate="XML in Six Lessons">Introduction to XML</title>
</book>
```

In the above example the XML element `book` maps to a data group and has a `status` attribute with a value of `stocked`. The XML element `title` maps to a data value and has two attributes, a `language` attribute with a value of `en` and an `alternate` attribute with a value of `"XML in Six Lessons"`. By default the mapping is as follows.

```
[dataGroup (book)]
   [dataValue (status) = "stocked" contains="metadata"]
   [dataValue (ISBN)   = "15536455"]
   [dataValue (title)  = "Introduction to XML"]
      [dataValue (language) = "en" contains="metadata"]
      [dataValue (alternate) = "XML in Six Lessons" contains="metadata"]
```

Note that in the above mapping the dataValue node called `title` has two `dataValue` children, yet its value does not include the values of either of those children. This is because mapped attributes are labelled as metadata, and only `dataValue` nodes that contain data are included in the value of a parent `dataValue` node. This is described in "Data Values Containing Mixed Content" on page 123.

When the data is written out to a new XML data document the order of attributes on the `title` element *may* change as follows.

Example 4.20 Attributes may be written out in a different order

```
<book status="stocked">
   <ISBN>15536455</ISBN>
   <title alternate="XML in Six Lessons"
      language="en">Introduction to XML</title>
</book>
```

The same rules apply to attributes of elements containing mixed content. Consider the following XML data document:

Example 4.21 Data containing attributes on mixed content

```
<book>
   <desc class="textbook">Basic primer on <keyword
      class="technical">XML</keyword> technology.</desc>
</book>
```

The result of the mapping is as follows:

```
[dataGroup (book)]
  [dataValue (desc) = "Basic primer on XML technology."
    [dataValue (class) = "textbook" contains="metadata"]
    [dataValue () = "Basic primer on"]
    [dataValue (keyword) = "XML"]
      [dataValue (class) = "technical" contains="metadata"]
    [dataValue () = " technology."]
```

Attributes of empty data value elements are processed via the same rules as other elements. Consider the following XML data document.

Example 4.22 Data containing an empty element with an attribute

```
<item>
  <book>
    <ISBN>15536455</ISBN>
    <title>Introduction to XML</title>
    <unitprice currency="USD">25.00</unitprice>
    <desc language="en-US"/>
  </book>
</item>
```

In the above example the empty `desc` element maps to a `dataValue` node and has a `language` attribute.

Assume the XFA Configuration DOM has an `attribute` element containing `preserve`. Given that empty elements map to `dataValue` nodes, as described in section "Data Values Containing Empty Elements" on page 124, the result of the mapping is as follows:

```
[dataGroup (item)]
  [dataGroup (book)]
    [dataValue (ISBN)       = "15536455"]
    [dataValue (title)      = "Introduction to XML"
    [dataValue (unitprice)  = "25.00"]
      [dataValue (currency) = "USD" contains="metadata"]
    [dataValue (desc)       = ""]
      [dataValue (language) = "en-US" contains="metadata"]
```

Null Values with Attributes

Note: The information in this section was incorrect prior to version 2.5 of this specification.

A data value which has attributes is by definition not null. For example, consider the following XML data document.

Example 4.23 Data containing a null valued element with an attribute

```
<item xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance">
   <book>
      <desc language="en-US" xsi:nil="1"/>
   </book>
</item>
```

Assume there is a data description which assigns the property `dd:nullType="xsi"` to the `desc` field. The `xsi:nil` markup would normally cause the empty `desc` element to be loaded as a null value, as

described in "Data Values Representing Null Data" on page 125. However the value node possesses a child node (the metadata node) so it cannot be null. The result of the mapping is:

```
[dataGroup (item)]
   [dataGroup (book)]
      [dataValue (desc) = "" isNull="0"]
         [dataValue (language) = "en-US" contains="metadata"]
```

White Space Handling

XML data documents often include additional white space within elements strictly as a legibility aid; this white space is considered insignificant. Establishing comprehensive white space handling rules contributes to predictable processing of XML data documents.

An XFA application must produce the same result from processing two documents that differ only by insignificant white space.

The following Unicode characters are considered white space, as defined by [XML]:

- space U+0020
- tab U+0009
- carriage return U+000D
- line feed U+000A

Note that the [XML] specification allows for white space to be contained within the definition of an element known to enclose only element content, and that such white space is considered insignificant.

Distinguishing between significant and insignificant white space within an XML data document depends upon rules described in the following sections, and is dependent upon whether the white space is contained within a data group or a data value.

White Space in Data Groups

If an element within an XML data document has been determined to represent a data group, then all white space within the data group is ignored by the data loader and is not present in the XFA Data DOM.

This rule respects the common cases where authors of XML data documents include white space as a legibility aid within elements known to enclose only other child elements.

Consider the following example to illustrate the insignificance of white space within data groups. In the following fragment of XML the `book` element is known to represent a data group of data values `ISBN` and `title`.

Example 4.24 White space inside an element that maps to a dataGroup object

```
<book>
   <ISBN>15536455</ISBN>
   <title>Introduction to XML</title>
</book>
```

Note the additional newlines before and after the data values and the spaces inserted to indent the data values from the data group to improve legibility. The additional newlines and other white space within the `book` data group element are considered insignificant.

The data loader produces the same result from the above example as it would from the following.

Example 4.25 Equivalent data without the whitespace

```
<book><ISBN>15536455</ISBN><title>Introduction to XML</title></book>
```

White Space in Data Values

If an element within an XML data document has been determined to represent a data value, then by default all white space within the data value is considered significant. Significant white space is processed as content by the data loader and is present in the XFA Data DOM.

Consider the following example to illustrate the significance of white space within data values. In the following fragment of XML, the `book` element is known to represent a data group of data values `ISBN` and `title`.

Example 4.26 White space inside elements that map to dataValue objects

```
<book>
   <ISBN>  15536455  </ISBN>
   <title>
      Introduction to XML
   </title>
</book>
```

Note the additional newlines before and after the data values and the spaces inserted to indent the data values from the data group to improve legibility. As described in "White Space in Data Groups" on page 131, the additional newlines and other white space within `book` data group element is considered insignificant.

However, the data value element `ISBN` contains additional leading and trailing space around the text `"15536455"`; this white space is considered significant. In addition, the data value element `title` contains leading and trailing newlines and white space; this white space is also considered significant.

The data loader produces the same result from the above example as from the following.

Example 4.27 Equivalent data formatted differently

```
<book><ISBN>  15536455  </ISBN><title>
     Introduction to XML
   </title></book>
```

Rich Text

Some content may represent *rich text*, which is text with markup representing formatting information such as underlining. A subset of HTML and CSS markup is supported in XFA, as described in "Rich Text Reference" on page 1027.

The markup within rich text data is not represented by nodes in the XFA Data DOM. Rather, it is stripped of markup and represented in the XFA Data DOM as plain text, as described in the chapter "Representing and Processing Rich Text" on page 188.

Image Data

XML data documents may include image data, either in-line or by reference. To be recognized as enclosing image data an element must have a `contentType` attribute in the XFA data namespace (`http://www.xfa.org/schema/xfa-data/1.0/`). The value of this attribute must be a MIME-type

identifying an image type. During loading the XFA processor copies this value into the `contentType` property of the corresponding `dataValue` node in the XFA Data DOM. However, after inspecting the image data, the XFA processor may decide that the image is in a different format and update the value of the property in the XFA Data DOM. Any such inspection, if and when it is done, is done on a best-effort basis.

In addition, if the image is included in-line, there may be a `transferEncoding` attribute, also in the XFA data namespace. The value of this attribute indicates the manner in which the image data is encoded. There are two supported values for `xfa:transferEncoding`.

An `xfa:transferEncoding` value of `base64` means the data is encoded using the base 64 method described in [RFC2045]. This encoding method packs only six data bits into each character, but the resulting character are all legal element content. This is the default encoding method assumed when there is no `xfa:transferEncoding` attribute. For example, the following fragment of a data file contains image data.

Example 4.28 Data representing an image using base 64 encoding

```
<logo xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
  xfa:contentType="image/bmp">
Qk1uAQAAAAAAAD4AAAAoAAAAJgAAACYAAAABAAEAAAAAAADABAADYDgAA2A4AAIAAAAAAAAAAAAA
AP///wD//////AAAAP/////8AAAA//////wAAAD//////AAAAP/////8AAAA//////wAAAD8AAAA
/AAAAP38AH78AAAA/fAAHvwAAAD9wAAG/AAAAP2AAAb8AAAA/QAAAvwAAAD9AAAC/AAAAPwAAAD8
AAAA/AAAAPwAAAD8AAAA/AAAAPwAAAD8AAAA/AAAAPwAAAD8AAAA/AAAAPwAAAD8AAAA/AAAAPwA
AAD8AAAA/AAAAPwAAAD8AAAA/AAAAPwAAAD9AAAC/AAAAP0AAAL8AAAA/YAABvwAAAD9wAAG/AAA
AP3gAA78AAAA/fAAHvwAAAD9/AB+/AAAAPwAAAD8AAAA//////wAAAD//////AAAAP/////8AAAA
//////wAAAD//////AAAAP/////8AAAA
</logo>
```

In the above example the image is a raster scan of a black circle within a square black frame, encoded as a BMP. The content of the BMP file was then coded as base 64. Finally the base 64 encoded data was formatted by inserting line breaks and white space. This is acceptable because base 64 decoders are required to ignore line breaks and white space.

An `xfa:transferEncoding` value of `cdata` means the image data encoded according to the rules for XML, with each byte of data represented by a character or a single-character XML entity. For example, an image might be encoded as follows (only part of the data is shown).

Example 4.29 Data representing an image using cdata encoding

```
<logo xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
  xfa:contentType="image/bmp" xfa:contentEncoding="cdata"
  >&#xC0;aR&#x2B;m8J&#x21;1~p&#x27;…</logo>
```

In this case, the image data can not include line break or white space characters except as a literal part of the image data. The data shown does not correspond to any particular image. The ellipsis (...) represents additional data which, for even a very small image, would not fit on the page.

Caution: XML 1.0 restricts characters to the following production:

```
Char ::= #x9 | #xA | #xD | [#x20-#xD7FF] | [#xE000-#xFFFD] | [#x10000-#x10FFFF]
```

This restriction applies even to characters expressed as numeric entities, according to "Well-formedness constraint: Legal Character" in [XML1.0]. Hence an image encoded this way must not include bytes with the values `#x0` through `#x8`, `#xB`, `#xC`, `#xE`, or `#xF`.

XML 1.1 [XML1.1] removes this restriction, but XML 1.1 is not widely used and XFA processors are not required to support it. (Acrobat does not.) In addition, XFA expressly forbids the use of character code zero (`#x0`), as described on page 112.

If the image is included by reference, there must be an `href` attribute. Note that the `href` attribute, unlike the `transferEncoding` attribute, must be in the default namespace. It must *not* have a namespace prefix. The assumption is that the external reference is already there in the data for some other purpose. The value of the `href` must be a URI.

Example 4.30 Image referenced using a URI

```
<body xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
   <p>
      <img xfa:contentType="image/bmp" href="#Logo.bmp" />
   </p>
</body>
```

In the above example the XML data document happens to be an HTML page. The `href` attribute is intended for use by a browser, which understands the `img` tag. However, XFA does not understand this tag. Adding the `xfa:contentType` attribute informs the XFA processor that the reference is to an image. The form of the URI `#Logo.bmp` indicates it is a relative URI, that is, it points to another element in the same XML data document.

Within the XFA Data DOM the element representing the image is represented by a `dataValue` node. The `dataValue` node is normal except that its `contentType` property is set to an image MIME type. The `value` property of the `dataValue` node is the text, if any, that appears inside the element in the XML data document. For example, for an image included by value and encoded in base 64, as in the first example above, the value is the base 64 encoded string. For an image included by reference the element must be empty and is treated according to the rules for empty-element processing as described in "Data Values Containing Empty Elements" on page 124.

The `contentType` property is initially set to the MIME type supplied by the `contentType` attribute. However the XFA application may at any time update the `contentType` property to some other image type. This could happen, for example, if it examines the image data and discovers that it corresponds to some other image type. The XFA application is *not* required to perform this analysis.

For an image included by reference the `href` attribute is treated as a normal attribute. That is, it is loaded by default but may be excluded along with other attributes. If loading of attributes is enabled, it is represented in the XFA Data DOM in the same way as any other attribute, by a `dataValue` node with its `contains` property set to `metadata`.

To preserve security of local data, when an XFA processor encounters an image referenced by URI, the XFA processor verifies that the referenced location is inside the current package, i.e. inside the XDP or PDF that supplied the template. If it is not inside the current package, the reference is blocked.

Updating the XML Data DOM for Changes Made to the XFA Data DOM

The XFA application may make edits to the XFA Data DOM during processing. These edits may include insertions and deletions of nodes, moving nodes, and updating the content of nodes. The XFA Data DOM is responsible for ensuring that all such edits are propagated to the XML Data DOM to ensure the XFA Data DOM and the XML Data DOM stay synchronized. The XFA Data DOM detects and refuse to carry out edits that would lead to the production of invalid XML during data unloading. For example, if the XFA application attempts to create a data value with the same name as a sibling and both siblings have their

`contains` properties set to `metadata`, the XFA Data DOM refuses to create the new data value on the grounds that attributes of the same element must have unique names.

Updating the XFA Data DOM for Changes Made to the XML Data DOM (Append Loading)

The XFA Data DOM provides a facility for loading data from an XML data document into an XFA Data DOM that already contains nodes. This specification refers to such a load as an *append-load*. Append-loads are not used in normal processing but may be employed by scripts.

When carrying out an append-load the data loader follows the same rules described elsewhere in this specification, except that:

- The start element for the load is determined at invocation time, without regard to `startNode` option of the XFA Configuration DOM.
- The new data is appended as a subtree, the root of which is the child of an existing node. The node to which the new subtree is appended must be determined at invocation time.

The XFA Data DOM is responsible for propagating **nullType** properties into the XFA Form DOM. When the the XFA Data DOM is asked to provide node information, it consults the node's data description to determine whether it provides null type information. If the node provides such information, the XFA Data DOM provides it as the new data value node's `nullType` property. Otherwise the `nullType` property is inherited from the node's parent. The highest-level data group cannot inherit, so if its `nullType` is not specified, it defaults to the value `empty`.

Unload Processing

The data unloader provides a facility for creating or updating an XML data document that represents the XML Data DOM.

When invoked, the data unloader produces an XML data document which reflects the contents of the XML Data DOM, as of the moment at which the data unloader was invoked.

When unloading the XML Data DOM, the XML data document produced by the data unloader is such that **insofar as possible** the data can make a round-trip back to the XFA Data DOM. Round-tripping means that when the new document is subsequently loaded into an empty XFA Data DOM using all the same data loader configuration options, the resulting XFA Data DOM is indistinguishable from the original XFA Data DOM at the moment the data unloader was invoked. When the default data mapping rules described in this chapter are used round-tripping is always supported. Round-tripping is also supported for most, but not all, extended data mapping rules.

The data unloader encodes characters using XML character references when necessary to conform to XML. In lieu of character references it may use the special strings defined by the XML Specification [XML] which includes `"<"` for `"<"` (less-than sign), `">"` for `">"` (greater-than sign), and `"&"` for `"&"` (ampersand). Inside attribute values it may also use `"&quot;"` for `"""` (quotation mark) and `"&apos;"` for `"'"` (apostrophe) as defined by the XML specification [XML].

The data unloader may insert XML-style comment(s) into the output document. It should insert a comment near the beginning identifying itself and its version number.

Unloading Node Type Information

In order to support round-tripping the data unloader when necessary inserts attributes in particular elements to cause them to be loaded as the correct type of node, as described in "The xfa:dataNode Attribute" on page 456.

The need for this attribute can arise because of deletions during processing. Consider the following excerpt from an XML data document.

Example 4.31 Data read in before deletions

```
<author>
   <firstname>Charles</firstname>
   <lastname>Porter</lastname>
</author>
```

When loaded into the XFA Data DOM using default mapping rules the result is:

```
[dataGroup (author)]
   [dataValue (firstname) = "Charles"]
   [dataValue (lastname) = "Porter"]
```

Suppose that during processing the `firstname` and `lastname` dataValue nodes are deleted. The result is that the XFA Data DOM contains:

```
[dataGroup (author)]
```

By default empty elements are loaded as data values. To prevent this, the data unloader writes out the `author` element with an attribute that marks it as a dataGroup.

Example 4.32 Data written out with data group forced

```
<author xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
   xfa:dataNode="dataGroup"/>
```

Similarly if the configuration options are such that an empty element would be loaded as a `dataGroup`, but the element is being written to represent the content of a `dataValue`, the data unloader writes out the element with an `xfa:dataNode` attribute having a value of `dataValue`. For example, suppose default mapping rules are in force and the XML Data DOM (after some processing) contains:

```
[dataValue (foo) = "xyz"]
   [dataValue (bar) = "xyz"]
```

The node `foo` corresponds to an element containing nothing but another element, but such elements are normally loaded as data groups. Yet `foo` is a data value. When the new XML data document is created the data unloader adds an attribute to mark `foo` as a data value.

Example 4.33 Data written out with data value forced

```
<foo xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
   xfa:dataNode="dataValue"><bar>xyz</bar></foo>
```

Unloading Null Data

While unloading data the XFA processor may encounter data values that are null. This is signified by the `isNull` property having a value of 1. When the XFA processor encounters a null data value, it consults the `nullType` property of the same node to find out how to represent the null data in XML. The behavior of

the XFA processor with respect to null data while unloading is summarized by the following tables. Each of the tables assumes that the `name` property of the data value node is set to `title`.

When `nullType` is `empty` the following table applies:

nullType	isNull	value	contains	Output XML
empty	0	"A Book"	data	<title>A Book</title>
empty	0	""	data	<title xsi:nil="false" />
empty	1	"A Book"	data	Cannot occur. If a script sets `isNull` to 1, the `value` property is automatically set to the null string, and if a script sets `value` to something other than the null string, `isNull` is automatically set to 0.
empty	1	""	data	<title />
empty	0	"A Book"	metadata	title="A Book"
empty	0	""	metadata	title=""
empty	1	"A Book"	metadata	Cannot occur. If a script sets `isNull` to 1, the `value` property is automatically set to the null string, and if a script sets `value` to something other than the null string, `isNull` is automatically set to 0.
empty	1	""	metadata	title=""

When `nullType` is `exclude` the following table applies:

nullType	isNull	value	contains	Output XML
exclude	0	"A Book"	data	<title>A Book</title>
exclude	0	""	data	<title />
exclude	1	"A Book"	data	No output for this data node or its children. The node is marked transient and is excluded from the output document.
exclude	1	""	data	
exclude	0	"A Book"	metadata	title="A Book"
exclude	0	""	metadata	title=""
exclude	1	"A Book"	metadata	No output for this data node. The node is marked transient and is excluded from the output document.
exclude	1	""	metadata	

When `nullType` is `xsi` the following table applies:

nullType	isNull	value	contains	Output XML
xsi	0	"A Book"	data	<title>A Book</title>
xsi	0	""	data	<title />

nullType	isNull	value	contains	Output XML
xsi	1	"A Book"	data	<title xsi:nil="true">A Book</title>
xsi	1	""	data	<title xsi:nil="true" />

Note that `nullType` cannot have the value `xsi` when `contains` is set to `metadata`.

Logical Equivalence

The rules stated above ensure that every implementation produces logically equivalent output given the same inputs. Logical equivalence includes exact character-for-character reproduction of the content of elements that map (or would map, if they were within the document range) to data values, including attribute values. However it does not include white space within the character data of elements that map (or would map) to data groups. Hence, the data unloader may insert white space characters and newlines within elements representing data groups. This is useful for improving readability. When the output XML data document is loaded into a new XML Data DOM the new XML Data DOM does not necessarily have the same content as the original XML Data DOM, however the XFA Data DOM that derives from it has the same content as the original XFA Data DOM.

Localization and Canonicalization

This section explains how XFA processing applications support locales (languages or nation-languages) when loading data into the XFA Data DOM or editing data in the Form DOM and when unloading, displaying, and printing such data.

Before beginning this section, it is important to understand that all data in the XFA Data DOM and the Form DOM is represented in a standardized, locale-agnostic format, called *canonical format*. Canonical format plays a role in the following conversions:

- *Input parsing*. Data to be loaded into the XFA Data DOM or provided by a user may be formatted in a style unique to the locale (a *locale-dependent format*) or in canonical format. Before data is loaded into the XFA Data DOM, the XFA processing application converts the data into canonical format, a process called *canonicalization*.
- *Output formatting*. After data is in the XFA Data DOM, there is eventually a need to display, save or print the canonical data in a locale-specific format. The process of converting the canonical data into a locale-specific form is called *localization*.

Requirements for Localization

Satisfying Locale-Dependent User Expectations

Users in different locales have varying expectations about the formats in which they expect to provide dates, times, numbers, and currencies. Such differences reflect ordering of information (for example, MM/DD/YY as opposed to DD/MM/YY), the names used for units (for example, January as opposed to janvier), and the characters/ideographs used to represent information (for example, 1998 as opposed to 一九九八).

Internationalized applications take into consideration varying user expectations regarding dates, times, numbers, and currencies. Such applications present and accept information in a locale-specific format that uses familiar characters or ideographs.

Specifying the Locale to Use During Localization and Canonicalization

The XFA template allows template designers to specify the locale to use for localization and canonicalization. If locales are specified, they override whatever locale database is in use. This makes it possible to correct database errors. It also makes it possible to preserve the state of a changing locale at the time the data was provided, so that for example a document originally giving a price as 100₣ (100 Francs) does not suddenly change to €100 (100 Euros) just because the Euro cutover date has been reached and the locale database updated. And it makes it possible to define custom locales to deal with special requirements.

Note: The XFA grammar does not support the convention `xml:lang`. If `xml:lang` appears in an XFA template, the template fails to load. If it appears in data supplied to the XFA processing application, it is ignored.

Locales are identified by a language code and/or a country code. Usually, both elements of a locale are important. For example, the names of weekdays and months in English Canada and in the United Kingdom are formatted identically, but dates are formatted differently. So, specifying an English language locale would not suffice. Conversely, specifying only a country as the locale may not suffice either — for example, Canada, has different date formats for English and French.

Locale may be specified at multiple levels in an XFA template, as shown in the following example. The sequence following this example explains how the prevailing locale for a field is determined; and the table on page page 140 provides examples of data localized by these locales and picture clauses.

Example 4.34 Template specifying locales

```
<template name="LocaleSpecificationExample">
   <subform name="SubformA" locale="en_CA"> <!--English as spoken in Canada-->
      <field name="FieldA" locale="fr_CA" …> <!--French as spoken in Canada-->
         <bind …>
            <picture …>
               "date(de_CH){D. MMMM YYYY}" <!--German as spoken in Switzerlnd-->
            </picture>
         </bind>
         <ui>
            <picture …>
               "D MMMM YYYY" <!--Inherits French as spoken in Canada-->
            </picture>
         </ui>
      </field>
      <field name="FieldB" …>
         <ui>
            <picture …>
               "MMMM DD, YYYY" <!--Inherits English as spoken in Canada-->
            </picture>
         </ui>
      </field>
   </subform>
</template>
```

An XFA processing application determines the locale to localize/canonicalize a specific field (called the *prevailing locale*) by examining the following sources, in order:

1. Explicit declaration in the picture clause, for example `date(fr){DD MMMM, YYYY}`.
2. Template field or subform declarations, using the `locale` property.
3. Ambient locale. *Ambient locale* is the system locale declared by the application or in effect at the time the XFA processing application is started. In the event the application is operating on a system or within an environment where a locale is not present, the ambient locale defaults to English United States (`en_US`).

Canonicalization of dates, all of which parse to 20041030

Input data (in locale-format)	Field name	Picture Clause	Locale
`"30. Oktober 2004"`	FieldA	`D. MMMM YYYY`	de_CH
`"30 octobre 2004"`	FieldA	`D MMMM YYYY`	fr_CA
`"30 October 2004"`	FieldB	`MMMM D, YYYY`	en_CA

About the Canonical Format Used in the Data and Form DOMs

Inside the Data and Form DOMs, data is always stored in a *canonical format*, in which date, time, date-time, and numbers are represented in a standard (canonical) format ("Canonical Format Reference" on page 887). For example, the canonical format for dates are as follows:

```
YYYY[MM[DD]]
YYYY[-MM[-DD]]
```

In the above examples, letters represent the numeric values described below, square brackets delineate optional parts, and dashes (-) represent literals.

Symbol	Meaning
YYYY	Zero-padded 4-digit year.
MM	Zero-padded 2 digit (01-12) month of the year.
DD	Zero-padded 2 digit (01-31) day of the month.

About Picture Clauses

A picture clause is a sequence of symbols (characters) that specify the rules for formatting and parsing textual data, such as dates, times, currency, and numbers. Each *symbol* is a place-holder that typically represents one or more characters that occur in the data. Some picture clause symbols are locale-sensitive and some are not. Hence, by specifying the appropriate operators the form creator fully controls localization on a field-by-field, and even character-by-character, basis.

When canonicalizing input data, a particular picture is used only if it matches the pattern of the input data. When the picture clause contains multiple alternate picture clauses, the data is compared to each pattern in sequence, until one is found that matches. If none of them match, the data is treated as already in canonical format and copied verbatim into the DOM. This conservative approach prevents data in an

unexpected format from being garbled, including data which unexpectedly arrives already in canonical format. Conservatism is desirable here because canonicalization often involves throwing away characters.

Limitations in Picture Clauses

While this solves some problems, it does not address every need. For example, the interchange date and time format is based on the Gregorian calendar. It would be possible to do conversions to and from other calendars, but locale support on most platforms does not go this far. Hence in this version of XFA only the Gregorian calendar is supported. Another limitation is that times may be entered through the UI without any accompanying date or time zone. Such a time is inherently ambiguous. When a user enters a time without time zone information, the application supplies the time zone from the prevailing locale.

A more fundamental limitation applies to currencies, namely that there is no way to automate conversions between currencies. Currency exchange rates are constantly fluctuating and in any case the appropriate rate varies depending on circumstances (a retail banking customer can't get the same conversion rate as a financial institution can). Hence currency conversions should not and can not be done automatically. Therefore locale can be used for simple currency formatting and parsing but it entirely is up to the creator of the form and designer of the web service or data file to arrange for monetary amounts to be computed in the appropriate currency.

Defining Locales

An XFA form may carry with it a set of locale definitions. These are carried in the `localeSet` packet. See "The localeSet Element" on page 150 for more information about the `localeSet` packet.

It is not necessary to include a definition for `en_US` (United States of America English) because it is the default for XML and consequently is available and thoroughly tested on all platforms. On the other hand including a locale definition this way makes the definition accessible to scripts for reading via SOM expressions. Built-in locale definitions are not accessible this way, although of course the locales they define can still be used.

Caution: Scripts within the form must not alter locale definitions. The result of any such attempt by a script is undefined.

Run-time locale definition

Starting with XFA 2.5, additional grammar is provided for a form to allow an external document to override the definition of locale(s) at runtime. This allows the form to automatically incorporate updates which do not require changes to the form logic, for example if the national unit of currency is renamed.

For each XFA agent there is a property in the Config DOM which may supply a URI pointing to a locale database. For more information about the syntax see `localeSet` in the "Config Specification". If this property is non-empty the URI points to an XDP document containing a `localeSet` packet. When the XFA processor needs to obtain the definition of a locale, it searches locale databases in the following order:

1. The locale database packaged with the form itself in the `localeSet` packet. This always takes precedence so that a form creator who wants the form to always use the same locale definition can be certain it will.

2. The external locale database, if any, pointed to by the Config DOM.

3. The built-in default locale database.

The database in the form or external document may be a partial specification, that is it may omit elements or attributes. When this happens the locale definition inherits the unspecified properties from corresponding but progressively more generic locale(s). The following list shows the progression of inheritance.

1. Matching *language_country_modifier*.
2. Matching *language_country*.
3. Matching *language*.
4. Generic root locale whose values are essentially US English.

This facility is intended for use on servers in automatic workflows, not in forms distributed widely for interactive use, and is restricted accordingly. XFA forms may be packaged inside XDP or PDF files. However when an XFA form is packaged inside a PDF file the form must not refer to an external locale database. Instead locale definitions that are used by the form must be fully resolved and the resolved locale database must be included in the PDF file. This preserves the portability and permanence of the PDF file and closes a security hole. Without this an attacker could use a network redirection attack (such as DNS spoofing) to substitute a modified locale dictionary, changing the currency symbol so that the amount indicated would be different from the amount approved. For additional information see "Structuring Forms for Portability and Archivability" on page 480.

Note: Acrobat by design ignores the `localeSet` property in the Acrobat section of the Config DOM.

It is not practical to include all locales ever defined because the definitions take about 3 kilobytes apiece and there are a great many of them. It is more efficient to include only those that might be used.

The following example shows an external locale database that defines only those properties which it needs to override. For German-language locales it replaces month and day names with uppercase and lowercase Roman numerals. For English-US locales it changes the default format for dates.

Example 4.35 Locale database

```
<?xml version="1.0" encoding="UTF-8"?>
<xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/">
   <localeSet xmlns="http://www.xfa.org/schema/xfa-locale-set/2.1/">
      <locale name="de" desc="German">
         <calendarSymbols name="gregorian">
            <monthNames abbr="1">
               <month>I</month>
               <month>II</month>
                  ...
               <month>XII</month>
            </monthNames>
            <dayNames abbr="1">
               <day>i</day>
                  ...
               <day>vii</day>
            </dayNames>
         </calendarSymbols>
      </locale>
      <locale name="en_US" desc="English (US)">
         <datePatterns>
            <datePattern name="med">D MMM YYYY</datePattern>
            <datePattern name="short">DD/MM/YY</datePattern>
```

```
        </datePatterns>
      </locale>
   </localeSet>
</xdp:xdp>
```

Assume that the above document is available at the URI `http://example.com/myproject/locale.xdp`. The configuration document enables the use of this locale database for an agent named `payroll` as follows.

Example 4.36 Configuration packet for payroll agent

```
<config xlmns="http://www.xfa.org/schema/xci/1.0/">
   <agent name="payroll">
      <common>
         <localeSet>http://example.com/myproject/mylocale.xdp</localeSet>
            ...
      </common>
      ...
   </agent>
</config>
```

Selecting Between Alternate Picture Clauses

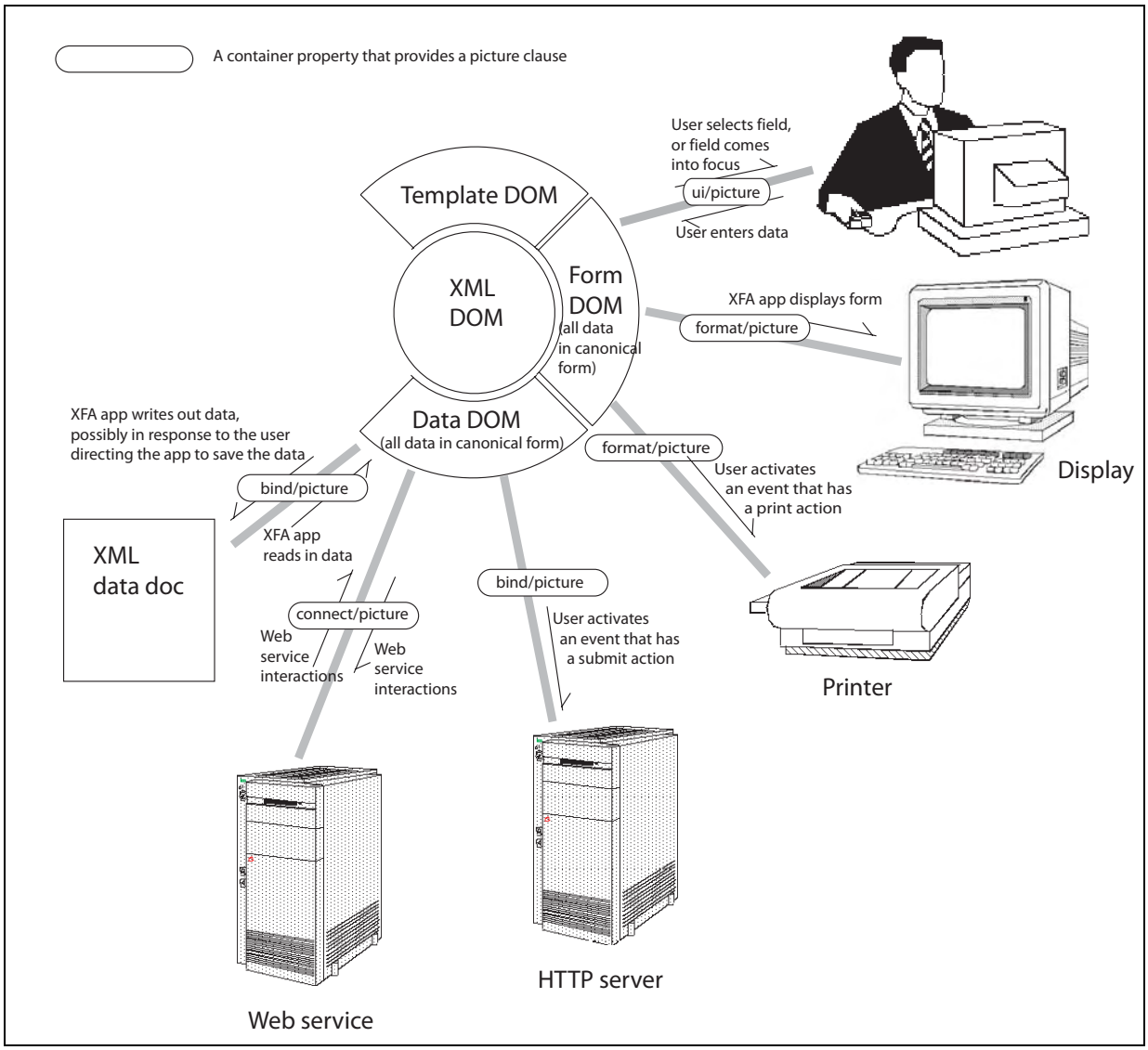
Template designers can create picture clauses that support multiple locales. Such picture clauses contain a series of alternate picture clauses, each of which specifies a locale. ("Picture Clause Specification" on page 991) This technique is useful only for canonicalizing data (during data loading or input parsing). That is, when canonical data is localized using a set of alternate picture clauses, only the first picture clause is used.

Dataflow Paths for Localizing and Canonicalizing Data

The XFA Data DOM and Form DOM maintain their contents in canonical format. Data supplied to an XFA processing application may appear in localized or canonical format. The XFA processing application ensures such data is in canonical format before adding it to the Data DOM or the Form DOM. Later, data in the XFA Data or Form DOMs may be localized before being presented in a human-readable form.

The diagram below illustrates the situations in which data is localized and canonicalized. It also shows the picture clause elements influence conversions for each type of situation and explains in general what triggers such conversions. The table on page 145 further describes the different picture clause-containing elements that influence localization and canonicalization.

"The localeSet Element" on page 150 describes the default picture clauses used when a picture clause element is not defined for a field.



Dataflow paths for localizing and canonicalizing data

Picture clauses and their role in output formatting (localization) and input parsing (canonicalization)

Picture clause parent element (Alternate name)	**Output formatting**	**Input parsing**	**Role of picture clause**
`ui` (edit pattern)	✔	✔	User modification of a data value. For output formatting (localization), the picture clause specifies the format used when the container comes into focus (is selected). For input parsing (canonicalization), the picture clause specifies the format expected from the user. `<field name="field1" … > <ui> <picture> … </picture> </ui> </field>`
`format` (output pattern)	✔		Display or print of data. In the case of display, this form prevails only if the field is not currently in focus (selected). `<field name="field1" … > <format> <picture> … </picture> </format> </field>`
`connect`	✔	✔	Web Services interactions. `<field name="field1" … > <connect> <picture> … </picture> </format> </field>`

Picture clauses and their role in output formatting (localization) and input parsing (canonicalization)

Picture clause parent element (Alternate name)	**Output formatting**	**Input parsing**	**Role of picture clause**
`validate`		✔	Specifies a test applied to data in the Form DOM, where the test verifies the data complies with the given picture clause. Generally, the validation picture clause reflects canonical format and is locale-agnostic because all data in the Form DOM is represented in canonical format. The validation test must succeed before the XFA processing application updates the Data DOM with the value from the Form DOM. `<field name="field1" … > <validate> <picture> … </picture> </validate> </field>` The validate picture clause is used for testing, not for conversion.
`bind`	✔	✔	XML data file or submitting data to a server. `<field name="field1" … > <bind> <picture> … </picture> </bind> </field>`

Rules for Localizing Data

The following rules determine which data can be canonicalized and/or localized. Data not covered by those rules is copied unmodified into and out of the XFA Data DOM. The diagram "Dataflow paths for localizing and canonicalizing data" on page 144 illustrates how different picture clauses are used for different localization/canonicalization paths.

This section describes the following rules:

➤ Rule 1, Non-Localizable Data

Non-textual data (such as values defined with the `arc`, `image`, or `rectangle` elements) is not localizable. For example, the image defined in the following fragment cannot be localized.

Example 4.37 Image is not localizable

```
<field …>
   <value …>
      <image> … </image>
   </value>
</field>
```

➤ Rule 2, Input Parsing a Data Value Using a Picture Clause

Text entered into a field with an `bind`, `ui`, or `connect` picture clause is canonicalized per the picture clause, provided it matches the picture clause. If any of the following conditions are true, the text is assumed to have been entered in canonical format:

- Supplied data does not match the picture clause
- Picture clause omitted
- Picture clause is defective

For example, assume a field has an input picture clause and a `textEdit` widget, as follows.

Example 4.38 Field using an input picture clause

```
<field name="field1" … >
   <ui>
      <picture>9,999.99</picture>
      <textEdit … />
   </ui>
</field>
```

Regardless of the widget type, canonicalization on input is controlled solely by the picture clause. Assume that the locale is `fr_FR` and the input text is `"1.234,56"`. This matches the picture clause, so the data is canonicalized into `"1234.56"` and goes into the XFA Data DOM this way. On the other hand, had the input text been `"nicht"`, it would not have matched the picture clause, so it would have been copied literally into the DOM as `"nicht"`. A special case of this occurs if the input text is `"1234.56"`, that is, if it is already canonical. Because this does not match the picture clause (which expects the text to be localized), the already-canonical data is copied directly into the DOM.

The following field accepts a date.

Example 4.39 Date field using an input picture clause

```
<field name="field1" … >
   <ui>
      <picture>D-MMMM-YYYY</picture>
      <dateTimeEdit … />
   </ui>
</field>
```

If our French user enters, for example, `"12-janvier-2004"`, the data is canonicalized to `"20040112"`.

A picture clause may contain multiple pictures. In the following example two different date formats are recognized and canonicalized automatically.

Example 4.40 Field using multiple input picture clauses

```
<field name="field1" … >
   <ui>
      <picture>D-MMMM-YYYY|D-M-YYYY</picture>
      <dateTimeEdit … />
   </ui>
</field>
```

If the user enters either `"12-janvier-2004"` or `"12-01-2004"` the result is the same. The input text matches the first or second picture, respectively, and either way is canonicalized to `"20040112"`.

➤ Rule 2A, Input Parsing a Without a Picture Clause May Use Default Picture Clauses for Current Locale

If the field omits a picture clause but describes data that is traditionally localized, the data is input parsed using default picture clauses for the locale. For example, if the user provides a date value for the field in the following example and the current locale is `fr_FR`, the data is input parsed against the default date picture clause for French speakers in France. The locale-dependent picture clauses are specified in the locale set grammar, "Locale Set Specification" on page 794. For example, if the default date picture clause for France is `D-MMMM-YYYY` and the user enters either `"12-janvier-2004"`, the data is canonicalized to `"20040112"`.

Example 4.41 Field using a default picture clause

```
<field name="field1" … >
   <ui>
      <dateTimeEdit … />
   </ui>
</field>
```

➤ Rule 3, Output Formatting

Text in a field with an output picture clause is formatted for display per the picture clause when the field is redrawn (including when the user tabs out of the field). This does not affect the content in the DOM.

For example, assume a field has a format picture clause as follows.

Example 4.42 Field using an output picture clause

```
<field name="field1" …>
   <format>
      <picture>$9,999.99</picture>
   </format>
</field>
```

Assume further that the locale is `fr_FR` and the content of the field in the DOM is `"1234.56"`. When the user tabs out of the field, the field is re-displayed as `"€1.234,56"`.

The following field contains a date.

Example 4.43 Date field using an output picture clause

```
<field name="field1" … >
   <format>
      <picture>D-MMMM-YYYY</picture>
   </format>
</field>
```

If the content of the field `"20040112"`, the data is displayed or printed as `"12-janvier-2004"`.

If the picture clause contains more than one picture, the first picture is used and the others ignored.

➤ Rule 4, Output Formatting When Output Picture Clause Omitted

The following table describes how an XFA processing application localizes data whose container omits the picture clause for a particular type of output.

Type of picture clause	How unpictured data is formatted
`connect`	Canonical format.
`bind`	Canonical format.
`ui` and `format` (edit and output patterns)	Formatted as follows:

Content element name	Formatted with …
`date`	Default date picture clause for the locale (usually the medium date), as described in "The localeSet Element" on page 150
`time`	Default time picture clause for the locale (usually the medium time), as described in "The localeSet Element" on page 150
`datetime`	Default date picture clause and default time picture, separated with a "T".
`decimal` and `float`	Default decimal radix (decimal point) for the locale, as described in "The localeSet Element" on page 150. Separators are not inserted.

➤ Rule 4A, Output Formatting a Null Value

In the following example, a field is defined with a null `float` value.

Example 4.44 Field with a null default value

```
<field name="field1" … >
   <value>
      <float/>
   </value>
</field>
```

If the float element had contained a value, it would have been the default value for the field. When empty as in this case, there is no default value; however, the type declaration still applies. Thus, the field qualifies for automatic localization.

➤ **Rule 5, FormCalc Scripts May Localize/Canonicalize Data in the XFA Data DOM**

Scripts written in FormCalc can call functions to localize or canonicalize specific data in the XFA Data DOM, as discussed in "FormCalc Specification" on page 891. ECMAScript does not have corresponding methods.

The localeSet Element

For each locale definition in the `localeSet` element, the string of substitute characters is contained in the `dateTimeSymbols` element.

The following example illustrates the overall structure of an XDP containing a `localeSet` packet.

Example 4.45 A typical localeSet packet

```
<?xml version="1.0" encoding="UTF-8"?>
<xdp xmlns="http://ns.adobe.com/xdp/">
  <localeSet xmlns="http://www.xfa.org/schema/xfa-locale-set/2.5/">
  <!-- Start of a locale definition. -->
    <locale name="fr_FR" desc="French (France)">
      <typeFace name="Times Roman"/>
      <calendarSymbols name="gregorian">
        <monthNames>
          <month>janvier</month>
          <month>février</month>
          <month>mars</month>
          <month>avril</month>
          <month>mai</month>
          <month>juin</month>
          <month>juillet</month>
          <month>août</month>
          <month>septembre</month>
          <month>octobre</month>
          <month>novembre</month>
          <month>décembre</month>
        </monthNames>
        <monthNames abbr="1">
          <month>janv.</month>
          <month>févr.</month>
          <month>mars</month>
          <month>avr.</month>
          <month>mai</month>
          <month>juin</month>
          <month>juil.</month>
          <month>août</month>
          <month>sept.</month>
          <month>oct.</month>
          <month>nov.</month>
          <month>déc.</month>
        </monthNames>
        <dayNames>
```

```
      <day>dimanche</day>
      <day>lundi</day>
      <day>mardi</day>
      <day>mercredi</day>
      <day>jeudi</day>
      <day>vendredi</day>
      <day>samedi</day>
   </dayNames>
   <dayNames abbr="1">
      <day>dim.</day>
      <day>lun.</day>
      <day>mar.</day>
      <day>mer.</day>
      <day>jeu.</day>
      <day>ven.</day>
      <day>sam.</day>
   </dayNames>
   <meridiemNames>
      <meridiem>AM</meridiem>
      <meridiem>PM</meridiem>
   </meridiemNames>
   <eraNames>
      <era>av. J.-C.</era>
      <era>ap. J.-C.</era>
   </eraNames>
</calendarSymbols>
<datePatterns>
   <datePattern name="full">EEEE D MMMM YYYY</datePattern>
   <datePattern name="long">D MMMM YYYY</datePattern>
   <datePattern name="med">D MMM YYYY</datePattern>
   <datePattern name="short">DD/MM/YY</datePattern>
</datePatterns>
<timePatterns>
   <timePattern name="full">HH' h 'MM Z</timePattern>
   <timePattern name="long">HH:MM:SS Z</timePattern>
   <timePattern name="med">HH:MM:SS</timePattern>
   <timePattern name="short">HH:MM</timePattern>
</timePatterns>
<dateTimeSymbols>GaMjkHmsSEDFwWxhKzZ</dateTimeSymbols>
<numberPatterns>
   <numberPattern name="numeric">z,zz9.zzz</numberPattern>
   <numberPattern name="currency">z,zz9.99 $</numberPattern>
   <numberPattern name="percent">z,zz9%</numberPattern>
   </numberPatterns>
<numberSymbols>
   <numberSymbol name="decimal">,</numberSymbol>
   <numberSymbol name="grouping"></numberSymbol>
   <numberSymbol name="percent">%</numberSymbol>
   <numberSymbol name="minus">-</numberSymbol>
   <numberSymbol name="zero">0</numberSymbol>
</numberSymbols>
<currencySymbols>
   <currencySymbol name="symbol">€</currencySymbol>
   <currencySymbol name="isoname">EUR</currencySymbol>
   <currencySymbol name="decimal">,</currencySymbol>
```

```
        </currencySymbols>
      </locale>
    <!-- Start of a locale definition. -->
      <locale name="en_GB" desc="English (United Kingdom)">
        …
      </locale>
      …
    </localeSet>
</xdp>
```

The numeric grouping character for `fr_FR` is a non-breaking space, depicted here as "". In the actual XML file all characters are encoded using Unicode code points and UTF-8 encoding as specified by [XML 1.0].

Note: The `localeSet` element and all of its contents reside in the namespace `http://www.xfa.org/schema/xfa-locale-set/2.5/`.

The order in which the locales appear is not significant. The information for each locale consists of several sections, which may be present in any order. If any of the sections is omitted, or an element or attribute omitted from within a section, the effect is to select the corresponding default value.

Calendar symbols

This section supplies the names for months of the year and days of the week (both the full names and the abbreviated names). It also supplies the names for modifiers equivalent to A.D. and B.C., A.M. and P.M. The placement of these names and modifiers is determined by the date, time, or date-time picture clause in use.

Date and time symbols

This section supplies a vector of character mappings for use in generating localized prompts, as described above under "FormCalc and Locale".

Date patterns

This section supplies picture clauses for four standard date formats. The formats are distinguished by verbosity ranging from `full` to `short`.

Time patterns

This section supplies picture clauses for four standard time formats. The formats are distinguished by verbosity ranging from `full` to `short`.

Currency symbols

This section supplies the characters to be used for the currency symbol and the currency radix. It also supplies the string to be used for the ISO currency name. The placement of these symbols within currency amounts is determined by the numeric picture clause in use.

Number patterns

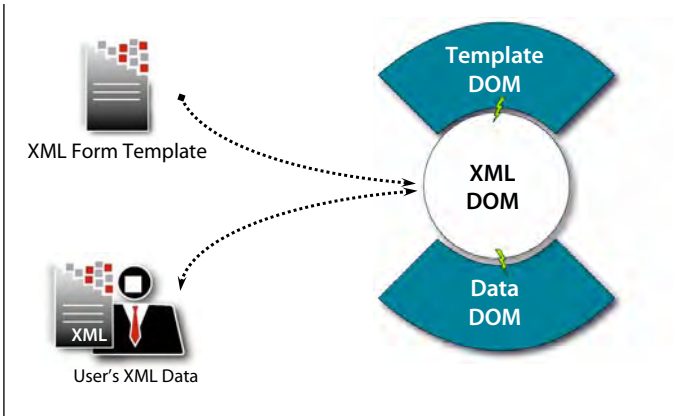
This section supplies picture clauses for four standard number formats. There are only three picture clauses because two number formats (`decimal` and `integer`) use the same `numeric` picture clause; the `integer` format uses only the integer part of the numeric clause whereas `decimal` uses the whole thing.

Number symbols

This section supplies the characters to be used for the non-currency decimal radix, grouping separator, percentage sign, and minus sign. The placement of these characters within numbers is determined by the numeric picture clause in use.

Loading a Template to Produce the XFA Template DOM

Loading an XFA Template DOM is a two-step process, which involves first creating an XML Template DOM and from that creating an XFA Template DOM.



Template loading produces an XFA Template DOM

Creating an XML Template DOM

The XML Template DOM is filled with the complete content of the XML Form Template in the usual manner for XML DOMs.

Creating an XFA Template DOM

A subset of the contents of the XML Template DOM is copied into the XFA Template DOM. This subset excludes:

- elements and attributes which are not in the XFA template namespace;
- elements and attributes which are not recognized by the XFA processor as part of the template schema;
- elements which do not belong in the current view, for example an element that is marked as `relevant` only to printing but the current application is interactive.

It is an error for the XML Template DOM to contain elements or attributes in the XFA template namespace that are not part of the template schema, but it is not a fatal error. A warning message may be generated but processing continues. It is not an error for the other types of excluded content to be present.

The nodes of the XFA Template DOM are peered to the nodes of the XML Template DOM to allow the XFA processor to make changes to the XFA Template DOM and then save the modifed template as a new XML Form Template.

After this the XFA processor resolves prototypes. For each prototype reference it copies content from the referenced prototype into the XFA Template DOM. Some prototypes may be in external XML Form Templates, so resolving prototypes may involve creating and loading additional transient XML Template DOMs and corresponding transient XFA Template DOMs. The transient DOMs are retained during subsequent prototype processing so that subsequent references to the same document can be processed without reloading. (This is not only for performance reasons. It also protects against loss of internal consistency should the external document be modified during processing.) Once all prototypes have been resolved, including nested prototype references, the transient DOMs are deleted.

Supporting Template-Creator Stamps

Templates may include information that uniquely identifies their creator and when they were last modified. XFA processing applications are required to respect this information as described in "Tracking and Controlling Templates Through Unique Identifiers" on page 460.

Basic Data Binding to Produce the XFA Form DOM

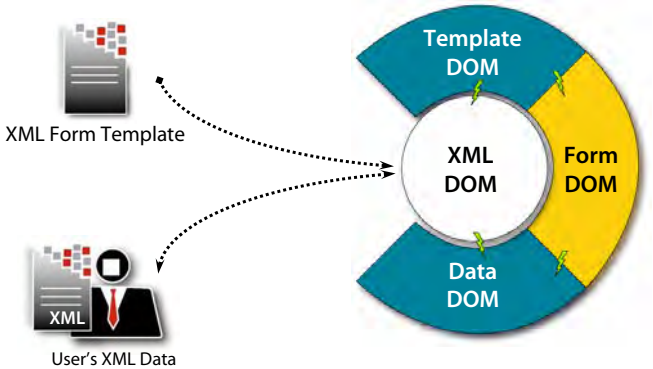
This section explains how data is bound to specific nodes within a Form DOM and how disconnects in naming, types, and level are handled. This section describes data binding only for static forms, which are forms that have a pre-defined number of subforms. Data binding for dynamic forms is described later in the chapter "Dynamic Forms" on page 286.

The reader is assumed to be familiar with the principles and vocabulary of XML, as set out in *Extensible Markup Language (XML) 1.0* [XML]. The reader is also assumed to be familiar with the following concepts:

- Overall principles of XFA processing of user data, as described in "Creating, Updating, and Unloading a Basic XFA Data DOM" on page 108
- Structure of an XFA form template, as described in "Template Features for Designing Static Forms" on page 29
- SOM expressions (including scope matching), as described in "Scripting Object Model" on page 73

About Data Binding

Within XFA applications the template is instantiated as a tree-structured set of nodes called the Template Data Object Model (Template DOM). Similarly, user data is instantiated as a tree-structured set of nodes called the XFA Data Object Model (XFA DOM). The XFA Data DOM is further subdivided into one or more non-overlapping subtrees, with each subtree representing a record. Data binding is the process by which nodes in the Data DOM (data nodes) representing one record are associated with nodes in the Template DOM (data nodes). The result is a new DOM (the Form DOM) which embodies the association. Although the content of the Form DOM is copied from the Data DOM, its structure (arrangement and hierarchy of nodes) is based mainly upon the Template DOM.



Binding the Data DOM to the Template to produce the Form DOM

However under some circumstances a node is created in the Form DOM which does not match any existing note in the Data DOM. When this happens, if there is a data description, the data description is consulted to determine what structure the data would have had if it had been there. Then processing proceeds as though all that structure was present in the Data DOM, except of course that the node(s)

being added to the Form DOM are not associated with any nodes in the Data DOM. By contrast if there is no data description then the new structure in the Form DOM is based purely on the Template DOM.

Note: The presence of a data description does not by itself guarantee that the Data DOM will conform to the data description at the end of the data binding process. To ensure conformance after binding, the data must start off conforming to, or being a subset of, the data description. In addition the structure of merge-able nodes in the template must correspond to the data description or to a super set of it. However these restrictions are not imposed or enforced by the data binding process. It does not check for conformance and has no problem dealing with non-conforming data or indeed with the absence of a data description.

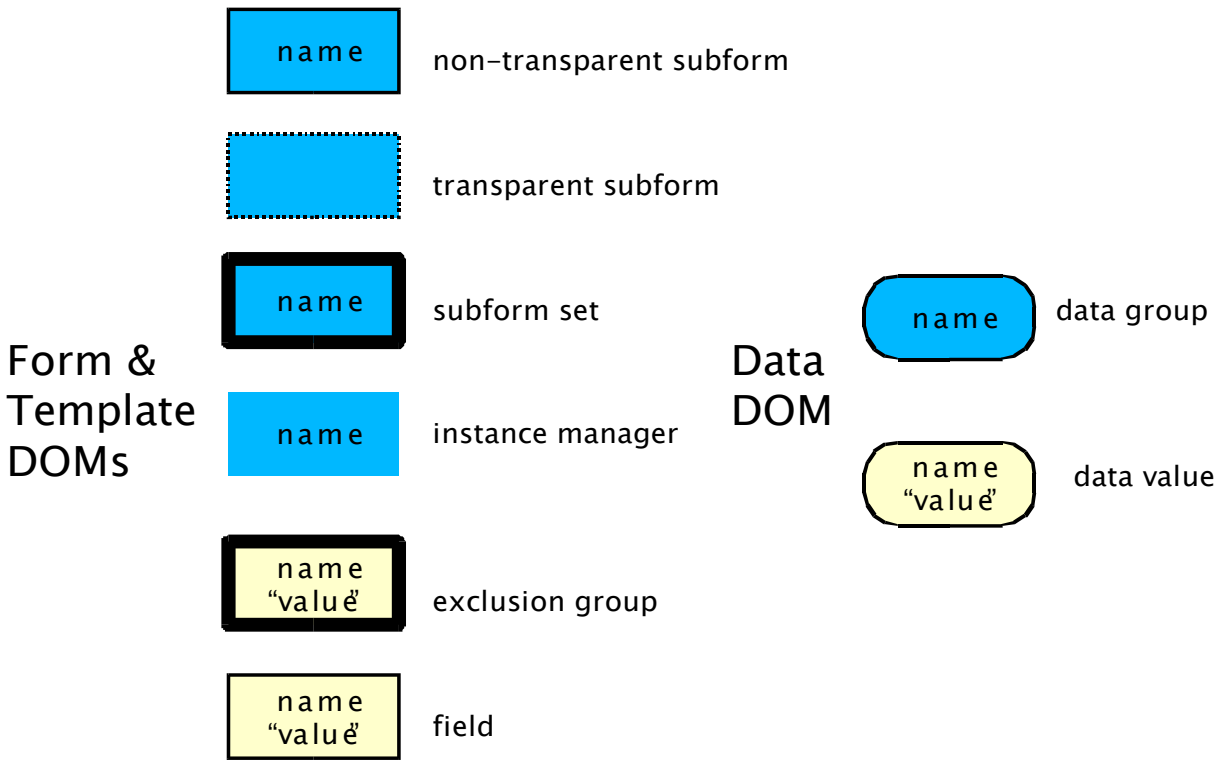
Optionally, after all other processing on a record is complete, the data binding process may adjust the Data DOM to make it exactly parallel the structure of the Form DOM. This forces the data into a shape imposed by the template and the data description. Use of this option precludes round-tripping.

Data binding is also known as *merging* because it can be thought of as merging the data (and possibly data description) with the template.

A variant known as an *empty merge* does not use data from the Data DOM. In this case, the Form DOM is created based on the template and the data description. Optionally, default data matching the Form DOM may be inserted into the Data DOM.

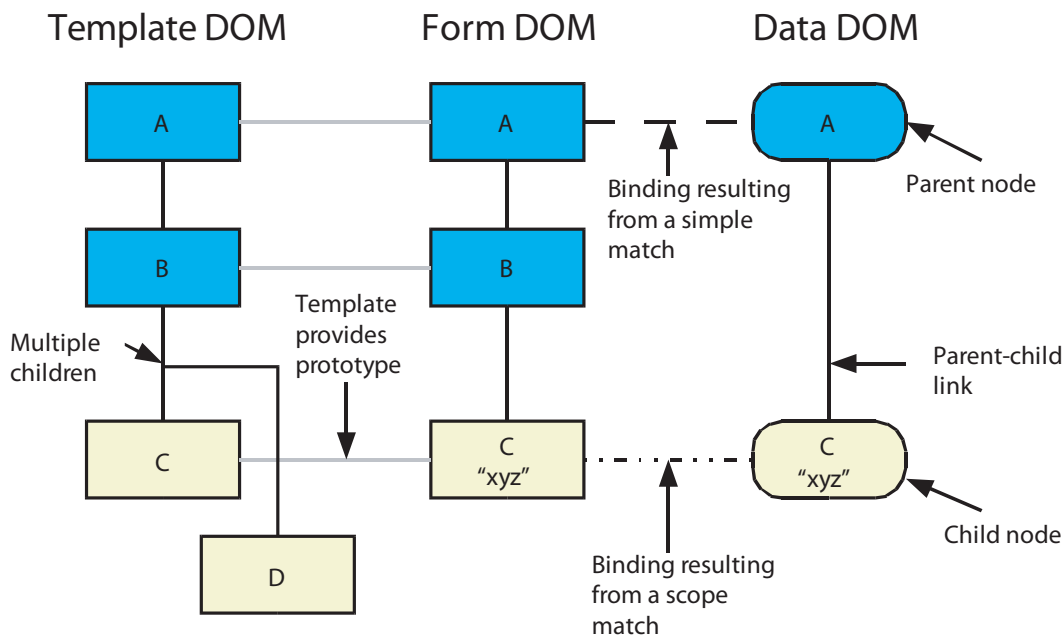
Conventions

Many drawings in this section depict relationships between nodes in tree graphs. Nodes are depicted using different shapes and shadings, as shown in the following figure.



Key to node symbols

The relationships between nodes can be of several types. The following diagram shows the depiction of different types of relationships. A node can have multiple children. The ordering of the children is significant. The children are in order of decreasing age (that is, from the first child added to the last child added) from left-to-right and top-to-bottom. As with English text, top-to-bottom has a higher priority than left-to-right. In the same way, the words on a line of text precede the words on the next line of text, even if the words on the next line are to the left of the words on the first line.



Key to node interconnects

Within the Template DOM and Data DOM nodes are loaded in document order, so the ordering of nodes by age is also the ordering by document order. To traverse any portion of the Template DOM or Data DOM in document order, start at the topmost node and perform a depth-first descent of the tree, descending from each node through its eldest child first, then upon returning to that node descending through the next-eldest child, and so on. In the above figure, document order for nodes in the Template DOM is A-B-C-D. The corresponding XML would follow the following pattern.

Example 4.46 Template corresponding to the preceding illustration

```
<template …>
   <subform name="A">
      <subform name="B">
         <field name="C"> … </field>
         <field name="D"> … </field>
      </subform>
   </subform>
</template>
```

Data binding fills the Form DOM by copying individual nodes from the corresponding prototype nodes in the Template DOM. In diagrams this relationship is shown by the dashed lines between the prototype nodes in the Template DOM and the corresponding nodes in the Form DOM. Often nodes in the Form DOM are bound to nodes in the Data DOM. When this happens diagrams show the value in the data node duplicated in the form node. In addition the binding is shown with either a dashed line or a dot-dashed line. The two types of line represent bindings that occur for different reasons. The different types of binding correspond to the different types of matches to SOM expressions, dashed lines to direct matches

and dot-dashed lines to either indirect or scope matches. The types of matches are explained in "Scripting Object Model" on page 73.

Principles of Data Binding

Data binding supports data independence, which is a key XFA feature. *Data independence* allows a form designer to change a template without requiring that corresponding changes be made to the structure of the data. It also allows the same data to be used with different templates. Data independence is enhanced by retaining the original structure of the XML Data DOM and the values of any objects in the XML Data DOM that are not used by the template.

Data binding is sometimes called *merging* or the *merge process*. This refers to the goal of data binding, which is to merge the data with the template. Data binding achieves this by creating the Form DOM, which instantiates the mapping between the data and the template. The Form DOM is actively linked to the XFA Data DOM so that changes made to data values in the Form DOM propagate automatically into the Data DOM and vice-versa. The Form DOM can be thought of as the filled-in form. Indeed, scripts usually act upon the Form DOM rather than dealing directly with either the Template DOM or the XFA Data DOM.

Extraneous data may be present in the Data DOM; it is preserved but does not take part in the mapping. Likewise subforms may optionally be included in the Form DOM even though unmatched by data; the fields within such subforms receive default values. The Form DOM may also be organized differently from the incoming data, with content reordered and/or redistributed into subforms. Hence, some changes can be made to the organization and content of the template independently of the format of the incoming data. Some changes can also be made to the data independently of the template. The binding algorithm handles such changes without any need for scripting.

The exact behavior of the data-binding process is defined in later sub-sections of this specification. Here is a simplified overview:

The data binding process walks through the Template DOM and Data DOM, populating the Form DOM with nodes. If a Data DOM was supplied, the data binding process attempts to match up each new form node with a data node in accordance with the following rules:

- Explicit bind targets defined in the template take precedence over automatic bindings.
- For automatic bindings the relative order of same-named data values or groups **is** significant.
- For automatic bindings the relative order of uniquely-named data values or groups **is not** significant.
- For automatic bindings the hierarchy of structure described by data values or groups **is** significant.

The rules for automatic bindings are equivalent to the ones used for resolving SOM expressions. Indeed one way of describing the matching process is that the data binding process attempts to find a data node such that, when the path from the root of the Data DOM to the data node is written as an unqualified SOM expression, the resulting SOM expression matches the form node as seen from the root of the Form DOM. This is explained in more detail later.

The data binding process sometimes adds data nodes but it never deletes any data nodes. In addition it reads but does not modify the Template DOM. On the other hand it populates the Form DOM.

The Bind Element

Each subform, field, and exclusion group object in the Template DOM and Form DOM has a `bind` property, corresponding to the `bind` element in an XML template. The `bind` property contains various

sub-properties controlling the object's behavior during data binding and afterward. The sub-properties are `match`, `picture`, and `ref`.

The match property

This property controls the role played the by parent object in a data binding operation. It must be set to one of the following values:

`once`

The node representing the parent object will bind to a node in the XFA Data DOM in accordance with the standard matching rules. This is the default value.

`none`

The node representing the parent object will not bind to any node in the XFA Data DOM. This is normally used for nodes that are *transient*, that is, that will not be written out if the DOM is saved to a file.

`global`

This is only allowed if the parent object is a field. It signifies that field is capable of binding to global data. If the normal matching rules fail to provide a match for it, the data-binding process will look outside the current record for global data to bind to the field. Note that, whereas a regular data value node can only bind to one field, a single global data value node can bind to many fields.

The current record is always a subtree within the Data DOM. Global data is any data value that is not inside any record but that is at least as high in the hierarchy as a record. See "Creating, Updating, and Unloading a Basic XFA Data DOM" on page 108 for more information.

`dataRef`

The parent object will bind to the node in the XFA Data DOM specified by the accompanying `ref` property.

The picture property

This property specifies the format of data in the Data DOM. When data is copied into the Form DOM the bind picture is used to convert it into *canonical format*. Canonical format for numbers has a `"."` as a decimal point and no thousands separator. Canonical format for dates is a subset of [ISO 8601]. Converting data to canonical format makes it possible to manipulate the data using scripts and applications that are unaware of the local format. When the data is saved to a file, the bind picture is used in the reverse direction to convert the data from canonical format into local format. For more information about localization and canonicalization, see "Localization and Canonicalization" on page 138

When a script reads the `value` property of a form node, the value that it receives is localized using the bind picture. When it assigns to the `value` property of a form node, the value it supplies is canonicalized using the bind picture. In effect the script is treated as though it is a user who wishes to deal in localized formats. However, unlike users, scripts can also read from and assign directly to the canonical format using the `rawValue` property of the form node. In addition scripts can access the original uncanonicalized data, as supplied in the XML data document, using the `value` property of the associated data node.

Note that a bind picture may be supplied using the `picture` child of the `transform` element in the XFA configuration document. A conflict would arise if two bind pictures were supplied for the same form node, one in the template and one in the configuration. It is up to the form creator to ensure that this does not happen.

The ref property

This property is used to explicitly bind a field to a particular data value node, overriding the automatic data-binding process. This property is used only when the accompanying `match` property has a value of `dataRef`. When used, the value of this property must be a fully-qualified SOM expression referring to a data value node in the Data DOM. See "Explicit Data References" on page 177 for more information.

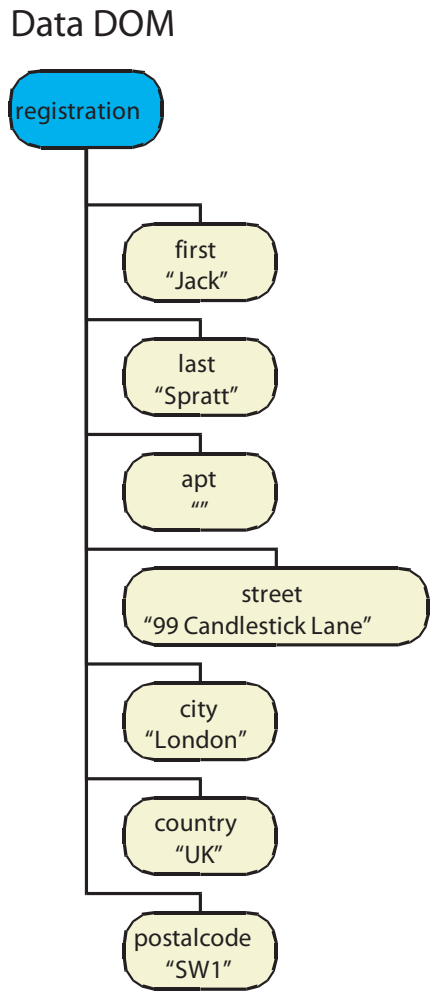
Simple Example of Data Binding

The simplest kind of form has a unique name for each subform or field and for each corresponding data group or data value. For example, suppose an online form is being used to edit the registration information for a registered user. The registration data consists of first and last name and mailing address. The data retrieved from the registration data base is as follows.

Example 4.47 Registration data

```
<?xml version="1.0"?>
<registration>
   <first>Jack</first>
   <last>Spratt</last>
   <apt></apt>
   <street>99 Candlestick Lane</street>
   <city>London</city>
   <country>UK</country>
   <postalcode>SW1</postalcode>
</registration>
```

When the registration data is loaded into the Data DOM the result is as shown (right).



Data DOM with registration data

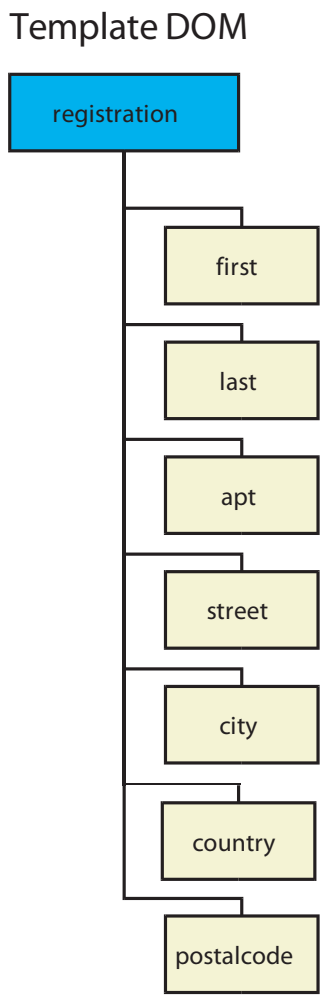
The template was created with field names that match the data elements one-for-one. A highly simplified skeleton of the template follows.

Example 4.48 Registration template skeleton

```
<template …>
   <subform name="registration">
      <field name="first" …>… </field>
      <field name="last" …> … </field>
      <field name="apt" …> … </field>
      <field name="street" …> … </field>
      <field name="city"…> … </field>
      <field name="country"…> … </field>
      <field name="postalcode"…>…
      </field>
   </subform>
</template>
```

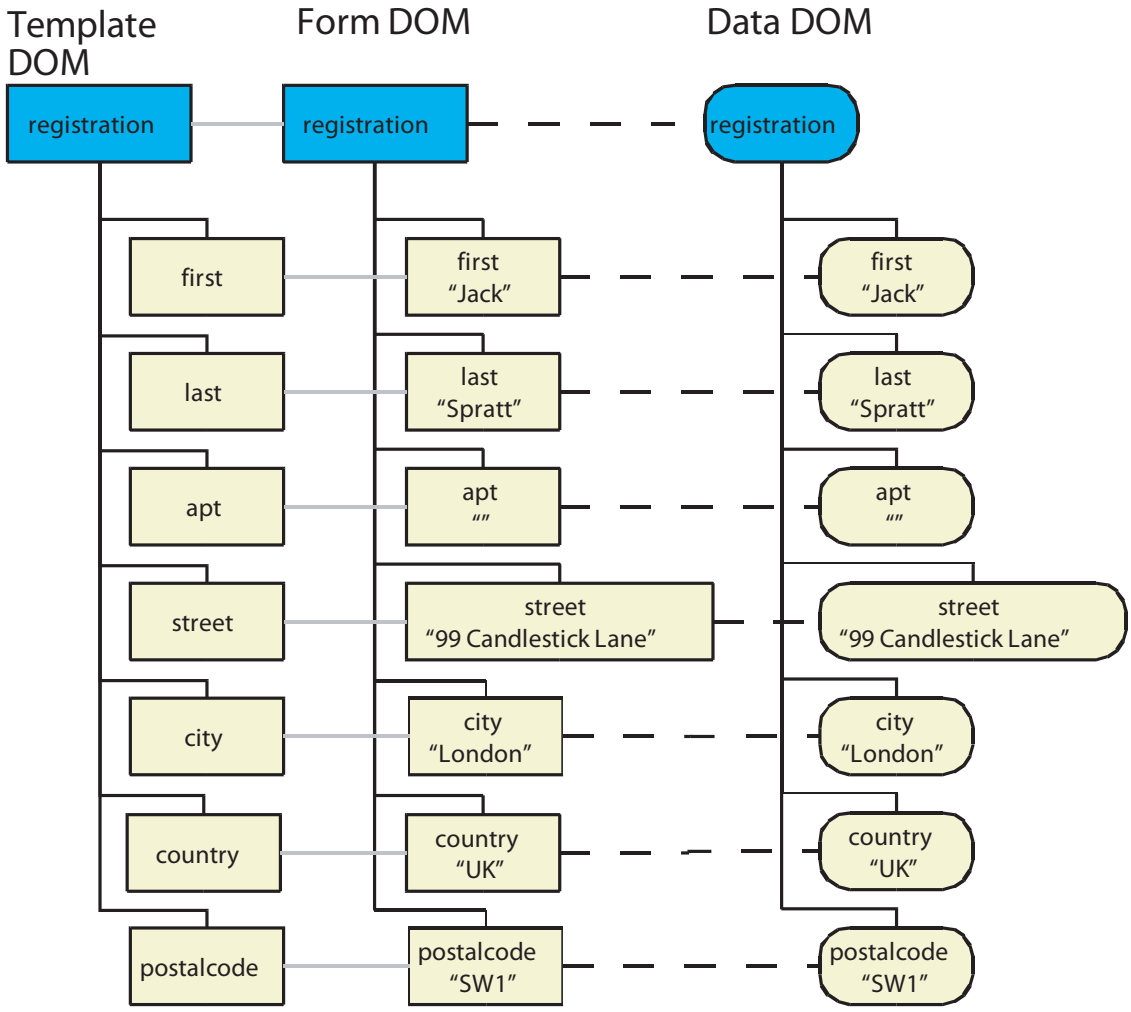
Note that the field names match the data element names in letter-case. This is required because when the data-binding process matches data values with fields it uses a case-sensitive name comparison.

When the template is loaded into the Template DOM the result is as shown (right).



Template DOM with registration template

The Form DOM resulting from this operation represents the binding between data and template. The relationship between the three DOMs is shown below.



Result of binding registration data to registration template

In the above diagram, each node of the Template DOM has acted as a prototype for a node in the Form DOM (shown by the solid grey lines between them). Each node of the Form DOM has been bound to a node in the Data DOM (shown by the dotted black lines between them). When a form field node is bound to a data value node the content of the data node is copied to the form node.

Data Binding Steps

Data binding a simple form such as the one described in the previous section involves the following steps:.

1. Create form nodes as copies of template nodes ("Create Form Nodes and Match with Data Nodes (Steps 1 and 2)" on page 163).
2. Match non-attribute data nodes to form nodes.
3. Match attributes to unmatched form nodes ("Match Attributes (Step 3)" on page 178).

4. Re-normalize (adjust the Data DOM to mirror the Form DOM) ("Re-Normalization (Step 4)" on page 178).
5. Perform calculations and validations ("Calculations and Validations (Step 6)" on page 186).
6. Issue the form ready event ("Form Ready Event (Step 7)" on page 186).
7. If scripts modify the Data DOM after the Form DOM has been created, it may be necessary to wholly or partially repeat the data binding process ("Remerge and Incremental Merge (Step 8)" on page 187).

The following subsections expand on some of the steps above.

Create Form Nodes and Match with Data Nodes (Steps 1 and 2)

Continuing the example from the previous section, Step 1in the data-binding process is simple. Each node in the Template DOM is copied into the Form DOM. Some nodes are not merge-able; for example, draws can not match up with user data, nor do they contain other elements that can, so they are not merge-able.

As each node is copied into the Form DOM, if it is merge-able, it is matched with the same-named data element. (Only merge-able nodes have been shown in the accompanying illustrations.) These are so-called *direct matches* in which, not only do the node names match, but the names of all their merge-able ancestors match in sequence. This corresponds to the logic of SOM expressions; if a data node and a form node directly match, they are both named by the same SOM expression relative to the current record and the top-level subform, respectively. For example, the `city` field in the Template DOM could be expressed by the SOM expression `registration.city` relative to the root of the Template DOM. At the same time the `city` node in the Data DOM could be expressed by the SOM expression `registration.city` relative to the root of the Data DOM. Hence the two nodes match and the data binding process binds them together.

It is important to note that for a data node and a form node to bind together they must be compatible types. A subform can bind to a data group but not a data value. A field can bind to a data value but not a data group.

The highest-level subform and the data node representing the current record are special; they are always bound even if their names don't match. In fact it is common for the highest-level subform in a template to be unnamed, that is to not have a `name` attribute. In the example assume that the data holds just one record (the `registration` data group and its content). This is a common arrangement. In this case, the `registration` data node is the one representing the current record.

If the data was missing some elements, all fields would still be placed into the Form DOM but some field nodes would remain unbound. This corresponds to a paper form that has not been completely filled in. However the template may specify a default values for any field, thereby forcing the field to be initialized with the default value whenever the data does not fill it. Furthermore if a data description is present it may force additional structure to be included. However for this example assume that the data description is not supplied or simply mirrors the structure of the example data.

If the data had extra elements whose names differed from anything in the template, those extra data nodes would simply be left unbound. This is true regardless of the contents of the data description. The resulting Form DOM would in effect represent a subset of the data. Applications can therefore use multiple templates with different template objects to present different views of the same data. In addition many types of template nodes have a `relevant` property which gives a different kind of control. The `relevant` property affects what portions of the template are loaded into the Template DOM by particular applications. For example, a particular element might be marked relevant only to printing. An interactive

client would ignore that element when loading the Template DOM. In this way the same template can present different views in different contexts.

Now suppose that the form designer decides to separate part of the `registration` subform into a separate `address` subform. This might be done in order to make it easier to reuse the `address` subform in other templates. The resulting template has the following skeletal structure.

Example 4.49 Registration template with address subform

```
<template …>
   <subform name="registration">
      <field name="first" …>… </field>
      <field name="last" …> … </field>
      <subform name="address">
         <field name="apt" …> … </field>
         <field name="street" …> … </field>
         <field name="city"…> … </field>
         <field name="country"…> … </field>
         <field name="postalcode"…> … </field>
      </subform>
   </subform>
</template>
```

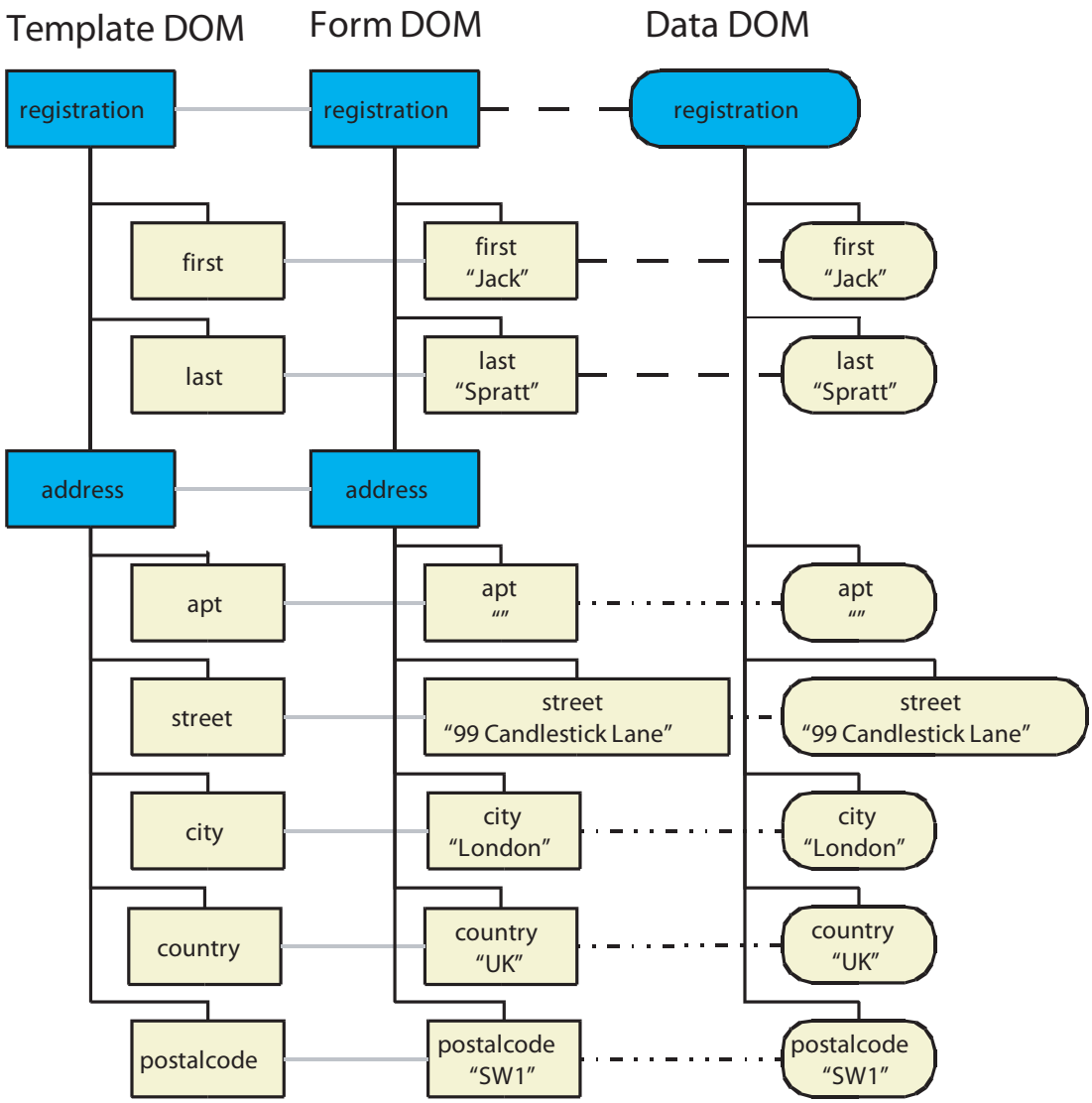
Despite this change to the template, the same data will still bind correctly to the template. Consider the same example data as in Example 4.47, repeated here for convenience.

Example 4.50 The same data works with the modified template

```
<?xml version="1.0"?>
<registration>
   <first>Jack</first>
   <last>Spratt</last>
   <apt></apt>
   <street>99 Candlestick Lane</street>
   <city>London</city>
   <country>UK</country>
   <postalcode>SW1</postalcode>
</registration>
```

The `registration` subform still matches the `registration` data group so they are bound as before. Similarly the data values `first` and `last` still match their respective fields within the `registration` subform. However when the data-binding process reaches the `apt` data value, it finds that there is no direct match. In the absence of a direct match, the data binding process looks for a scope match. A scope match occurs when the data node in question is the sibling of a node which has an ancestor bound to an ancestor of a node with the same name as the data node. In this case, it finds that the `apt` data value is a sibling of the `first` data value, which has an ancestor (the `registration` data group) bound to the `registration` subform, which in turn contains a field named `apt`. Hence the `apt` data value scope matches the `apt` field. (The same thing can be expressed in terms of parallel SOM expressions thus: The SOM expression for the `apt` field is `$form.registration.address.apt`. When applied to the root of the data record this would directly match `$record.registration.address.apt`, but there is no such node. Instead, there is a `$data.registration.apt` which when mapped to `$form.registration.apt` scope-matches `$form.registration.address.apt`.) Therefore the data-binding process copies the `address` subform into the Form DOM, followed by the `apt` field, and binds the field and data value nodes together. By the same logic the `street`, `city`, `country` and

`postalcode` data values are bound to the fields which they scope match in the `address` subform. The result is shown in the following diagram.



Result of registration binding with template changed

In the above example, the data description is not consulted during data binding because a match is found. The data description is only consulted during data binding when there is no match of any kind for the current form node.

Scope matches have been defined to allow changes to be made to the template without requiring changes to the data (data independence). Note, however, that this is not symmetrical; if the data changes probably the template will have to change. This is because fields in subforms can match data values at a higher level, but data values in data groups can not match fields at a higher level. The principle involved is that structure in the template is often not meaningful but structure in the data is usually meaningful.

Forms with Non-Unique Names

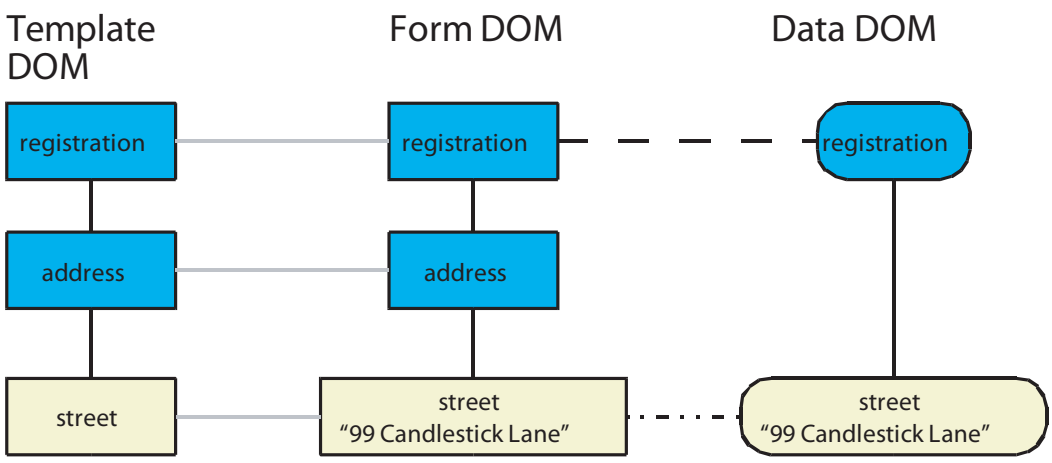
It is legal for forms and data to use the same names in different contexts. For example a field in one subform may have the same name as a field in a different subform. The two fields represent quite different things. It is the job of the data binding process to match form nodes correctly to data nodes whenever possible, even when the names are not unique.

A direct match is not ambiguous unless somewhere in the chain of ancestors there are siblings with the same name. This ambiguity is resolved wherever it occurs along the chain by matching index numbers as well as names.

By contrast it is common for a single data node to directly match one field node and scope match some others which are in different contexts. To deal with this there is a hierarchy of matches, in which a direct match has highest precedence, followed by scope matches. Furthermore, there are two kinds of scope matches with different priorities. A scope match involving only direct ancestors (also known as an *ancestor match*) is preferable because like a direct match it matches not only the names of the nodes but also their index numbers where there are multiple siblings with the same name. This has an effect similar to index inferral in the resolution of SOM expressions. Only if unable to find an ancestor match does the data binding process fall back upon a search for a scope match involving sibling(s) of ancestor(s) (also known as a *sibling match*). In other words, the data binding process tries to find a match within the current branch of the Data DOM, but if it can't, it looks for a match in a related branch. (This two-step process does not correspond exactly to SOM expression resolution because SOM expressions only search an existing hierarchy, they do not create new nodes.) Finally, within the set of ancestor matches and independently within the set of sibling matches, priority is given to matches which ascend through fewer generations toward the root before matching. This reproduces the prioritization rule of SOM expressions. See "Scripting Object Model" on page 73.

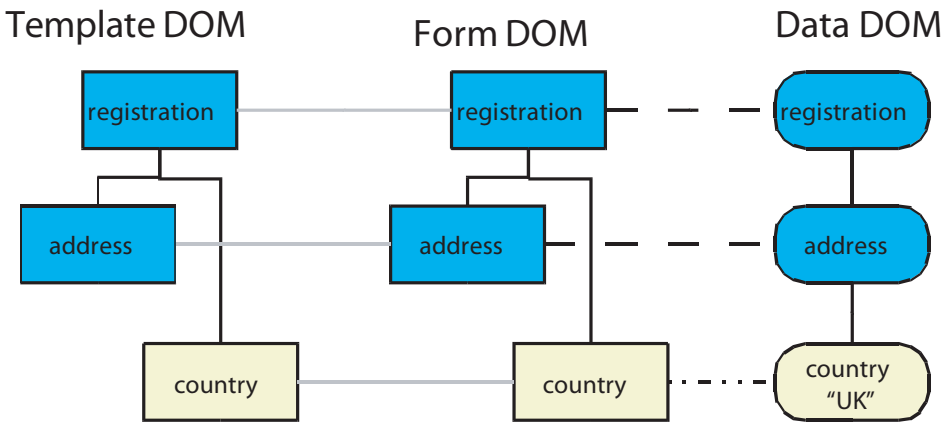
The above details sound complicated but conceptually the distinction between ancestor and sibling matches is simple. An ancestor match deals with the case where a portion of the template has been enclosed in a new subform, so the form node is now lower in the form hierarchy than the corresponding data node is in the data hierarchy. A sibling match deals with the case where a portion of the data has been enclosed in a new subform, so the form node is now higher in the form hierarchy than the corresponding data node is in the data hierarchy.

The following figure shows an ancestor match. This is a single binding from the data binding results shown in the figure on page 165. All of the scope matches in the following figure are ancestor matches.



Ancestor match

The following figure shows a sibling match. In this case, the data has the address information contained within an `address` data group, but in the template the `country` field is still at a higher level. (Perhaps because the country information is often used as a sort key separately from the rest of the address information.) The `country` field is the sibling of the `address` subform, and the `address` subform is bound to the `address` data group. Therefore the `country` data value scope-matches to the `country` field.



Sibling match

These matching rules ensure that once a data group and a subform are bound, descendants of the data group are never bound to form nodes which are not descendants of the subform. This does not apply in the reverse direction, that is, descendants of the subform may be bound to nodes that are not descendants of the data group because of scope matching.

The hierarchy of matches is not important when every field on a form has a unique name. It becomes important when fields in different data groups share the same name, although they are logically distinct. Depending on the exact structure of the form, data values sharing the name may be able to scope-match to each other's fields. For example, consider the following fragment from a passport application.

Example 4.51 Template for a passport application

```
<template>
   <subform name="application">
      <subform name="sponsor">
         <field name="lastname"> … </field> <!-- sponsor's last name -->
         …
      </subform>
      <field name="lastname"> … </field> <!-- applicant's last name -->
      …
   </subform>
</template>
```

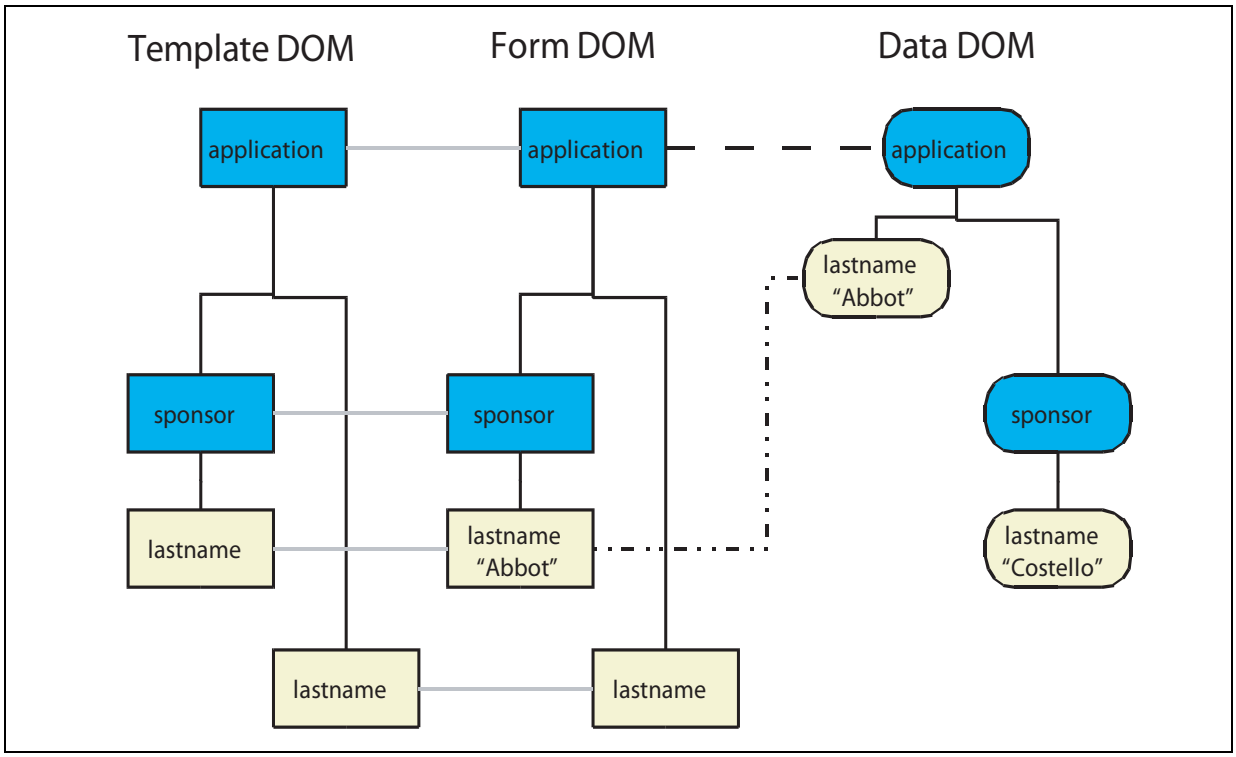
Note that there are two fields called `lastname`, and one subform containing a `lastname` field is descended from another subform containing a `lastname` field. This template is merged with the following data.

Example 4.52 Data for the passport application

```
<application>
   <lastname>Abott</lastname>
   …
```

```
   <sponsor>
      <lastname>Costello</lastname>
   </sponsor>
</application>
```

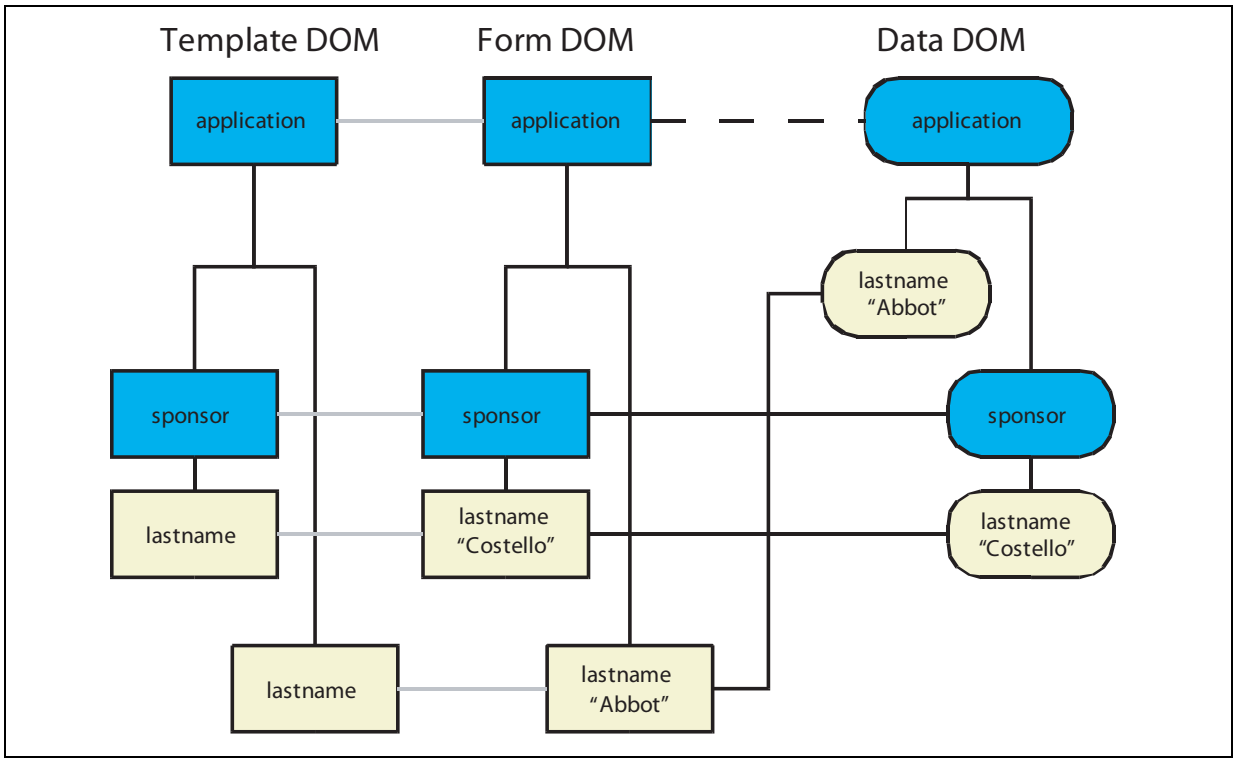
The hypothetical result, if there was no matching hierarchy, is shown in the following figure.



Hypothetical scope-matching without hierarchy would cause an incorrect binding

As depicted in the hypothetical figure above, without a hierarchy the `lastname` field within the `sponsor` subform would bind to the applicant's last name. This would come about because when the data binding process looked for a match for the first `lastname` field in the template (`$form.application.sponsor.lastname`), the first match it would find would be a scope-match to `$data.application.lastname` (containing "`Abbot`"). After this binding took place, the second `lastname` field would fail to match any unbound data. Hence, when the form was displayed, it would display the applicant's last name in the field for the applicant's last name and nothing at all in the field for the sponsor's last name.

The matching hierarchy prevents this from happening. When the data binding process looks for a match for the first `lastname` field, it looks for a direct match first. This causes it to correctly bind this field to `$data.application.sponsor.lastname`. Then, when it looks for a match for the second `lastname` field, it finds `$data.application.lastname` which is still unbound. This yields the correct binding.



Matching hierarchy yields correct binding

In the above example, the matching hierarchy solved the problem because there was enough structure in the data and form to disambiguate the match. But there are times when the hierarchy of matching cannot save the day. For example, assume the same passport application data as the above example. But in this case, the same subform has fields that are required to bind to the two separate same-named data values. The basic template is as follows.

Example 4.53 Modified passport template with no way to deduce proper binding

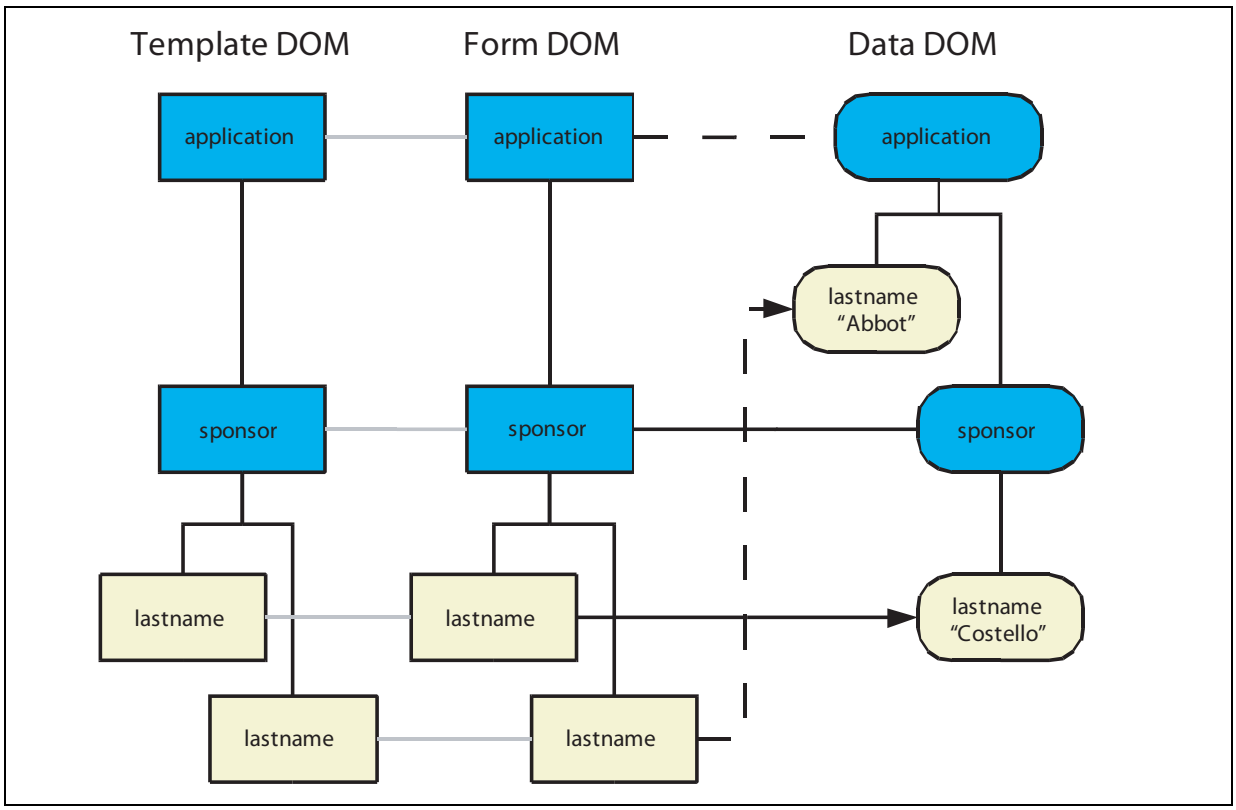
```
<template>
   <subform name="application">
      <subform name="sponsor">
           <field name="lastname"> … </field> <!-- sponsor's last name -->
           <field name="lastname"> … </field> <!-- applicant's last name -->
           …
      </subform>
         …
   </subform>
</template>
```

This template *does not* bind correctly to the data. The data binding process has no way to know which field should bind to which data value. With nothing else to go by it binds them in document order, which in this case is incorrect. The form creator has two remedies. One remedy is to change a field name so that all field names are unique, but this requires either changing the data or modifying it on the way into the Data DOM using a configuration option as described in "Creating, Updating, and Unloading a Basic XFA Data DOM" on page 108. Or, one of the fields can be qualified with an explicit data reference so that it can only bind directly to the desired data value, as described in "Explicit Data References" on page 177. However the explicit data reference only works if the data always has the same hierarchy. Each remedy sacrifices some kind of data independence in order to disambiguate the match. Here is the same template fragment with an explicit data reference added to fix the problem,

Example 4.54 Preceding template repaired with explicit data reference

```
<template>
   <subform name="application">
      <subform name="sponsor">
           <field name="lastname"> … </field> <!-- sponsor's last name -->
           <field name="lastname">
              <bind match="dataRef" ref="$data.application.lastname"/>
              …
           </field> <!-- applicant's last name -->
           …
      </subform>
         …
   </subform>
</template>
```

The result using this template fragment and the same data is shown in the following figure. This is the desired result.



Scope-matching prevented using an explicit data reference

Content Type

In XFA, the template may supply a field with default data that is rich text or an image. However the type of data bound into a field, as indicated by its `contentType` property, may differ from the type of the default data. For example a field with a textual default may bind to an image. The data binding process makes no attempt to match the content type.

Similarly, in an interactive context, the UI may constrain the user to enter data of a particular type. However when the data is supplied from some other source it need not match the UI type. For example a

field with a numeric UI may bind to alphabetic data. This is considered normal and proper. The user is still constrained to enter a date when editing the field.

Transparent Nodes

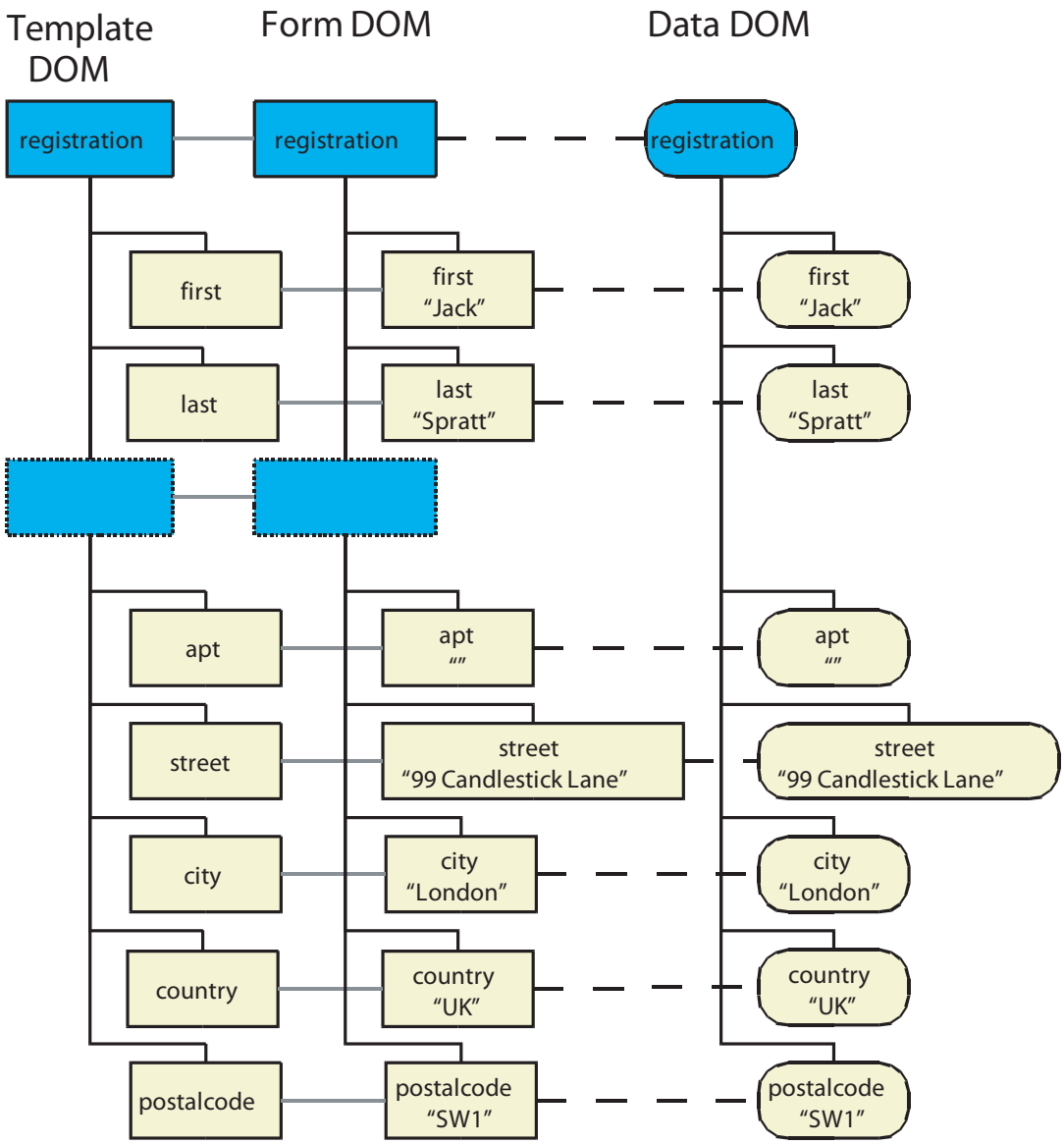
In data binding, as in SOM expressions, certain template nodes (such as nameless subforms) are transparent. This means that data binding, like SOM expression resolution, behaves as though the transparent nodes were removed and their children adopted by the transparent node's parent. For example, if a nameless subform is wrapped around a field, the field still binds to the same data value in the same place in the data hierarchy. The data binding process does copy the transparent node into the Form DOM, so the Form DOM echoes the hierarchy of the Template DOM, but the form node which is a copy of a transparent node remains unbound.

The following example shows the registration template with a nameless subform wrapping around the address information.

Example 4.55 Registration template with a transparent subform added

```
<template …>
   <subform name="registration">
      <field name="first" …>… </field>
      <field name="last" …> … </field>
      <subform>
         <field name="apt" …> … </field>
         <field name="street" …> … </field>
         <field name="city"…> … </field>
         <field name="country"…> … </field>
         <field name="postalcode"…> … </field>
      </subform>
   </subform>
</template>
```

The following figure shows what results when this template is bound to the original data. All of the data bindings are still direct matches because the nameless subform is transparent.



Transparent node handling

Nameless fields are also transparent, unless they make explicit data references. Explicit data references are explained below in "Explicit Data References" on page 177. Note that the transparency of nameless subforms and fields is not arbitrary. Rather, because they are nameless, they cannot be referred to in the normal way by SOM expressions. Hence they are also excluded from data binding. However like nameless subforms they are still copied into the Form DOM where appropriate so that scripts and other XFA subsystems can use them. (Because they are nameless the script has to refer to them via SOM expressions using class names. For example the above nameless subform would be referred to as `$form.registration.#subform`.)

A subform can also be transparent even though it has a name. This happens when the subform's `scope` property is set to `none`. Such a subform is treated like a nameless subform by the data binding process but at the same time scripts can access the transparent subform by name.

Nodes representing `area` elements are also transparent to data binding, even when they are named. There are also various nodes that are descended from subforms but do not participate in data binding

because they can not contain subforms or fields, for example `pageSet` nodes. See "Template Features for Designing Static Forms" on page 29 for more information about these objects. Again, these are copied into the Form DOM for use by scripts and other XFA subsystems.

Exclusion Groups

An exclusion group is a template construct that contains a set of fields, each of which has an activated state and a deactivated state. In an interactive context an exclusion group is normally presented to the user as either a set of radio buttons or a set of checkboxes.

When presented as radio buttons not more than one member of the set can be activated at the same time. When one radio button is turned `on` (depressed) any other radio button in the group that was `on` is forced `off` (released). It is also permissible for every button to be `off`. By contrast, when an exclusion group is presented as check boxes the fields can be activated and deactivated independently.

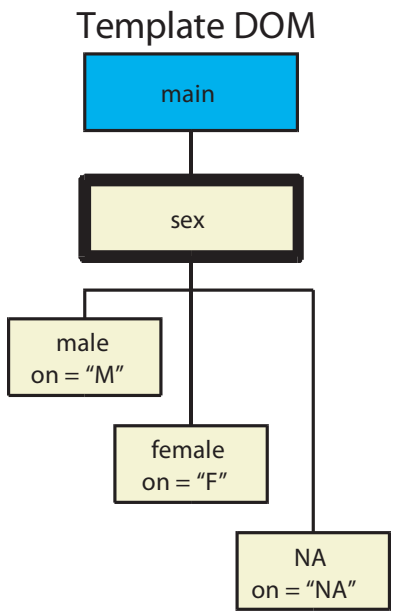
Each field within an exclusion group is associated with a key value. When a field is activated a variable is set to the key value for that field. At any time the field can tell whether it is on or off by comparing the value of the variable to its own key value.

Exclusion groups are declared in the template via an `exclGroup` element enclosing the members of the set. In the following example, the exclusion group itself is named `sex` and it contains three radio button fields named `male`, `female` and `NA` (to represent a declined response). The field named `male` is on when and only when the controlling variable is `"M"`. Similarly `female` is on when it is `"F"` and `NA` is on when it is `"NA"`. For simplicity the accompanying GUI elements are not shown.

Example 4.56 Template using an exclusion group

```
<subform name="main" …>
   <exclGroup name="sex">
      <field name="male">
         <items><text>M</text></items>
      </field>
      <field name="female">
         <items><text>F</text></items>
      </field>
      <field name="NA">
         <items><text>NA</text></items>
      </field>
   </exclGroup>
</subform>
```

Inside the Template DOM, the exclusion group is represented by a node, as shown at right. The node exists purely to encapsulate the logical relationship between its children. It is not itself displayable.



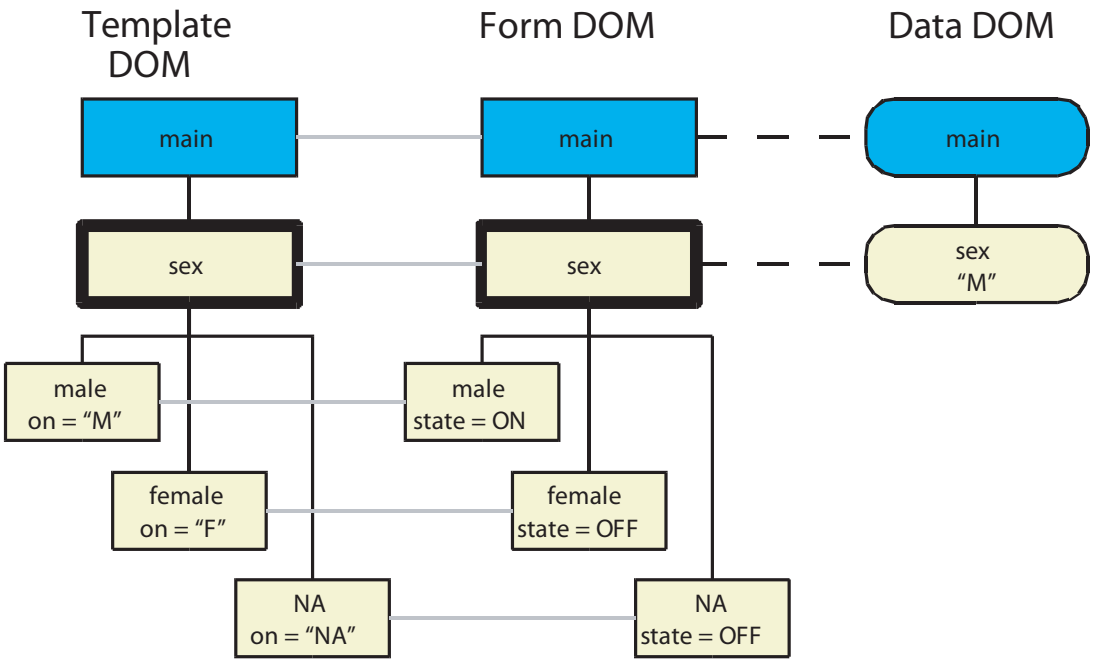
Exclusion group in the Template DOM

If the exclusion group has a name, the exclusion group node itself may be supplied with content by the data. This is called the *short exclusion format*. In this case, the fields belonging to the exclusion group are left unbound. The fields rely on the value of their parent exclusion group to determine whether they are on or off. The following example shows short exclusion format.

Example 4.57 Data using the short exclusion format

```
<?xml version="1.0"?>
<main>
   <sex>M</sex>
</main>
```

After binding the above template to this data, the result is as shown in the following figure.



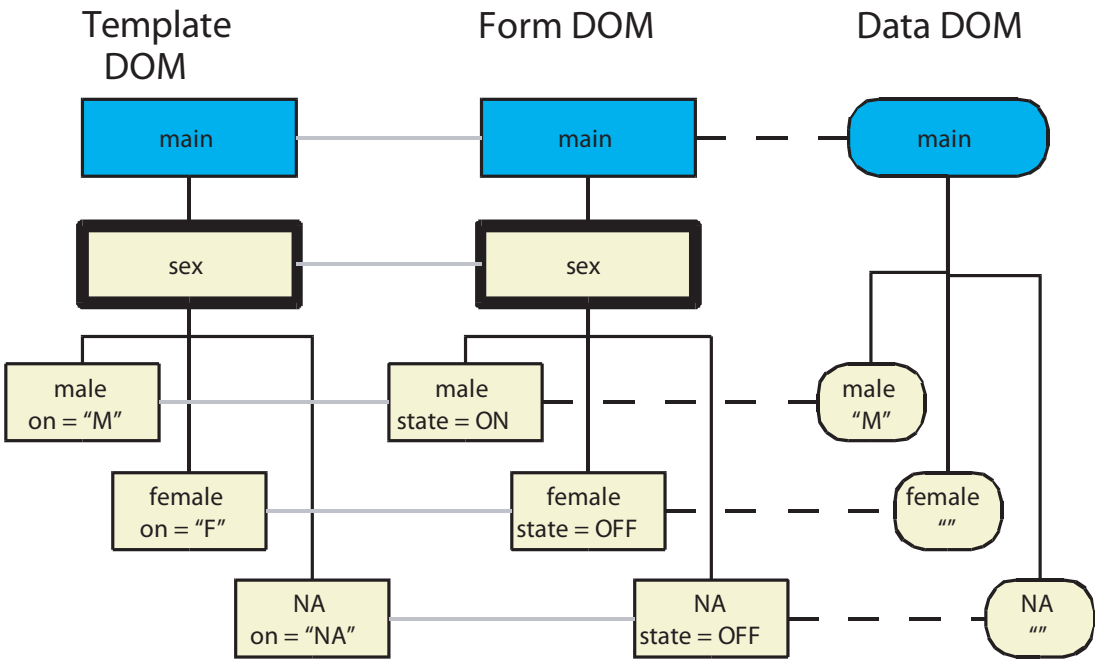
Exclusion group bound using short format

Alternatively the data may provide content explicitly for one or more of the fields within an exclusion group. This is known as the *long exclusion format*. In this case, the field nodes are bound to the corresponding data value nodes. Each bound field node relies on its own value to determine whether it is on or off. The following example shows long exclusion format.

Example 4.58 Data showing the long exclusion format

```
<?xml version="1.0"?>
<main>
   <sex>
      <male>M</male>
      <female></female>
      <NA></NA>
   </sex>
</main>
```

The following figure shows the result of binding the above template to this data.



Exclusion group bound using long format

When supplied with data in the long exclusion format, the binding process is not responsible for enforcing exclusivity. The supplied data must obey the exclusivity constraint. The data binding process may detect violations of this constraint and take appropriate action (for example, emitting a warning).

Binding with the long exclusion format does not require that the exclusion group have a name. However not having a name for the exclusion group may cause problems when submitting data from an exclusion group to a host, because the script writer has no easy way to identify which exclusion group is being submitted. Consequently it is recommended to supply a name for the exclusion group even when planning to use the long exclusion format.

Choice Lists That Can Have Multiple Values

The choice list widget (`choiceList`) may be configured to allow the user to select multiple values, as shown in the following example. The property `open="multiSelect"` indicates the user may select multiple values.

Example 4.59 Field using a multivalued choice list

```
<field name="grains" … >
   <ui>
      <choiceList open="multiSelect"/>
   </ui>
   <items save="1">
      <text>wheat</text>
      <text>rye</text>
      <text>millet</text>
   </items>
</field>
```

During data binding, a field object having a multi-select choice list is bound to a data group, rather than to a data value. The value of the field object in the Form DOM is the concatenation of the values of all of that node's children, with newlines between them. This includes children that do not match any of the items in the choice list. However, the choice list widget does not display values that are not in the list. For example, suppose the relevant data is as follows.

Example 4.60 Data for the multivalued choice list

```
<grains>
   <value>rye</value>
   <value>barley</value>
   <value>wheat</value>
</grains>
```

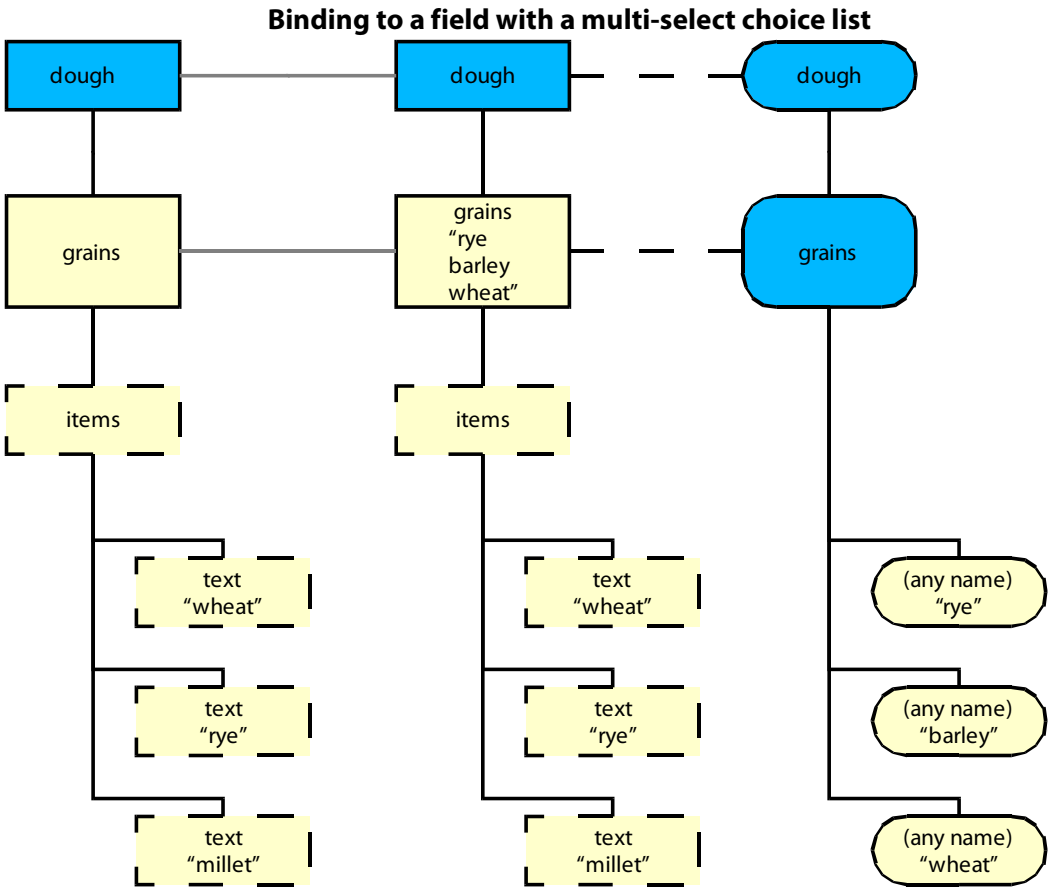
In this case the value of the `grains` field in the Form DOM is as follows:

```
rye
barley
wheat
```

Note that there is no newline after `wheat`. If the template supplies a default value, it must be in the same format.

Although `barley` is in the value it does not appear in the choice list, so the multi-select choice widget ignores it. Instead it displays a list consisting of `wheat`, `rye`, and `millet` (in that order), with `wheat` and `rye` selected. The user can delete `wheat`, delete `rye`, or add `millet`, but the user has no way to remove the value `barley`.

The DOMs for this example are shown below.



Explicit Data References

It is possible for the template to override the automatic matching process. When a field, exclusion group, or subform has a `bind` property with a `match` attribute having a value of `dataRef`, the accompanying `ref` attribute supplies a SOM expression which points directly to a data node. Because a SOM expression can point any place in the Data DOM, the referencing node can bind to any data node of the appropriate type in the Data DOM, regardless of its own location in the Form DOM. If there is no data node matching the expression, the binding process creates one with default properties. If the expression specifies ancestor data nodes that do not exist, they too are created so that the referenced data node can link into the Data DOM at the specified location.

When an explicit data reference is supplied, the SOM expression must expand unambiguously to one data node, that is, it must not contain ".." or "*". The latter restriction is necessary because the expression may be used not only to identify an existing node but to create a new node.

When a form node supplies an explicit data reference, the data binding process does not use the `name` attribute of the form node. For this reason the name can be omitted for such form nodes (i.e. they can be made nameless) without becoming transparent to binding.

The `ref` attribute may point to a data node which is already bound to some other field or exclusion group. When multiple fields or exclusion groups bind to the same data value the result is that they share the same data. If any one of them is updated they are all updated. The situation is different when multiple subforms bind to the same data group. In this case, assuming the children of the subforms do not employ explicit data references, the children still bind to unique data nodes, even if they share the same names. This happens because ordinary binding always excludes data nodes which are already bound.

Match Attributes (Step 3)

Attributes are by default loaded into the Data DOM. If loaded, they are represented by data value nodes with the `contentType` property set to `metadata`. All of the preceding steps in the data-binding process ignore data value nodes representing attributes. Instead they are handled separately at this point in the process.

If attributes have been loaded into the Data DOM, after all the above processing is complete, the data binding process makes one more try to match any yet-unmatched ordinary fields or exclusion groups to data. It looks for attributes of data values that match the names of unbound fields. Note that it ignores attributes of data groups; only attributes of data values are processed. Also, attributes that are already bound via explicit data references are excluded.

For example, suppose the data is as follows.

Example 4.61 Registration data with attributes

```
<?xml version="1.0"?>
<registration>
   <first>Jack</first>
   <last>Spratt</last>
   <street apt="2">99 Candlestick Lane</street>
   <city>London</city>
   <country>UK</country>
   <postalcode>SW1</postalcode>
</registration>
```

Failing to find a match for the `apt` field, the binding process extends the search to attributes of data values. It finds the `apt` attribute of the `street` data value and binds it to the `apt` field. This is useful with data produced by third-party programs which may choose to pass data in attributes rather than content. (There is no general rule in XML for deciding what should be an attribute and what should be content.) Attributes that are not needed to supply values for unbound fields or exclusion groups are ignored.

Re-Normalization (Step 4)

In certain cases, a data node may end up bound to a form node even though the nearest merge-able ancestor of the data node and the nearest merge-able ancestor of the form node are not bound to each other. XFA applications may provide an option to move data nodes around to reconcile these contradictions. This process is referred to as *re-normalizing (or adjusting) the Data DOM*. Re-normalization always does the least moving it can, so the data structure is kept as close to original structure as possible. If the application does not request this service, existing nodes in the Data DOM stay where they are.

The template that was used above in Example 4.49 to illustrate scope matching will also serve to illustrate re-normalization. The template has the following skeleton:

Example 4.62 Registration template with added address subform

```
<template …>
   <subform name="registration">
      <field name="first" …> … </field>
      <field name="last" …> … </field>
      <subform name="address">
         <field name="apt" …> … </field>
         <field name="street" …> … </field>
```

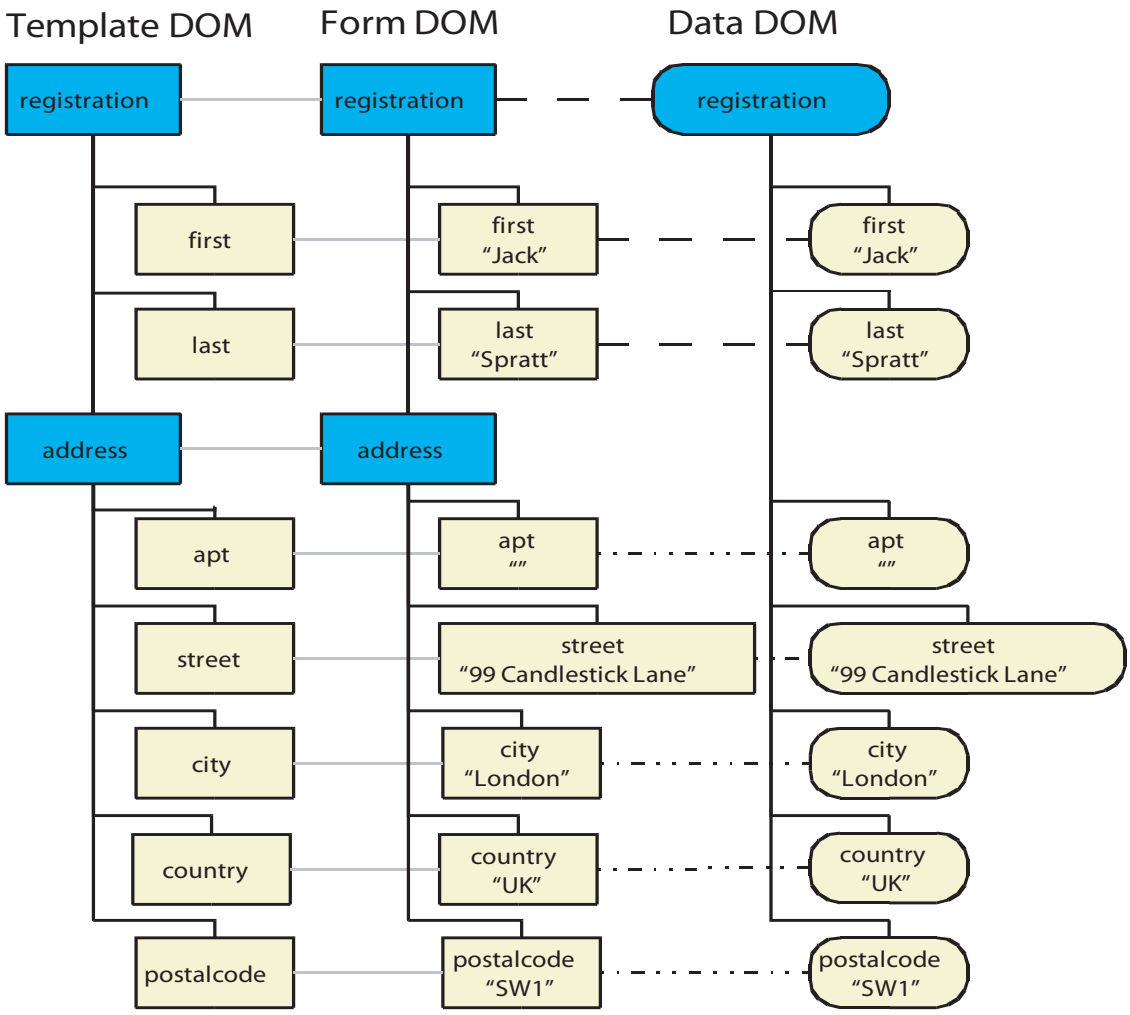
```
        <field name="city"…> … </field>
        <field name="country"…> … </field>
        <field name="postalcode"…> … </field>
      </subform>
   </subform>
</template>
```

The supplied data is also the same as Example 4.47. The data is repeated below for convenience.

Example 4.63 Input registration data

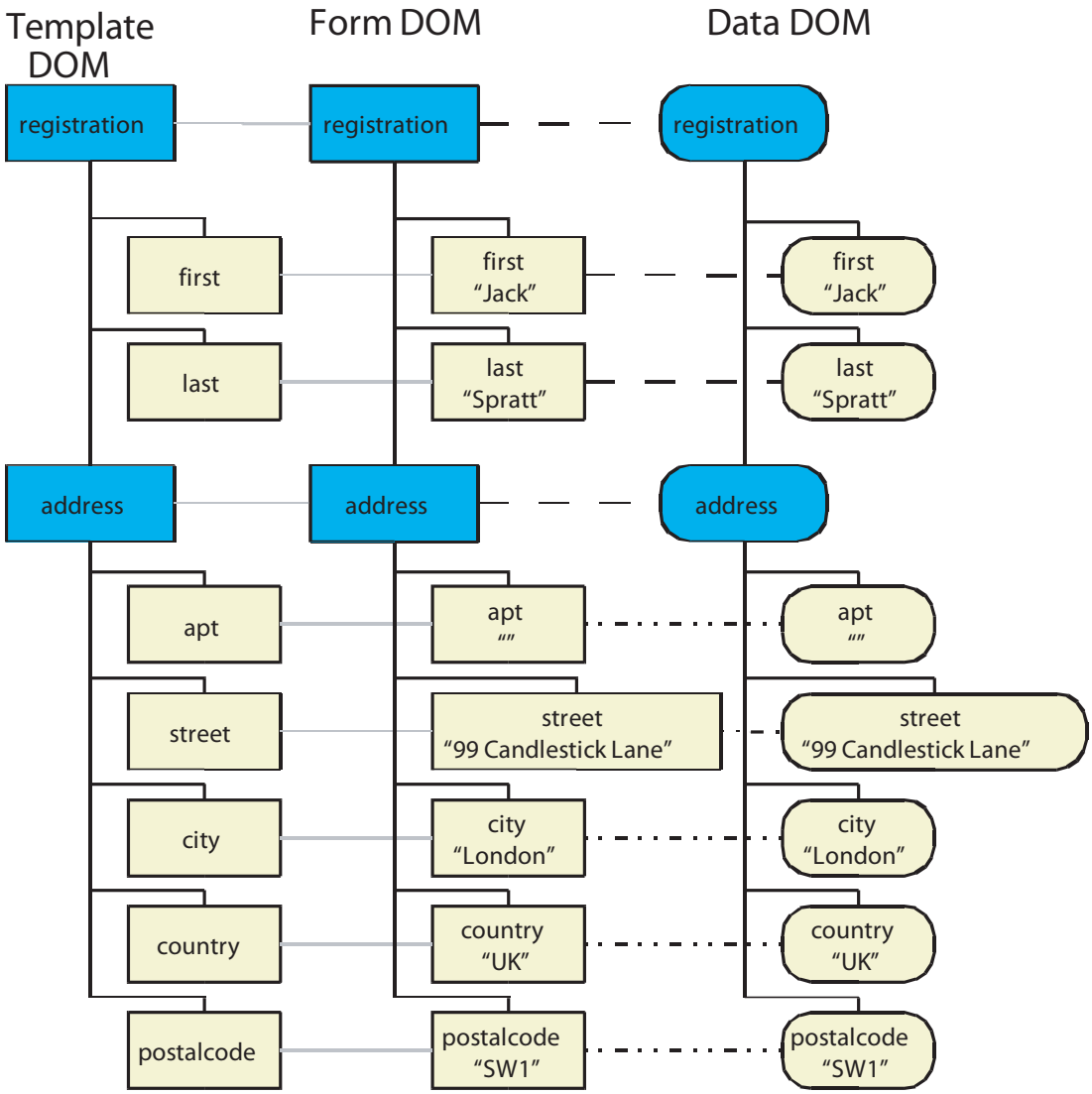
```
<?xml version="1.0"?>
<registration>
   <first>Jack</first>
   <last>Spratt</last>
   <apt></apt>
   <street>99 Candlestick Lane</street>
   <city>London</city>
   <country>UK</country>
   <postalcode>SW1</postalcode>
</registration>
```

The following figure shows the result of the data binding process, before re-normalization.



Result of registration binding with template (duplicated from page 162)

During re-normalization an `address` data group is added to the Data DOM and the scope matched data nodes are moved under the new data group so that the structure of the Data DOM agrees with that of the Form DOM. The following figure shows the result after re-normalization.



Results of binding data with template after re-normalization

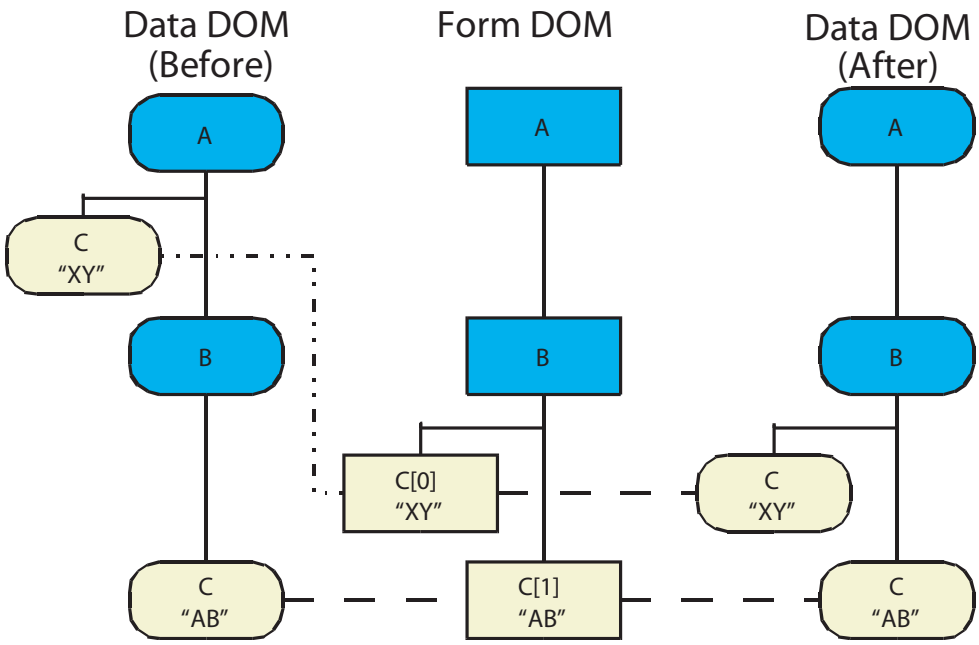
If the re-normalized Data DOM is subsequently written out in XML, the result is as follows.

Example 4.64 Output document showing renormalized data

```
<?xml version="1.0"?>
<registration>
   <first>Jack</first>
   <last>Spratt</last>
   <address>
      <apt></apt>
      <street>99 Candlestick Lane</street>
      <city>London</city>
      <country>UK</country>
      <postalcode>SW1</postalcode>
   </address>
</registration>
```

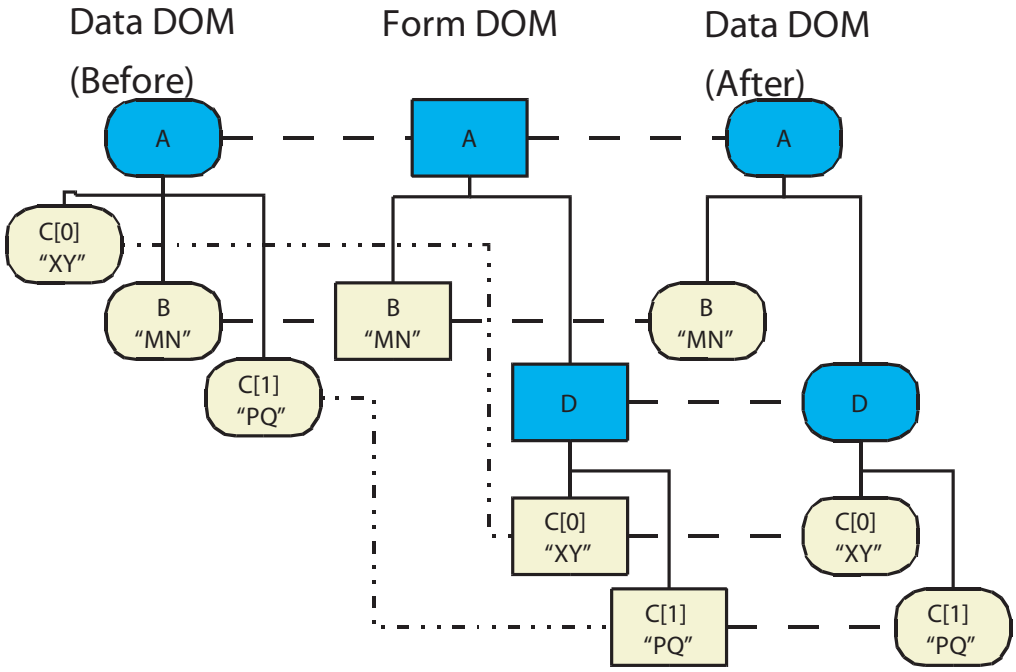
Hence if the application reloads the resulting XML (that is, if the data makes a round trip) the application's data is forced into the structure laid out by the template. This is sometimes very useful, but it is also dangerous - a different template may produce a different restructuring of the data. Therefore the application designer must carefully consider the effects of the option. For a complete set of examples showing the ways in which the Data DOM can be altered by re-normalization, see the following illustrations. Note that in these illustrations the left side shows the original Data DOM (rather than the usual Template DOM).

As shown in the following figure, value `C` (originally a child of Group `A`) is moved to become a child of Group `B`.



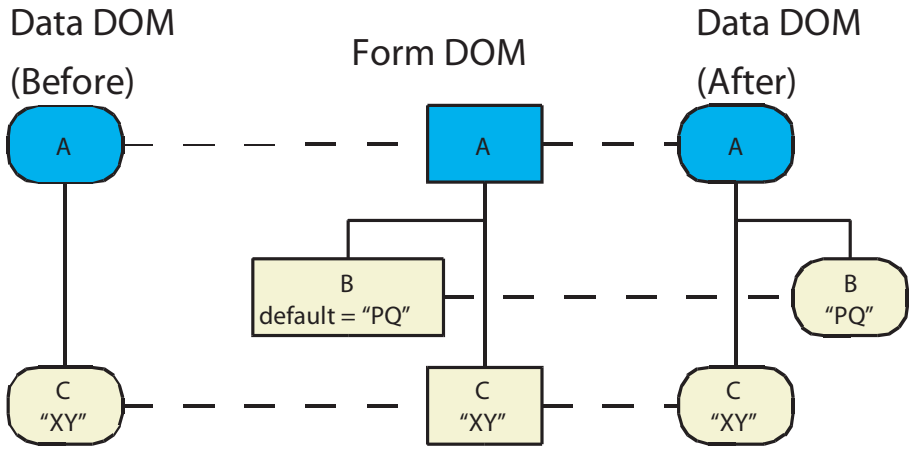
Re-normalization moves a data node

As shown in the following figure, Group D is inserted and two values, `C[0]` and `C[1]`, are moved to become children of the new group.



Re-normalization inserts a data group

As shown in the following figure, value B is created with default properties in order to provide a match for Field B.



Re-normalization creates a data node

Bind to Properties (Step 5)

This step was added in XFA 2.4. The binding process so far has populated by XFA Form DOM and bound data values to the `value` properties of field, exclusion group, and subform nodes in the XFA Form DOM. All the other properties of form nodes are simple copies of the corresponding properties in the template. In this step the XFA processor updates properties (other than `value`) of form nodes from data values. The

data values can be taken from the XFA Data DOM or they can be obtained via connections to web services. This process is controlled by two properties, `setProperty` and `bindItems`.

The setProperty property

This property is used to explicitly copy a particular data value node or a value returned by a web service to a property of the containing field, draw, exclusion group, or subform. This can be used to make captions, assists, and other portions of the form data-driven even though they are not modifiable through the user interface.

A field, draw, exclusion group, or subform may have any number of `setProperty` children. Each `setProperty` child affects one property of its parent.

The `setProperty` element takes a `ref` attribute, plus an optional `connection` attribute, which together define the source of the data to be copied. A mandatory `target` attribute identifies the property to copy into.

The `ref` attribute takes as its value a SOM expression with one special restriction: the expression may not use the ".." syntax. The SOM expression may be a relative expression. When there is no `connect` attribute the expression is evaluated in the context of its parent (which is its container). When there is a `connect` attribute the expression is evaluated in the context of the the nearest ancestor that asserts a fully-qualified XFA SOM expression as its value of `ref` for the same connection. For example if a subform has a `ref` attribute with a value of `!connectionData.queryDatabase.body` then its child field could use the relative expression `queryID` as a synonym for `!connectionData.queryDatabase.body.queryID`. In all other ways the value of this property is a normal XFA SOM expression.

The target must be a property (or subproperty) of the parent object. It cannot be a property of an object contained by the parent. For example, within a subform the target may be specified as `assist.tooltip` (a subproperty of the subform itself) but it may not be specified as `#field.rotate` (a property of a field object contained within the subform). You have to put the `setProperty` on the object to which the target belongs. For example, the following template fragment copies data from the XFA Data DOM into several properties of a subform and other data into properties of a field contained within the subform.

Example 4.65 Subform and field using setProperty

```
<subform name="Customer" ...>
   <setProperty ref="$data.Tips.Customers" target="assist.toolTip"/>
   <setProperty ref="$data.Style.Background" target="fill.color.value"/>
   <field name="LastName" ...>
      <setProperty ref="$data.Style.NameFont" target="font.typeface"/>
      <setProperty ref="$data.Style.NameSize" target="font.size"/>
   </field>
</subform>
```

The target can be almost any property of the containing object. The following restrictions apply.

- It is not legal for the target to be the `setProperty` property itself or any of its subproperties.
- It is not legal for the target to be a `bindItems` property or any of its children. Both `setProperty` and `bindItems` are processed during a single phase of the data merge process and their respective order of evaluation is not guaranteed.

Some targets are legal but not recommended.

- It is legal for the target to be the `relevant`, `use` or `usehref` property but it is not recommended. These properties are processed early and changing them afterward has no effect.

- It is legal for the target to be a bind-related property (such as the `name` property or any `bind` or `occur` subproperties) but it is not recommended. Specifying a bind-related property as the target is unlikely to yield a useful result. Processing of `setProperty` is done near the end of the merge operation when the bind-related properties have already had their effects. Changing them with `setProperty` has no effect on the current merge operation. It may however affect subsequent incremental merge operations.
- It is legal for the target to be the `value` property of the parent field or exclusion group, but it is not recommended. It is better to do an explicit data reference by setting the parent's `bind.match` property to `dataRef` and the parent's `ref` attribute to the target instead. By contrast when the parent is a `draw` object there is no way to specify an explicit data reference and it is proper and expected to specify its `value` property as the target of a `setProperty`.

Note that `setProperty` processing is a templating process, not a binding process. Whenever a merge is performed a snapshot of the specified data is copied into the specified properties. Subsequent updates to the data do not propagate into the target or vice-versa. This is appropriate because the targets of `setProperty` are things that the user does not normally have the ability to change interactively.

Caution: Careless use of `setProperty` can create security vulnerabilities. It is up to the form creator to ensure that security is not compromised.

The bindItems property

This property is used to load the `items` property of a field from a set of data value nodes or from a set of values returned by a web service. If the `items` property already exists and is non-empty the old contents of the `items` property are lost.

It is always legal for a field to have a `bindItems` property, but as with `items` the property is ignored unless the field has a suitable user interface. Suitable user interfaces include choice lists, check boxes, and radio buttons. Note that when `bindItems` refers to a web service but the user interface is unsuitable the web service is *not* accessed, so there are no side-effects.

The `bindItems` element takes a `ref` attribute, plus an optional `connection` attribute, which together identify a set of data nodes. Each member of the set is used to create an item for the list. A mandatory `valueRef` attribute identifies a particular data value within each member of the set. These values are used to generate the actual value strings for the items. In addition an optional `labelRef` attribute identifies a data value within each member of the set that is used to generate label strings for the items. If there is no `labelRef` attribute then the value strings are also used as labels.

Note: Unlike `setProperty`, whenever possible `bindItems` does not merely use the data as a template. Instead it makes a true binding. Any subsequent change in the bound data automatically propagates into the `items` property. This is an exception to the usual XFA practice for non-interactive properties. The exception was made to simplify interoperation with forms written in XForms and using an `itemset` element. However when the `connection` property is non-empty binding cannot be done because connection data is transient. In this case the XFA processor performs a simple templating (copying) operation.

The `ref` attribute takes as its value a SOM expression with one special restriction: the expression may not use the `".."` syntax. When there is no `connect` attribute this expression is evaluated in the context of the data node bound to the containing field object in the XFA Form DOM. When there is a `connect` attribute this expression is evaluated in the context of the the nearest ancestor that asserts a fully-qualified XFA SOM expression as its value of `ref` for the same connection. For example if a subform has a `ref` attribute with a value of `!connectionData.queryDatabase.body` then its child field could use the relative expression `queryID` as a synonym for `!connectionData.queryDatabase.body.queryID`. In all other ways the value of this property is a normal XFA SOM expression.

The values of `valueRef` and `labelRef` must be SOM expressions. These expressions may contain `"..."` if desired. They are evaluated in the context of the data node to which the `ref` expression points.

For example, the author of a form wishes to populate a choicelist with a set of credit cards. The set of credit cards is to be taken from the data file. The data contains the following structure.

Example 4.66 Data fragment containing items for a choice list

```
<ccs>
   <cc uiname="Mastercard" token="MC"/>
   <cc uiname="American Express" token="AMEX"/>
   ...
</ccs>
```

The author accomplishes his goal using the following template fragment.

Example 4.67 Field loading choice list from data

```
<field ...>
   <bindItems ref="$data.ccs.cc[*]" labelRef="uiname" valueRef="token"/>
   <ui>
      <choiceList/>
   </ui>
</field>
```

Calculations and Validations (Step 6)

The last thing that the data binding process does for any given record is to trigger the execution of certain scripts.

Fields in the template may have calculation and validation scripts attached to them. A calculation script returns a value which becomes the new value for the field. A validation script returns a status value which, if false, causes actions to be taken such as displaying an error message. Calculation and validation scripts must not make alterations to the structure of any of the DOMs, such as adding, deleting, or moving nodes. In addition validation scripts must not alter any values. Calculation and validation scripts may be triggered under various other circumstances, not only upon completion of data binding.

Note that calculations are performed even for fields that were supplied with data in the course of data binding. The calculation may thereby update the supplied value. Similarly validations are applied even to values that are supplied by the template to an empty merge as default values. Hence a validation may declare a failure or warning in response to a default value.

For more information about these and other scripts and events see the chapter "Automation Objects" on page 322.

Form Ready Event (Step 7)

After all records have been successfully processed the data binding process triggers the `ready` event on the `$form` object. Scripts attached to this event can execute confident in the knowledge that data binding has successfully concluded and all data has validated.

For more information about events see "Events" on page 335.

Remerge and Incremental Merge (Step 8)

It is possible for scripts to modify the Data DOM after a merge operation has already taken place. Deleting a data object may leave a form object unbound but otherwise does not alter the Form DOM. Inserting a data object does not in itself alter the Form DOM at all; the newly inserted data object is unbound. However it may be desired to update the bindings between data and form objects. There are two approaches to doing this, known as *remerge* and *incremental merge*.

Remerge consists of deleting the contents of the Form DOM and performing a complete data binding operation from beginning to end. This is drastic and may be slow.

Incremental merge consists of applying the data binding process to a subset of the Data DOM and a subset of the existing Form DOM. In this case, the algorithm, apart from the initial conditions, is almost the same as for an ordinary merge. Processing starts with a particular pair of nodes, one form node and one data node, and operates only on the subtrees below those nodes. Within these subtrees, form and data nodes which are already bound are simply ignored. Otherwise the processing is just as described above for an ordinary merge. Incremental merge is often used after the Data DOM has been updated, for example after receiving updated data from a web service.

Form Processing

Form processing is described as part of "Updating the XML Data DOM for Changes Made to the XFA Data DOM" on page 134.

Data Output

Data output is described in "Unload Processing" on page 135.

5 Representing and Processing Rich Text

This chapter explains how rich text is represented in the DOMs that compose an XFA form. It explains how rich text is identified, how it is converted into plain text as it is represented in the XFA DOMs and how it is printed.

About Rich Text

Rich text is text data that uses a subset of HTML and CSS markup conventions to signify formatting such as bold and underline. Rich text may also include embedded text objects. XFA supports the subset of HTML and CSS markup conventions described in "Rich Text Reference" on page 1027.

Rich text may appear in data supplied to the XFA form. Rich text may also appear in XFA templates as boilerplate text, field captions, or default text values.

Prior to XFA 2.4 rich text was limited to languages that presented in the left-to-right, top-to-bottom order that European languages use. Starting with XFA 2.4 right-to-left top-to-bottom languages such as Hebrew and Arabic were also supported. This change expanded the set of Unicode character points that are supported. The set was expanded to include all characters in right-to-left top-to-bottom languages plus those code points which Unicode assigns to explicitly control flow direction. XFA processors also infer flow direction from the locale property of the container and from the content of the text. However these added capabilities are invisible to rich text markup, which is the subject of this chapter. Instead, for more information about text flow direction see "Flowing Text Within a Container" on page 52.

Rich Text Used for Formatting

Rich text data is formatted as specified by the markup specifications in the rich text. The markup specifications take precedence over formatting specifications in the containing element, which appear in the `font` and `para` elements.

In general, GUI-based template design applications and XFA processing applications provide formatting buttons that allow users to apply styling characteristics to text. For example, the UI in such applications may provide a **Bold** button the user applies to selected text. In response, the application converts the entire body of in-focus text into a rich text representation and encapsulates the selected text within a `span` element, as shown in the following example.

Example 5.1 Fragment of data containing rich text

```
<field1>
   <body xmlns="http://www.w3.org/1999/xhtml">
      <p>The following <span style="font-weight:bold">word</span>
      is in bold.</p>
   </body>
</field1>
```

The set of formatting markup supported by XFA processors is discussed in detail in "Rich Text Reference" on page 1027.

Rich Text That Inserts External Objects

Another use for the rich text idiom is run-time insertion of data into a larger body of text. This can be done even with text that is nominally boilerplate, such as the content of a `draw` element. The embedded data is the content of some object accessible via an XFA-SOM expression. Most commonly it is the content of a field, but it is not restricted to fields.

If the embedded object contains rich text then the content of the object may be incorporated as rich text. However it may optionally be reduced to plain text before incorporation.

The embedded object may contain an image. In this case the image flows with the surrounding larger body of text as though it was a single character. This is discussed in more detail in "Text" on page 48.

For more information about this use of the rich text idiom, see "Rich Text That Contains External Objects" on page 193.

Version Identifiers for Rich Text Producers and Rich Text Specifications

Rich text may be produced by a variety of sources and may include a range of XHTML and CSS features, not all of which are supported in XFA.

XFA grammar adds version numbers as optional attributes in the rich text HTML `body` element. These attributes identify the version of the application producing the rich text (`xfa:APIVersion`) and identify the version of the rich text spec to which the rich text complies (`xfa:spec`). These attributes are described in "Version Specification" on page 1044.

Representation of Rich Text Across XML and XFA DOMs

Recognizing Rich Text

One set of rules is used to recognize rich text in the template and another set is used to recognize rich text in data. The rules for recognizing rich text in data is more relaxed, reflecting the varied origins of data. Rich text data appears within some element named according to the user's own schema, whereas in the template it is within an element defined by an XFA schema.

Recognizing Data as Rich Text

For data to be recognized as rich text, at least one of the following criteria must be met:

- *Content type*. The enclosing element bears a `contentType` attribute in the namespace `http://www.xfa.org/schema/xfa-data/1.0/` with the value of `text/html`.

  Note: Do not use `text/xhtml` as the value for `contentType`; it is *not* recognized.
- *Namespace*. The element content belongs to the XHTML 1.0 namespace

Recognizing Rich Text Introduced in the Template

For template text values to be recognized as rich text, *all* of the following criteria must be met:

- Contained in `exData` with `contentType="text/html"`. (The default value for `contentType` is `"text/plain"`.
- Includes the XHTML namespace
- Rich text contained in `<body>` or `<span>` element

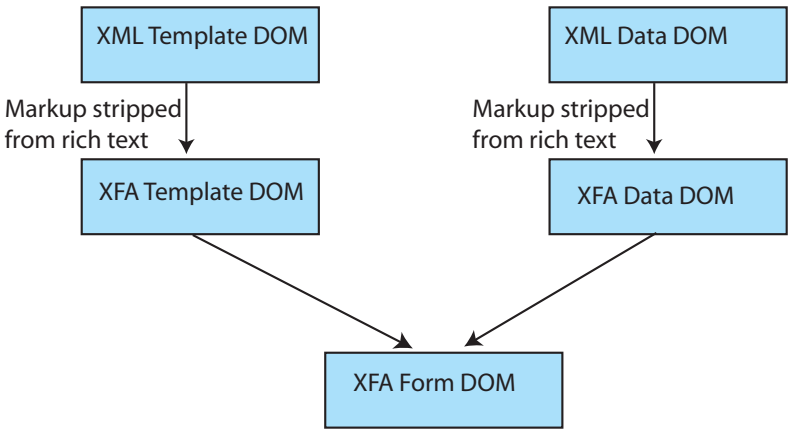
Additionally, rich text should include an `xfa:APIVersion` attribute to indicate the rich text support it expects.

Representing Rich Text in the XFA Data DOM

Rich text does not appear within XFA Data DOM. XFA processors are required to perform some operations directly upon the rich text (for example rendering it on a display). They do so by accessing the originating XML DOM. Scripting languages may also make the associated rich text available but they are not required to.

In the case of the XFA Template DOM, the rich text is simply excluded from the DOM. By contrast in the XFA Data DOM the rich text is represented by plain unstyled text derived from the rich text. This ensures that scripting languages can recognize and operate on the data regardless of styling. The relationship between the XML Data DOM and the XFA Data DOM is discussed at greater length in "XML Data DOM and XFA Data DOM" on page 72.

Rich text converted into plain text for XFA DOMs



Converting Rich Text into Plain Text

Rich text is converted to plain text in the following manner:

1. Start with a copy of the rich text including all markup.
2. Delete all start and end tags and empty elements (whether they represent supported markup or not).
3. Convert all XML character escapes to their literal form (for example, `"<"` to `"<"`).
4. Normalize the white space by replacing all contiguous sequences of white space and/or newline characters with single spaces.
5. Trim any leading and trailing spaces.

Properties of XFA Data DOM Objects That Represent Converted Rich Text

When the data loader recognizes rich text, it sets the `contentType` property of the `dataValue` node corresponding to the enclosing element to `"text/html"`. This property tells rich text capable applications that they should look below the corresponding node of the XML Data DOM for the original rich text. As mentioned in "About the XFA Data DOM" on page 109, each node in the XFA Data DOM

contains a pointer to the corresponding node in the XML Data DOM. However, XFA-SOM does not provide access to that pointer, so access from script to the XML Data DOM is application dependent.

For example, the following XML fragment contains rich text. The rich text content is highlighted.

Example 5.2 Rich text data containing styling

```
<message>
 <p xmlns="http://www.w3.org/1999/xhtml">
    You owe <b>$25.00</b>.  Please pay up!
 </p>
</message>
```

After loading, the above fragment is represented in the XFA Data DOM as follows.

```
[dataValue message = "You owe $25.00. Please pay up!"
  contentType="text/html"]
```

In addition to the above constraints, when specified via a `contentType` attribute on the enclosing element, the rich text content must have a single outer element. Only white space is allowed within the region of rich text content and outside the outer element.

The data loader may emit a warning message when it encounters a construct that violates the above rule. How the application subsequently processes the affected content is implementation defined.

The content in the following example is illegal because the rich text is not enclosed within a single outer element.

Example 5.3 Illegal rich-text data with no outer element

```
<message xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
     xfa:contentType="text/html">
     You owe <b>$25.00</b>.  Please pay up!
</message>
```

In the example, the `message` element is not part of the rich text because the `contentType` attribute applies to the content of the declaring element but not to the element itself. Hence, the rich text does not have include an enclosing element. However, it would not be a good idea to declare that the `message` element was part of the rich text, because HTML markup does not include a `message` element. Rather, the above example of illegal content could be made legal by wrapping the text in a `span` element as follows.

Example 5.4 Previous example corrected

```
<message xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
  xfa:contentType="text/html">
    <span>You owe <b>$25.00</b>.  Please pay up!</span>
</message>
```

Properties of XFA Template DOM Objects That Represent Converted Rich Text

The template loader creates a node in the XFA Template DOM for the `exData` object. The node has nothing below it nor does it not have a `value` property. The `exData` object has a pointer into the XML (not XFA) Template DOM, which allows the XFA processing application to read the original rich text. Each node in the XFA Template DOM contains a pointer to the corresponding node in the XML Template DOM. However, XFA-SOM does not provide access to that pointer, so any access from scripts to the XML Template DOM is application dependent.

The following presents several examples of template expressions related to rich text.

Example 5.5 A draw element that includes rich text

```
<draw … >
   <ui/>
   <value>
      <exData contentType="text/html" maxLength="0">
         <body xmlns="http://www.w3.org/1999/xhtml"
            xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
            xfa:APIVersion="1.4.4136.0"><p>The Title of my
            <span style="font-style:italic">Document</span></p>
         </body>
      </exData>
   </value>
</draw>
```

Example 5.6 A field caption that includes rich text

```
<field … >
   …
   <caption reserve="18.26mm">
      <value>
         <exData contentType="text/html" maxLength="0">
            <body xmlns="http://www.w3.org/1999/xhtml"
               xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
               xfa:APIVersion="1.4.4136.0">
               <p>Text<span style="xfa-spacerun:yes"> </span>
               <span style="font-weight:bold">Field</span></p>
            </body>
         </exData>
      </value>
   </caption>
</field>
```

Example 5.7 A field that accepts rich text as a data value

```
<field … >
   <ui>
      <textEdit allowRichText="1">
         <border/>
         <margin/>
      </textEdit>
   </ui>
   <value>
      <exData contentType="text/html" maxLength="0">
         <body xmlns="http://www.w3.org/1999/xhtml"
            xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
            xfa:APIVersion="1.4.4136.0"><p>A default data
            <span style="font-style:italic">Value</span></p>
         </body>
      </exData>
   </value>
   …
</field>
```

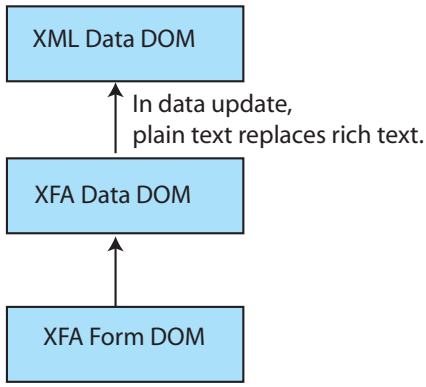
Providing Rich Text Data Through a User Interface

The XFA template may indicate whether the user interface can accept rich text as the value for text. This indication is supplied with the `allowRichText` attribute in the `textEdit` element. If the `textEdit` element allows rich text, the user may specify styled text, usually through styling buttons. If the `textEdit` element prohibits rich text, rich text may not be provided for the value.

Updating the XML Data DOM

Rich text supplied by the XML Data DOM may be replaced by plain text when the XML Data DOM is updated. Such replacement can happen in the following situations:

- Rich text is not supported by the XFA processing application or the platform upon which it is running. For example, the platform is a cell phone with text entry via the keypad.
- User is not allowed to supply rich text, as indicated by the `allowRichText` property. For example, the content of the field is a name (which cannot usefully be styled) but the default value for the field is, in italics, the words "*Not supplied*".
- Text data provided by a calculation.



Rich Text That Contains External Objects

Rich text may contained attributes that reference external plain-text or rich-text objects. Such external references are resolved during the layout process. The referenced data is inserted at the point where the external reference appears and is formatted according to any relevant format picture clauses as described "Dataflow Paths for Localizing and Canonicalizing Data" on page 143. XFA provides `span` attributes that specify the type of reference and whether HTML or CSS specifications in the imported rich text should be retained, as described in "Embedded Object Specifications" on page 1044.

The bold expressions in the following XFA segment is an example of an embedded object. This example uses a Script Object Model (SOM) expression to reference the contents of the field named "AMOUNT_OWING". Such SOM expressions are later described in "Scripting Object Model" on page 73.

Example 5.8 Rich text containing an embedded object

```
<subform>
   <field name="NOTICE">
      <ui> … </ui>
      <value>
         <exData contentType="text/html">
            <html xmlns="http://www.w3.org/1999/xhtml"
               xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
               <p>You owe us
                  <span xfa:embed="AMOUNT_OWING"
                     xfa:embedMode="formatted"/>! Please
                  pay before the end of the month.
               </p>
            </html>
         </exData>
      </value>
```

```
   </field>
   …
   <field name="AMOUNT_OWING" presence="hidden">
      <format>
         <picture>s$zz,zz9.99</picture>
      </format>
   </field>
   …
</subform>
```

In the example the default content of the NOTICE field is rich text which contains an embedded reference to the hidden AMOUNT_OWING field. Assuming that the value in the AMOUNT_OWING field is 52981.89, the NOTICE field is rendered as shown below.

```
You owe us $52,981.89! Please pay before the end of the month.
```

Displaying and Printing Rich Text

When rich text is printed or displayed it may be impossible to make it look exactly as the rich text markup commands. For example, the specified typeface may not be available on the printer that is being used. The renderer makes a best effort to present the rich text as closely as possible to what the markup demands. The heuristics it uses are application-dependent, however completeness (displaying all the text) always takes precedence over appearance.

Note that the layout stage of processing necessarily assumes that the required typeface is available in the required style and size. The renderer may discover that the required typeface is not available, but the layout has already been done. Therefore when the renderer is forced to use a different typeface, or even a different type size, the rendered text may have an unattractive appearance. For example, glyphs may overlap.

6 Template Features for Designing Forms with Repeating Sections

Often forms have sections that are very similar or identical to other sections. For example, a form may accommodate multiple records of data. For the sake of convenience and compactness XFA provides facilities to support reuse of sections within the form.

There are two types of reuse supported in XFA. Prototypes allow for a declaration to be incorporated by reference at different places in the template. The prototyping mechanism allows properties and children of the prototype to be overridden by each instance, so it can be used for components that are similar but not exactly the same. The mechanism for prototypes is described below. By contrast the repetition mechanism for subforms causes the subform and its content (including fields) to be replicated exactly. In this case only the contents of fields can vary between instances. The mechanism for exact repetition is described in "Forms with Repeated Fields or Subforms" on page 202.

Prototypes

A form typically contains a great number of duplicated or similar objects; the same fonts, colors, borders, etc. are used repeatedly throughout the form. Often, there are objects such as lines, rectangles, and even whole field and draw objects repeated.

This presents an opportunity to significantly reduce the file size of a form template, by factoring out the common aspects of the form into prototypical objects. As an additional benefit, the form designer may easily apply sweeping change to the form template by making changes to a prototypical object and any objects that are based upon that object will receive the changes.

Defining Prototypes

An element located anywhere in the template or in an external template can be used as a prototype. However it is often convenient to enclose an element in a `proto` element to indicate that it is included in the template purely for use as a prototype. Almost any XFA template element can be enclosed within a `proto` element. When enclosed this way the enclosed element plays no part in the form except when it is referenced by a `use` or `usehref` attribute on some other element. For example, the following fragment defines a prototype for a field element.

Example 6.1 Defining a prototype using the proto element

```
<proto>
   <field id="LastNameProto"
      name="LastName"
      anchorType="middleLeft">
      <ui>
         <textEdit multiLine="0"/>
      </ui>
   </font>
</proto>
```

Even though the field is fully specified, it will never be directly visible or accessible to a form filling user, nor will it participate directly in data binding.

The proto element itself can appear as a child of only the subform element. This isn't an undue restriction, as every template element is a descendant of some subform, except for the root-level subforms and their enclosing template element.

The subform may hold more than one proto element for multiple prototypes, or the prototypes may be grouped in a single such element. The following two examples are equivalent.

Example 6.2 Subform using a separate proto element for each prototype

```
<subform>
   <proto>
      <color id="RED" value="255,0,0"/>
   </proto>
   <proto>
      <color id="GREEN" value="0,255,0"/>
   </proto>
...
</subform>
```

Example 6.3 Subform grouping prototypes inside a single proto element

```
<subform>
   <proto>
      <color id="RED" value="255,0,0"/>
      <color id="GREEN" value="0,255,0"/>
   </proto>
...
</subform>
```

Almost any template element can be a prototype. Notable exceptions include the proto and template elements.

Referencing Prototypes

An element can refer to a prototype through either its (the referencing element's) `use` or `usehref` attribute. These two attributes have similar effects but `usehref` is more flexible. The `use` attribute can only refer to an *internal prototype* - a prototype in the same template. By contrast `usehref` can also refer to an *external prototype* - a prototype in an external document. Any particular element can employ only one prototype. If both `use` and `usehref` are present and non-empty `usehref` is employed.

The `use` attribute, if it is non-empty, holds a reference to the prototype to be used. The form of the reference can be either of

- *#ID*
- *expression*

where *ID* is an XML ID string and *expression* is a SOM expression.

For a successful reference to occur, the reference must refer to a single element that is located in the template packet of the document and is of the same type as the referencing element.

Example 6.4 Declaration and invocation of a simple prototype.

```
<proto>
   <font id="HELV-RED"
     typeface="Helvetica"
     size="10pt"
     weight="normal"
     posture="normal"
     >
   <color value="255,0,0"/>
   </font>
</proto>
<field ...>
   <font use="#HELV-RED"/>
   ...
</field>
```

This defines a field whose font is red 10pt Helvetica regular. Note that several fields would likely reference this font prototype, thereby reducing file size and allowing for global format changes.

The `usehref` attribute, when it is non-empty, holds a reference to a prototype located in an external document. Although its function closely parallels the `use` attribute its syntax is different. The form of the reference in a `usehref` attribute can be any of

- `.#`*ID*
- *URI*`#`*ID*
- `.#som(`*expression*`)`
- *URI*`#som(`*expression*`)`
- *URI*

where *URI* is the Universal Resource Identifier for an external document, *ID* is the XML ID of the prototype, and *expression* is a SOM expression resolving to the prototype. When `'.'` is supplied instead of a URI the prototype is in the referencing document. Again the reference must be in the template section of whatever document is referenced and must resolve to a single element of the same type as the referencing element.

When a SOM expression is provided it is resolved in the context of the root XFA node, `xfa`. In practice the SOM expression is usually fully-qualified.

When neither an XML ID nor a SOM expression is provided (the third case) the expression `#som($template.#subform.#subform)` is assumed. In other words the default place to look is the first subform child of the root subform in the external document's template packet.

Prototypes may reference other prototypes. In addition, descendant elements of a prototype may reference prototypes. For example, in the following template fragment a `field` invokes a `font` prototype called `HELV-RED`, which in turn invokes another font prototype called `HELV`. The relationship is similar to

a superclass-subclass relationship in an object-oriented language. `HELV-RED` also invokes a color prototype called `RED`. This achieves the same result as the previous example.

Example 6.5 Nested prototype invocations.

```
<proto>
   <color id="RED" value="255,0,0"/>
   <font id="HELV"
      typeface="helvetica"
      size="10pt"
      weight="regular"
      posture="upright"
      >
   </font>
   <font id="HELV-RED" use="HELV">
      <color use="#RED"/>
   </font>
</proto>
<field ...>
   <font use="#HELV-RED"/>
   ...
</field>
```

Caution: It is permissible for internal prototypes to reference external prototypes and vice versa. However when an external prototype references an internal prototype the internal prototype is resolved within the context of the source document, that is to say the original template. For example, in the following example a template in the file mytemp.xdp invokes an external prototype `ClientSubform` in `myprot.xdp`. This prototype in turn tries to make use of an internal prototype `ClientNameField` within `myprot.xdp`. This reference fails to resolve because the XFA processor tries to resolve it in `mytemplate.xdp`.

Example 6.6 Incorrect application of the use attribute within an external prototype.

Fragment from mytemplate.xdp

```
<subform name="root">
   <subform usehref="myprot.xdp#ClientSubform"/>
</subform>
```

Fragment from myprot.xdp (incorrect)

```
<proto>
   <subform name="Client" ID="ClientSubform" ... >
      <field use="ClientNameField" .../>
   </subform>
   <field name="ClientName" ID="ClientNameField" .../>
</proto>
```

The solution is to employ usehref instead of use in myprot.xdp, as follows.

Example 6.7 Corrected external prototype using the usehref attribute.

Fragment from myprot.xdp (corrected)

```
<proto>
   <subform name="Client" ID="ClientSubform" ... >
      <field usehref="myprot.xdp#ClientNameField" .../>
```

```
   </subform>
   <field name="ClientName" ID="ClientNameField" .../>
</proto>
```

It is possible for a template to improperly specify an endless loop of prototype references. For example, in the following template fragment a prototype's child invokes its own parent as a prototype.

Example 6.8 An endless prototyping loop

```
<proto>
   <subform name="Client" ID="ClientSubform" ... >
      <subform name="ClientName" use="#ClientSubform" ... />
   </subform>
</proto>
```

It is the responsibility of the form creator to ensure that there are no endless prototyping loops.

Overriding Prototype Properties

An element that references a prototype is said to inherit all of the attributes, data content and child elements of that prototype. When an element references a prototype, it has the option of overriding what gets inherited. The general rule for inheritance is that a referencing object inherits the following:

- All attributes of the prototype, except the following:
  - The id attribute
  - The name attribute
  - The use attribute
  - Any attributes specifically overridden in the referencing element
- The data content of the prototype, unless specifically overridden
- All child elements of the prototype, unless specifically overridden

Where the referencing element does not explicitly provide values for attributes, child elements, and data content and no such values are inherited from the referenced prototype, application defaults shall apply. The term *absolute omission* describes such an absence of content.

Overriding Attributes

Any attribute present in an element overrides that attribute from the prototype. For example, the following template fragment defines two draw elements whose fonts both reference the 10pt Helvetica prototype. However, the second one overrides the font size with a size of 14pt, and so, it will draw with a font of 14pt Helvetica. In the first draw element's font, the font size was omitted, so it is inherited from the prototype.

Example 6.9 Overriding an attribute

```
<proto>
   <font id="HELV-RED"
      typeface="Helvetica"
      size="10pt"
      weight="normal"
      posture="normal">
      <color value="255,0,0"/>
```

```
   </font>
</proto>
<draw ...>
   <font use="#HELV-RED"/>
   <value>
      <text>Helvetica 10pt</text>
   </value>
</draw>
<draw ...>
   <font use="#HELV-RED" size="14pt"/>
   <value>
      <text>Helvetica 14pt</text>
   </value>
</draw>
```

As implied in the previous paragraph, an attribute is considered to be omitted only if it was not explicitly specified with a value on an element. An attribute that is explicitly specified on an element with the value of an empty string is not considered to be omitted; as should be obvious, the attribute is specified as having the value of an empty string, which signifies the default.

Overriding Data Content

The presence of data content in a referencing element overrides data content from the prototype. For example, in the following template fragment the text value of the field will be "`Overriding text`".

Example 6.10 Overriding content

```
<proto>
   <text id="TEXT"/>default TEXT</text>
</proto>
<field ...>
   <value>
      <text use="#TEXT">Overriding text</text>
   </value>
</field>
```

Note: It is not possible to override prototype data content with empty data content.

Overriding Child Elements

When both the referencing element and the prototype contain child elements, those child elements are matched first by type and then by ordinal number within type. If the prototype has a child element of a particular type and the referencing element does not, the referencing element inherits the child from the prototype. When the child is present in both, the prototype's child acts as a prototype for the referencing element's child. In other words, the referencing element's child will inherit attributes and grandchild elements from the prototype's child, as long as it doesn't override them. The following example has a field that inherits from a prototype field element.

Example 6.11 Overriding child elements

```
<proto>
   <field id="DEFAULT-FIELD">
      <font typeface="Helvetica" size="10pt" weight="bold">
         <color value="255,0,0"/>
      </font>
```

```
      <value>
        <text/>
      </value>
    </field>
</proto>
<field use="#DEFAULT-FIELD" name="num" x="1in" y="1in" w="1in" h="14pt">
    <border>
      <edge thickness="1pt"/>
    </border>
    <font typeface="Times" size="12pt"/>
</field>
```

It's interesting to examine the treatment of four child elements:

- Child `ui` element: Omitted from both the referencing field and the prototype. Application default applies.
- Child `border` element: Present in the referencing field, but omitted from the prototype. Referencing field's border element applies, along with its child, edge element. Application defaults are invoked for any omitted border attributes.
- Child `value` element: Omitted from the referencing field, but present in the prototype. Referencing field inherits prototype's value element and its child text element.
- Child `font` element: Present in both the referencing field and the prototype. Referencing field's child font element inherits from prototype's child font element.

The last case is of special interest. Because a child `font` element is present in the both the prototype and the referencing field, we can recursively view the prototype's `font` element as being a prototype for the referencing field's `font` element. In consequence the referencing field will have a font of Times 12pt bold, colored red.

When an element can have repeating child elements, overrides are matched by ordinal number. For example, consider the following prototype `border` element with two `edge` children.

Example 6.12 Overriding children by ordinal number

```
<proto>
    <border id="DEFAULT-BORDER">
      <edge thickness="2pt"/>
      <edge thickness="1pt"/>
    </border>
</proto>
<field ...>
    <border use="#DEFAULT-BORDER">
      <edge thickness="3pt"/>
    </border>
    ...
</field>
```

The two `edge` children of the prototype `border` are taken as the top/bottom and left/right edges. Using the prototype without any overrides would therefore result in 2-pt edges along the top and bottom borders, and 1pt edges along the left and right. The prototype reference, however, overrides the first `edge` element. So, the result would be 3-point edges along the top and bottom of the border and 1-point edges at the left and right.

Forms with Repeated Fields or Subforms

Static non-XFAF forms (also known as old-style static forms) may have fields and/or subforms that repeat, that is, they may have multiple fields or subforms with the same name. This is used for lists of data. For example, consider the membership list form which is printed as a blank (the result of an empty merge), at right. To make subsequent illustrations easier the form has been cut down to a bare minimum, nevertheless it illustrates the principles.

Empty Static Form as Printed

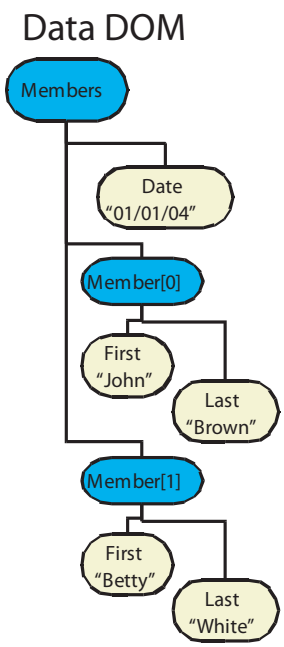
The number of members varies from one year to the next, but the form has a fixed number of places for members' names. (In this example the list is reduced to three to reduce the DOM sizes, but it could be any number.) In addition there is a date field. When some data is merged with the form and the result is printed, the result is shown at left.

Filled Static Form as Printed

As shown (left), this year the club has only two members. The complete data document follows.

Example 6.13 Membership data for the garden club

```
<?xml version="1.0"?>
<Members>
  <Date>01/01/04</Date>
  <Member>
     <First>John</First>
     <Last>Brown</Last>
  </Member>
  <Member>
     <First>Betty</First>
     <Last>White</Last>
  </Member>
</Members>
```



Data DOM after loading data with repeated Member data group

When this data is loaded into the Data DOM, the Data DOM has the structure shown at left. The two Member data groups can be individually referenced in SOM expressions as `Member[0]` and `Member[1]`. They are stored in the Data DOM in the same order that they occur in the data document. ("Scripting Object Model" on page 73)

Repeated Subform Declarations

A static template can express repeated subforms in two ways. The simpler way, conceptually, is repeated declarations within the template packet. For example, the template for the garden club membership roster could be expressed as follows.

Example 6.14 Membership roster template using repeated subform declarations

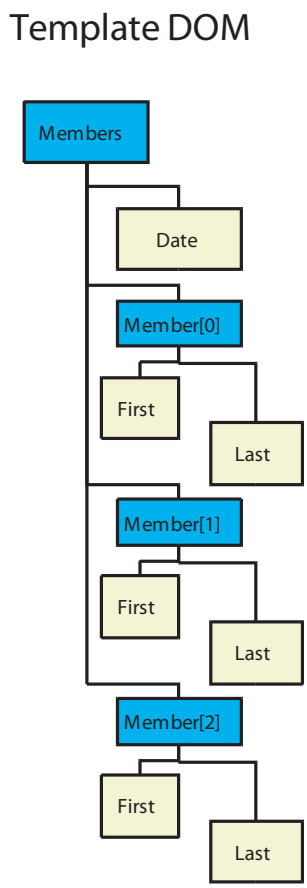
```
<template ...>
   <subform name="Members">
      <field name="Date" ...>...</field>
      <subform name="Member">
         <field name="First" ...>...</field>
         <field name="Last" ...>...</field>
      </subform>
      <subform name="Member">
         <field name="First" ...>...</field>
         <field name="Last" ...>...</field>
      </subform>
      <subform name="Member">
         <field name="First" ...>...</field>
         <field name="Last" ...>...</field>
      </subform>
      </subform>
</template>
```

Note that the template has three `Member` subforms and therefore has room for at most three lines of member information. If the data contains more than three `Member` data groups, only the first three will be bound into the form. Additional data groups will be loaded into the Data DOM but, because they are not

bound, will not normally be processed. It is up to the application supplying the data to subdivide the data into separate documents of the appropriate size. This is an inherent limit of static forms.

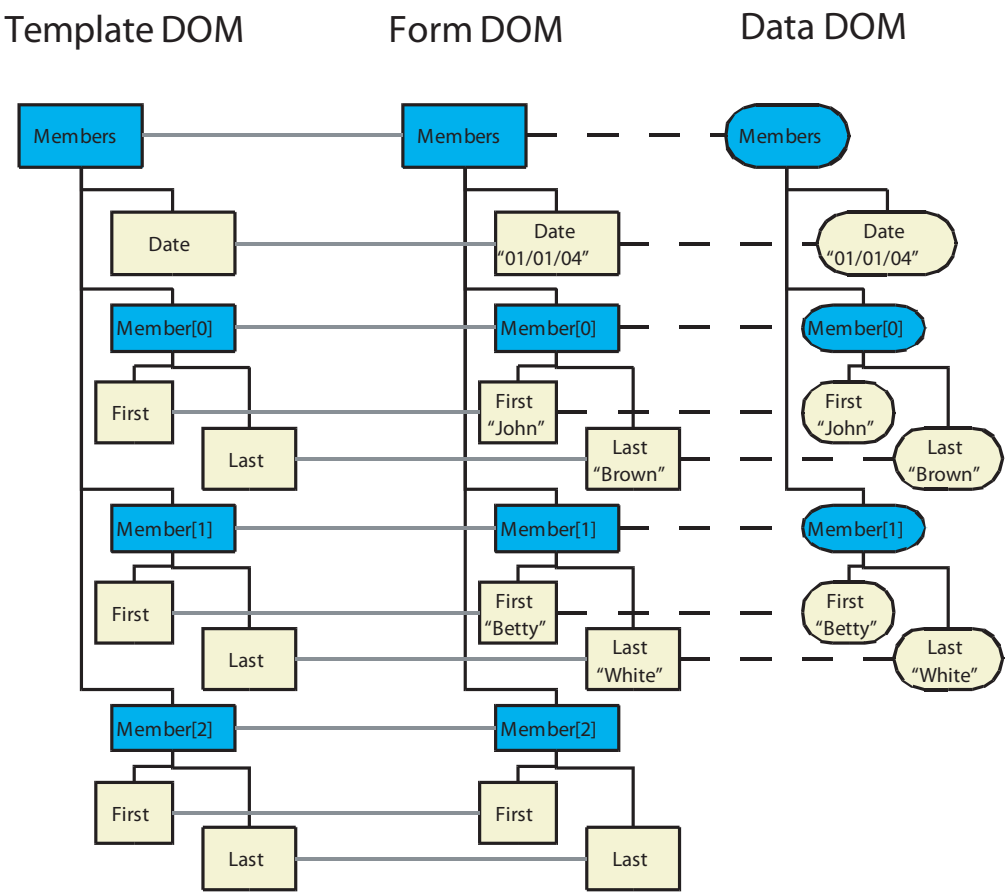
The figure at right shows the Template DOM after the above template has been loaded.

When the template contains identically-named sibling subforms, there are three rules that control which data items are bound to which subforms. First, subforms are copied to the Form DOM and processed in the same order that they occur in the template. Thus The data binding process copies and seeks a match for `Member[0]` first, then `Member[1]`, then `Member[2]`. Second, with one minor exception, each data node is only allowed to bind to a single form node. The exception is discussed below under "Record Mode" on page 211. Third, when searching for a match among identically-named sibling data nodes, the siblings are searched in data document order. The result of these three rules is that matching template and data node pairs are bound in sequence starting with the first of each in document order, as one would intuitively expect. In one possible implementation the data binding process traverses the Template DOM in document order. As it encounters nodes in the Template DOM it copies them into the Form DOM. After adding each node to the Form DOM it seeks a match in the Data DOM, excluding data nodes that are already bound and giving priority to data nodes that are earlier in document order. As described in Matching Hierarchy it seeks a direct match first, then any ancestor match, then any sibling match. When it finds a match it binds the data node to the form node. Then it moves on to the next template node in document order.



Template DOM after loading template with repeated Member subform

The following figure shows the DOMs for the membership form after data binding.



Result of binding repeated data groups to repeated subforms

When the given data and template are bound, the resulting Form DOM contains three `Member` subforms but only the first two are bound to data groups. The data values within those groups are bound to same-named fields of the appropriate subform. Thus the first data value called `First` (with the value "`John`") is bound to the `First` field under subform `Member[0]`, the second `First` (with the value "`Betty`") is bound the `First` field under subform `Member[1]`, and so on. The order of same-named data elements is significant, and the grouping of elements within container elements (data groups) is significant. However, the order in which differently-named sibling data values are placed in the data makes no difference. For example, the same bindings would have been produced if the data document had `First` and `Last` interchanged within one or more `Member` data groups, as follows.

Example 6.15 Membership data rearranged without affecting the presentation

```
<?xml version="1.0"?>
<Members>
   <Date>01/01/04</Date>
   <Member>
      <First>John</First>
      <Last>Brown</Last>
   </Member>
   <Member>
      <Last>White</Last>
      <First>Betty</First>
```

```
   </Member>
</Members>
```

With the data shown above, the Form DOM contains three `Member` subforms as before. However within the second `Member` subform the `Last` field (containing "`White`") precedes the `First` field (containing "`Betty`"), just as it does in the data.

In addition, if a data value is missing the binding of the other data values is not affected. Suppose that the `First` data value ("`John`") had been missing from the first `Member` data group, as follows.

Example 6.16 Membership data missing a field without affecting the presentation

```
<?xml version="1.0"?>
<Members>
   <Date>01/01/04</Date>
   <Member>
      <Last>Brown</Last>
   </Member>
   <Member>
      <First>Betty</First>
      <Last>White</Last>
   </Member>
</Members>
```

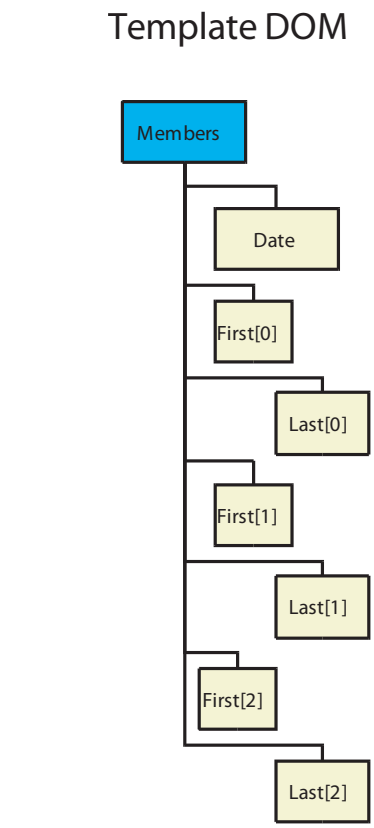
After the bind operation the `First` field under the subform `Member[0]` would have been left unbound, and set to its default value. The `First` field under `Member[1]`, however, would have been bound as before to the `First` data value containing "`Betty`". The `Member` data groups act as containers for the set of related data values, so that the contained data elements are grouped as intended.

Another way to construct a static form is to place repeated field declarations within a single subform. When the template is constructed this way the data must have corresponding multiple data values with the same name within a single data group. The data binding process binds data values to fields in the same order that they are encountered in the data. This binding order results from hierarchy of matching priorities described above in "Matching Hierarchy". For example, in the following template the member detail fields have been placed together in the `Members` subform.

Example 6.17 Repeated fields within a subform

```
<template ...>
   <subform name="Members">
      <field name="Date" ...>...</field>
      <field name="First" ...>...</field>
      <field name="Last" ...>...</field>
      <field name="First" ...>...</field>
      <field name="Last" ...>...</field>
      <field name="First" ...>...</field>
      <field name="Last" ...>...</field>
   </subform>
</template>
```

When this is loaded into the Template DOM, the result is as shown at right.



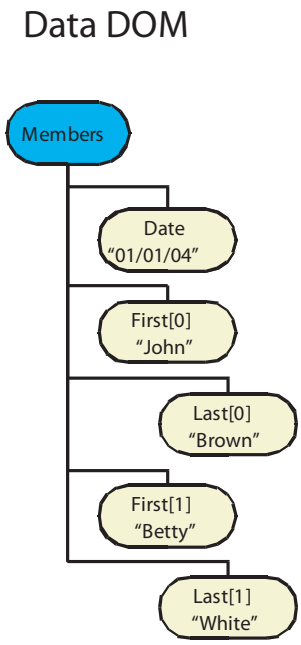
Template DOM with repeated fields within the same subform

Similarly, the data has the corresponding data values directly under the `Members` data group, as shown at right.

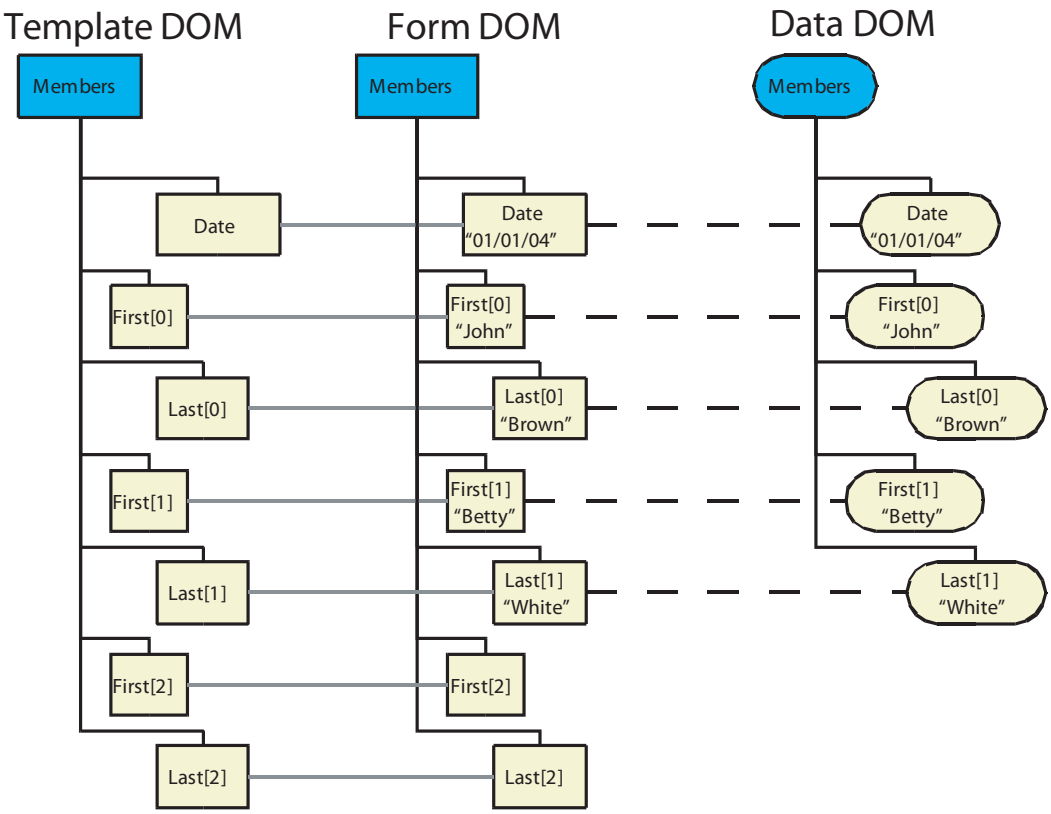
Example 6.18 Membership data flattened.

```
<?xml version="1.0"?>
<Members>
   <Date>01/01/04</Date>
   <First>John</First>
   <Last>Brown</Last>
   <First>Betty</First>
   <Last>White</Last>
</Members>
```

When the Template DOM is merged with the data, the rules of precedence cause field `$form.Members.First[0]` to bind to data node `$data.Members.First[0]`, but field `$form.Members.First[1]` to bind to data node `$data.Members.First[1]`. Similarly each `Last` field binds to its corresponding `Last` data node, which is the desired behavior. The result is shown in the figure below.



Data DOM with repeated data values within the same data group



Result of binding repeated data values to repeated fields

When printed or displayed, the result is the same as the previous example ("Filled Static Form as Printed" on page 202). However this method of constructing the form has an important drawback. Suppose that John Brown's first name is omitted from the data, as in example "Membership data missing a field without affecting the presentation" on page 206. For convenience the example data is reproduced below.

```
<?xml version="1.0"?>
<Members>
   <Date>01/01/04</Date>
   <Last>Brown</Last>
   <First>Betty</First>
   <Last>White</Last>
</Members>
```

In this case, when data binding takes place, the data value named `First` (containing "`Betty`") is bound not to `$data.Members.First[1]` but to `$data.Members.First[0]`. The result is that the membership list is printed as "`Betty Brown`" followed by a member with no first name and a last name of "`White`", as shown at right.

Undesirable result when data is missing and data is not grouped

This result comes about because when the data is not grouped there is not enough information for the data binding algorithm to resolve ambiguity. There are two approaches to fixing this problem; either change the data document or use the data regrouping facility in the data loader. The data regrouping facility uses additional information supplied in the configuration to parse a flat sequence of data values and transform it inside the Data DOM into a series of data groups containing data values. See "The groupParent Element" on page 430.

Fixed Occurrence Numbers

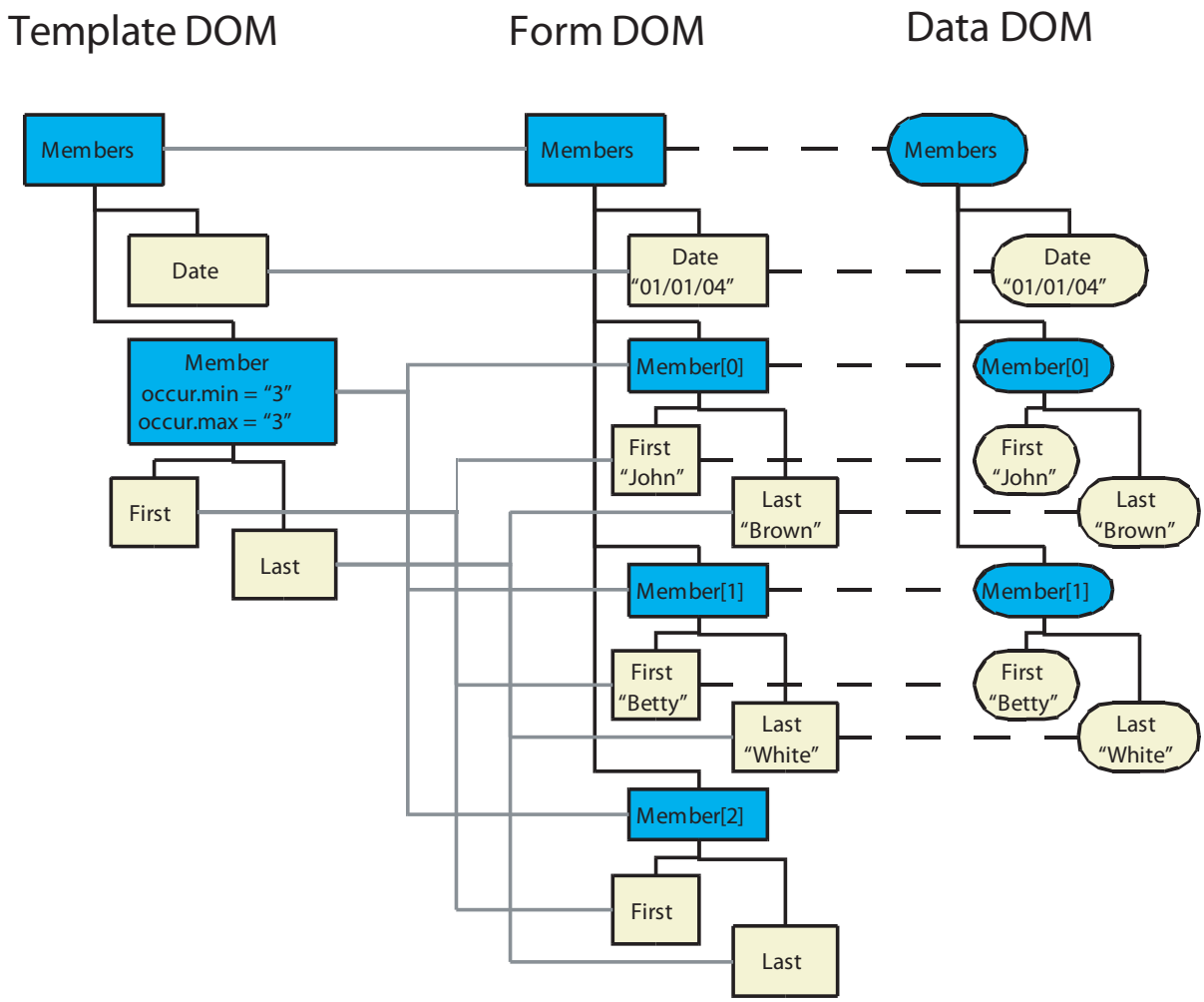
A more concise way to represent static forms of this type is available. Subforms have an `occur` property, which in turn has `max`, `min`, and `initial` sub-properties. By default these sub-properties are all 1 meaning that each subform occurs exactly once. However if they are all set to some other value *N* then the meaning is that the subform occurs *N* times. This makes it unnecessary to repeat the subform declaration *N* times in the template. Thus the membership list example template can be expressed more concisely as follows.

Example 6.19 Using fixed occurrence numbers

```
<template ...>
   <subform name="Members">
      <field name="Date" ...>...</field>
      <subform name="Member">
         <occur min="3" max="3" initial="3"/>
         <field name="First" ...>...</field>
         <field name="Last" ...>...</field>
      </subform>
   </subform>
</template>
```

This is fully equivalent to the earlier representation using three repetitions of the `Member` subform declaration. The Form DOM that results from the data binding operation has the exact same structure

except that multiple subforms in the Form DOM share the same prototype in the Template DOM, as shown in the following figure.



Result of binding repeated data groups to a multiply-occurring subform

As it happens, if the `max` attribute is not supplied then the `max` property defaults to the value of `min`. Therefore the above template can be expressed still more compactly as follows.

Example 6.20 Using a default max attribute with fixed occurrence numbers

```
<template ...>
  <subform name="Members">
    <field name="Date" ...>...</field>
    <subform name="Member">
      <occur min="3" initial="3"/>
      <field name="First" ...>...</field>
      <field name="Last" ...>...</field>
    </subform>
  </subform>
</template>
```

Nested subforms and subform sets can have multiple occurrences at each level of nesting. The result is to compound the occurrences. For example, suppose a template has a subform `Member` which is set to occur three times, and `Member` contains a subform `Name` which is set to occur twice. This is exactly equivalent to a template containing three subforms called `Member`, each of which contains two subforms called `Name`.

Note that fields do not have `occur` properties, hence can not automatically repeat. It is common to wrap a field in a subform simply to provide a way to associate an `occur` property indirectly with the field. In such cases it may be convenient to leave the subform nameless so it does not alter the SOM expression used to refer to the field in scripts. Alternatively, setting its `scope` property to `none` causes it to appear in SOM expressions but to be transparent to the data binding process so it has no effect on the data hierarchy.

The `occur` property is more capabilities that are not depicted here. It can be used to make the form adapt itself to the data, repeating subforms as necessary. See the chapter "Dynamic Forms" on page 286 for a full description of this capability.

Record Mode

XFA processors can operate in two modes, record mode and non-record mode. The choice of mode is determined by option settings described in "The record Element" on page 445 and "The range Element" on page 444. In record mode, the data document is treated as a sequence of records. In the simplest case, each record in turn is loaded, processed, and unloaded before the next record is loaded. Record mode is provided purely as a way to reduce resource consumption (memory and CPU cycles) by XFA processors when dealing with large data documents. Anything that can be done in record mode can also be done in non-record mode providing sufficient resources are available.

In non-record data binding proceeds as described under Forms With Uniquely Named Fields and Subforms. In record mode, for each record, all of the same processing steps except the last (issuing the form ready event) are executed in the same order before moving on to the next record. The last step, issuing the form ready event, occurs only after all records have been processed. Hence the cycle can be described as:

```
For each data record in document order
{
   Create a new Form DOM
   Load globals before the current record into the Data DOM
   Load the current record into the Data DOM
   Create form nodes as copies of template nodes
   Match non-attribute data nodes to form nodes
   Match attributes to unmatched form nodes
   Renormalize the Form DOM
   Perform calculations and validations
   Pass the Form DOM to the layout process
   Delete the Form DOM
   Unload the record from the Data DOM (but keep the globals)
}
Issue the form ready event
```

A record is by definition a subtree of the Data DOM contained by a data group. All records are contained by data groups which are at the same level in the hierarchy of the Data DOM. These data groups may all be peers but they don't have to be – they could have different ancestors. They may optionally be restricted to having the same names, so that data groups at the same level with different names are discarded. Alternatively records can be defined by level alone without any limitation by name.

In the membership list example, record processing could easily be used. Each `Member` data group corresponds to a record. But suppose that there are two classes of members, full members and associate members. In the data full members are represented by a `Member` element while associate members are represented by an `Associate` element. The data document looks like this.

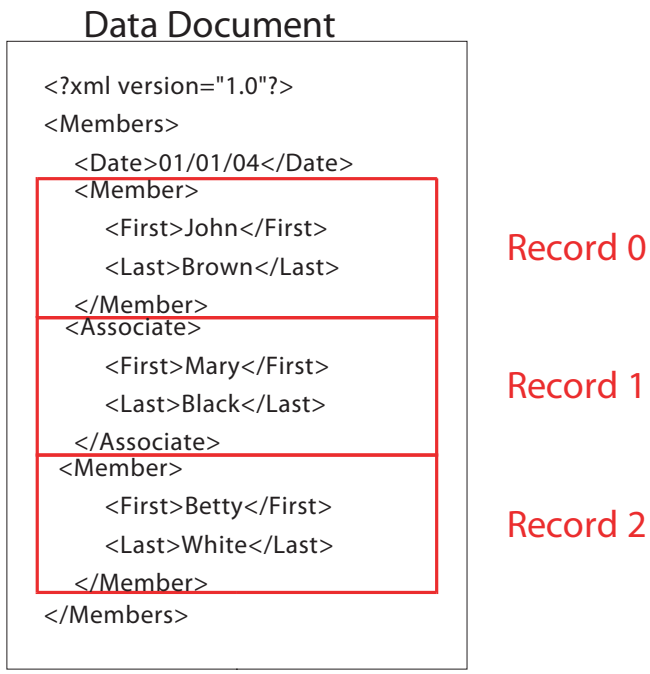
Example 6.21 Membership data with two classes of members

```
<?xml version="1.0"?>
<Members>
   <Date>01/01/04</Date>
   <Member>
      <First>John</First>
      <Last>Brown</Last>
   </Member>
   <Associate>
      <First>Mary</First>
      <Last>Black</Last>
   </Associate>
   <Member>
      <First>Betty</First>
      <Last>White</Last>
   </Member>
</Members>
```

One possible approach is to arrange that all data groups one level below the outermost data group are treated as records. The effect of this is shown at right.

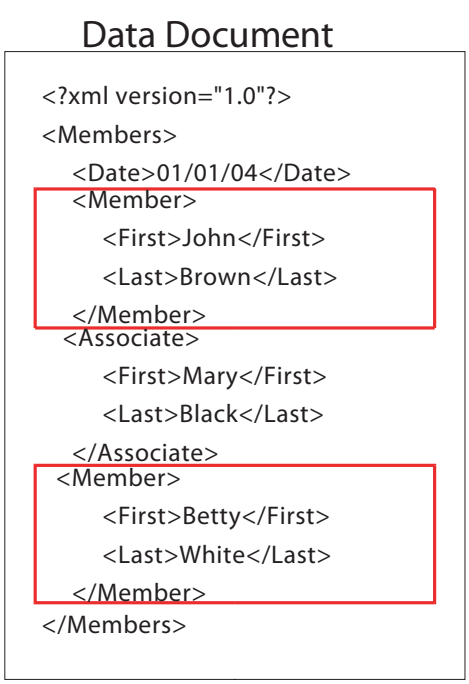
Note that the `date` data value does not count as a record because records are only defined by data groups, not data values.

In this case when records are defined by level in the hierarchy alone, records for both full members and associate members are processed. Note however that during data binding one or the other type of record may still be ignored, if it does not match structure in the template. For example, if the template does not contain any `Associate` subforms the associate member records, although present in the Data DOM, will not bind into the Form DOM.



Membership data with records defined by level alone

On the other hand it may be desired to process records for full members only. In that case the associate member records are not processed even if they match structure in the template. To accomplish this the XFA processor is configured to recognize only `Member` data groups as record containers. The effect of doing so is shown below.



Note that non-record mode is exactly equivalent to record mode with the mode options set such that the outermost level in the data hierarchy defines the record. In the example this is the level of the `Members` data group. The result is that the whole data document, apart from the processing instruction, is treated as a single record. The processing instruction is the line saying:

```
<?xml version="1.0"?>
```

Processing instructions are never loaded into the Data DOM either in record more or non-record mode, as specified in "Creating, Updating, and Unloading a Basic XFA Data DOM" on page 108. Hence excluding the processing instruction from the record makes no difference.

What if the data is not divided up into data groups? Suppose the data is in a flat stream of elements as follows.

Example 6.22 Membership data with two classes, flattened

```
<template ...>
   <subform name="Members">
      <field name="Date" ...>...</field>
      <field name="First" ...>...</field>
      <field name="Last" ...>...</field>
      <field name="First" ...>...</field>
      <field name="Last" ...>...</field>
      <field name="First" ...>...</field>
      <field name="Last" ...>...</field>
      </subform>
   </subform>
</template>
```

The XFA configuration options allow for grouping transformations which operate when loading data into the Data DOM. The effect of a grouping transformation is to collect sets of related data values into data groups. This makes it possible to process flat data in record mode. See "The groupParent Element" on page 430 for more information about the grouping transformation.

Globals

In record mode most bindings are constrained to bind a child of the record subform with a child of the record data group. This is appropriate most of the time. However sometimes it is desired to reuse particular data values at different places throughout a form. This can be done using global data. Global data is any data value which is outside of any record and at the same level as or higher level than the record data groups. For example, consider the purchase order data with records corresponding to `Detail` subforms. With this record definition, all of the data values that are not inside `Detail` data groups are global.

Global data can only bind to global fields. A global field is a field with a `match` attribute of `global`. Global fields are used for data which is only present in the data once but is presented multiple places in the form. For example, in a multi-page form it is common for a name or other identifier to be entered on the first page and reproduced on every page. The matching rules for globals are different from regular fields in order to support this usage. If a global field in the Form DOM can not be matched directly, a match is sought among global data values. This applies even if the binding process did not start at the root of the Data DOM, as in an incremental merge. For example, suppose the template is as follows.

Example 6.23 Global fields in a registration form

```
<template name="Registration">
   <subform name="registration">
      <subform name="name">
         <field name="first"><bind match="global"/> ... </field>
         <field name="last"><bind match="global"/> ... </field>
      </subform>
      <subform name="address">
         <field name="first"><bind match="global"/> ... </field>
         <field name="last"><bind match="global"/> ... </field>
         <field name="apt" ...> ... </field>
         <field name="street" ...> ... </field>
         <field name="city"...> ... </field>
```

```
        <field name="country"...> ... </field>
        <field name="postalcode"...> ... </field>
      </subform>
   </subform>
</template>
```

In this example the application first presents the `name` subform to the user and gathers content for `first` and `last`. It uses this information to perform a database query, which returns the data. The application loads the data from the file into the Data DOM. Assume the data is as follows.

Example 6.24 Registration data

```
<?xml version="1.0"?>
<registration>
   <first>Jack</first>
   <last>Spratt</last>
   <address>
      <apt></apt>
      <street>99 Candlestick Lane</street>
      <city>London</city>
      <country>UK</country>
      <postalcode>SW1</postalcode>
   </address>
</registration>
```

Assume also that record mode is enabled and the `address` data value contains the one and only record. Note that the data does not contain any data group corresponding to subform `name`. This is to be expected because the subform contains only global fields.

Now, in order to present the second screen, the application requests a data binding operation with the `address` data group as the current data record. This limits *most* of the binding operation to the portion of the Data DOM below the `address` data group. The data nodes below `address` contain all the information specific to the second screen, so they bind in accordance with the usual rules. However the data for the `first` and `last` fields is not present below the data group `address`. Failing to find a match in the given subset of the data, and seeing that the fields are marked global, the data binding process searches for a global data value to match the fields. It finds the data for the global fields and binds the field nodes in the Form DOM to the appropriate data value nodes in the Data DOM.

When searching for global data, the global data value can anywhere in the data document provided it is not inside a record and it has already been loaded. If the desired global data value comes after the current record it is necessary to adjust the data window to ensure the desired global data value is pre-loaded (see Data Window, below). The search ignores the depth of the global data value in the data hierarchy; instead it treats the Data DOM as though it was completely flattened. If there are multiple global data values matching the field, they are treated like siblings and the data binding process picks the one with the same index as the current record number. If an index is applied and the index is beyond the end of the list, the field is left unbound.

Note that, unlike regular bindings but like explicit data references, there can be bindings from many global field nodes to the same data value node. For example, if in the example the application requests a data binding operation starting at the root of the Data DOM, the resulting Form DOM will have two global field nodes (one from each subform) bound to the `first` data value node, and two other global field nodes bound to the `last` data value node.

Note: Fields in the template sharing the same name **must** either be all global or all non-global. A mixture of global and non-global fields with the same name is not supported. This restrictions prevents

ambiguities when data is round-tripped between client and server, hence merged with the template, extracted and then merged again.

Another difference between global fields and non-global fields is that global fields are strictly *passive*. A global field can never cause its ancestors to be copied into the Form DOM. Instead the field must be dragged in when its enclosing subform is dragged in by a direct or indirect match to a data node. Only then is an attempt made to match the global field to data in the current record, and if that fails to global data.

There is also a difference between the handling of attributes in global data and non-global data. Attributes of global data elements can not bind to fields. All such attributes are ignored by the data binding process.

As stated above, non-record mode is exactly equivalent to record mode with the mode options set such that the whole data document is treated as a single record. Hence in non-record mode there is no such thing as global data and marking a field as global has no effect.

Data Window

The placement of global data matters. If global data is placed after the record where it would be used, the data binding process may not yet have loaded it. This specification does *not* dictate that XFA processors perform two passes over the data document to pre-load globals. Rather, XFA processors support options to control the loading of adjacent records into the Data DOM along with the current record. This is known as a *data window* and it can include any specified number of records before and any specified number of records after the current record. In most cases global data naturally comes near the beginning of the document or just before the first record to use it. In other cases a larger data window is needed to ensure that all needed global data is loaded before it is needed.

Explicit data references may be affected by the data window as well. A data reference can point to a data node in a record other than the current record. If there is currently no match for the explicit data reference in the Data DOM, the data binding process creates a node with default properties that matches the explicit reference. This is the same behavior it displays when the appropriate section of the Data DOM is loaded but there is no node matching the reference. Explicit data references should use `$record` to refer to data in the current record, `$record[-1]` to refer to the previous record, `$record[+1]` to refer to the next record and so on. For these references to work the data window must be set to include the desired adjacent records.

7 Template Features for Designing Dynamic Forms

This chapter describes the template features used to create dynamic forms. Such forms are capable of adjusting to the data, moving, adding, or omitting sections as required. These features can also be used in old-style non-XFAF static forms. Such a form has access to the full XFA template grammar, and looks like a dynamic form to the XFA processor, but the template does not happen to use variable occurrences.

This chapter is a companion to the chapter "Template Features for Designing Static Forms" on page 29, which should be read first. It is intended for use by form designers and others who do not need to understand the more detailed processing guidelines of XFA forms. Accordingly, this chapter uses the terms *elements* and *attributes* to describe template entities, as does the "Template Specification" on page 482. Other chapters in Part 1: XFA Processing Guidelines use the terms *objects* and *properties* to describe such entities. This shift in terminology reflects a transition to describing processing guidelines relative to XML Document Object Model (DOM) representations of an XFA form.

Container Elements

Compared to static XFAF forms, dynamic forms have additional types of containers. These include containers for:

- Fixed content. In a dynamic form the boilerplate must be described in the template so that it can be laid out and rendered at run-time. Fixed content includes text, images, and basic line-drawings.
- Groups of containers. A dynamic form may assert a grouping which has no effect upon the user filling out the form but is useful to the form creator when modifying the form.
- Physical surfaces and regions. Additional information about the partitioning of space upon pages is required so that a dynamic form can be laid out at run time.

These additional containers are discussed below.

Containers of Fixed Content

Containers of fixed content are very similar to the containers of variable content already familiar from static forms. For example, they can have borders and captions. In addition they have `value` properties which represent the content of the element. The difference is just that fixed content cannot be altered by the user or by user data.

Draw

Forms invariably contain fixed content. This content, often referred to as boilerplate, typically provides context and assistance for consumers of the form. A `draw` element encloses each piece of fixed content. A user cannot directly interact with a `draw` element. Refer to the diagram "A simple XFA form" on page 21 for some examples of `draw` elements. Note that call-outs indicate only two of the many draw elements on the form. Note also that `draw` element content is not limited to text. For example, a `line` element is legitimate content for a `draw` element.

The following is an example of a `draw` element that will produce the outline of a rectangle with the dimensions of one-inch square, positioned at an (x,y) coordinate of (0,0).

Example 7.1 Draw element containing a rectangle

```
<draw x="0" y="0" w="1in" h="1in">
   <value>
      <rectangle/>
   </value>
</draw>
```

For more information, please see the syntax description of the draw element.

Containers That Group Other Container Elements

Area

An area is a grouping of form container elements. The grouping itself is not visible, although the elements themselves may be visible. For example, in the diagram "A simple XFA form" on page 21, the vendor name and address data entry elements, along with the corresponding static text elements might be grouped into an area. Areas provide the form creator with a means of organizing elements on a form, so that they may be moved or manipulated as a whole. They have no effect upon the user filling out the form.

An area is itself a container of containers.

The following is an example of an area element that encloses two text fields.

Example 7.2 Area element enclosing two fields

```
<area x="1in" y="2in">
   <field name="ModelNo" x="0" y="0" w="1in" h="12pt"/>
   <field name="SerialNo" x="0" y="16pt" w="1in" h="12pt"/>
</area>
```

For more information, please see the syntax description of the area element.

Containers That Represent Physical Surfaces and Regions

The process by which displayable content is allocated to particular places on the display surface(s) is called layout. The containers and content that are placed onto the display surface have been discussed earlier. This section introduces the elements which represent display surfaces and regions of display surfaces.

Content Area

A `contentArea` element represents a rectangular region of a display surface. This always has a fixed size and a fixed position upon the page.

Page Area

A `pageArea` element represents a single display surface, for instance one side of a printed page. Depending upon the pagination strategy of the enclosing `pageSet`, `pageArea` elements may be restricted to certain contexts such as odd pages only or the first page in a series of pages only. For more information about pagination strategy see "Flowing Between ContentArea Objects" on page 254.

It is the responsibility of the form creator, and the user when printing, to ensure that each individual page is big enough to hold the `contentArea` regions within it.

It is the responsibility of the form creator to ensure that the template contains at least one `pageArea` element with a `contentArea` inside it.

Page Set

A pageSet element represents an ordered set of display surfaces, for example a series of printed pages. A pageSet element may contain any number of child pageSet elements. For example, consider the following template fragment.

Example 7.3 Page set for two pages, each double-sided

```
<pageSet relation="duplexPaginated">
   <pageSet relation="duplexPaginated">
      <pageArea oddOrEven="odd" ... />
      <pageArea oddOrEven="even" ... />
   </pageSet>
   <pageSet relation="duplexPaginated">
      <pageArea oddOrEven="odd" ... />
      <pageArea oddOrEven="even" ... />
   </pageSet>
</pageSet>
```

In the above example each inner pageSet element contains two pageArea elements to represent the two sides of a duplex-printed page. The outer pageSet element groups all the duplex-printed pages together. In this case there are two pages, each printed on both sides, for a total of four printed surfaces.

Types of Layout Elements

Layout operates on layout elements. There are two general classes of layout elements. Any layout element corresponding to a pageSet element, a pageArea element or a contentArea element represents a *physical* display object or a region of a physical display object.All other layout objects are *layout content*. The function of the layout processor is to apportion layout content to and position layout content upon physical display objects.

Layout content can be further subdivided into *displayable entities* and *structure*.

- Displayable layout elements includes those elements which have no other function than to be visually presented upon the display, such as text, images, and geometric figures. Elements descended from draw elements are also classified as displayable because draw elements merely provide packaging around one of the other types of displayable entities. Displayable entities may originate either from the template (boilerplate) or from user-supplied data.
- Structural layout elements embody relationships between displayable entities and/or other structural layout elements. Subform elements and exclusion group elements are examples of structural layout elements. Note that structural layout elements may also be visually presented upon the display, as for example a subform that has a border and/or a fill color.

In the context of layout, displayable layout elements are generally passive. That is, generally they are acted upon by other layout elements but have no effect upon other layout elements except by the simple act of taking up space. Physical layout elements, by contrast, are always active; they both act directly upon and set constraints upon other layout elements. For example, a block of text may flow across multiple contentArea elements and be split up by them. Structural layout elements become active when they possess non-default breakBefore or breakAfter properties. For example, usingthe breakBefore property a structural layout element may state that it must be kept intact, or that it must be displayed on the front surface of a page.

The w (width) and h (height) properties of layout elements are particularly likely to be a source of confusion. The height of a contentArea is a constraint. For example when text being placed into a contentArea crosses the lower edge of the contentArea, the text may be split and only a fragment

placed into the contentArea. By contrast if a height is specified for a subform, it is not a physical constraint but a logical property. Hence the supplied height does not affect the layout or actual size of the subform or its contents; it only affects how much height the layout processor reserves upon the page for the subform. Widths work the other way around. The width of a contentArea is not a physical constraint; the content placed into the contentArea can extend to the right of the contentArea. However the width of a subform may be a physical constraint; text may wrap when it reaches the right or left edge of the subform. (This asymmetry arises from the fact that XFA does not currently support languages such as Chinese that flow vertically with lines stacked horizontally. Probably any future version of XFA that supports such languages will expand the repertoire of contentArea elements to include splitting by width, and of subforms to include wrapping by height.)

The following table summarizes the types of layout elements:

Type	Subtype	Description	Element
physical	N/A	physical display elements or regions thereof	pageSet, pageArea, contentArea
layout content	structural	logical and some physical relationships between layout elements	subform, subformSet, area, exclGroup, field, draw
	displayable	elements visibly presented upon the display	text, image, line, arc, rectangle, barcode, push button, checkbox, radio button, choice list, text edit widget, date edit widget, time edit widget, password edit widget, image picker widget, signature widget

"Layout Objects" on page 1048 contains a table showing the characteristics and capabilities of each type of layout element.

Basic Composition

This section describes aspects of creating a template that are not applicable to static forms. "Basic Layout in Dynamic Forms" on page 227 describes how container elements are placed on a page.

Line, Arc, Rectangle and Border Formatting

A container element may describe formatting characteristics for lines, arcs, rectangles and borders. These characteristics include all those which have already been described in the context of static forms. Dynamic forms have in addition the ability to specify the handedness of the strokes that make up the figure.

Handedness

Any sort of a line, whether it be a line element or a border edge, follows a logical path. This path has zero width. During the process of rendering of the line, however, the application applies a thickness to create a visible line. Handedness provides the forms designer with a means to specify how that thickness is applied to the line.

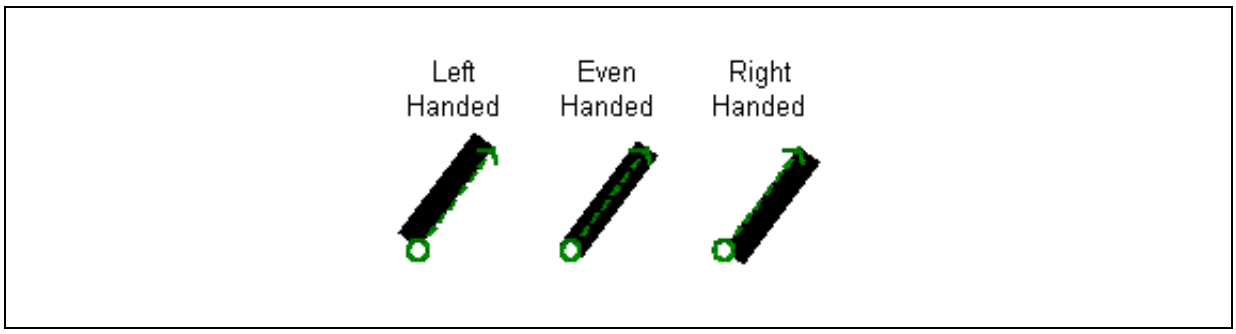
Handedness of Stroke Elements

The edges, and corners elements represent strokes. Many XFA elements that represent graphical elements (such as lines, rectangles, and borders) have outlines that are rendered according to one or more strokes.

These elements posses an attribute which determines the thickness of the stroke, and as the thickness increases the stroke appears to become wider and spread in one or more directions. To understand this, recognize that a stroke is a vector possessing a point of origin, and a measurement representing the length; the imaginary line that extends from the origin along the length is the stroke's path. Therefore, there are three different ways for the thickness of a stroke element to be defined:

- The stroke's thickness extends to left of the path — this stroke is defined as left-handed
- The stroke's thickness extends equally to both the left and right of the path — this stroke is defined as even-handed
- The stroke's thickness extends to right of the path — this stroke is defined as right-handed

The following diagram illustrates the three possibilities, as three thick black strokes along a common path shown in green.



HAND-1 — Edge thickness rendering and handedness

The elements that produce the above diagram are as follows.

Example 7.4 Draw elements containing lines of different handedness

```
<draw x="1in" y="1in" w="0.6in" h="0.8in">
   <value>
      <line hand="left" slope="/">
         <edge thickness="0.2in"/>
      </line>
   </value>
</draw>
<draw x="2in" y="1in" w="0.6in" h="0.8in">
   <value>
      <line hand="even" slope="/">
         <edge thickness="0.2in"/>
      </line>
   </value>
</draw>
<draw x="3in" y="1in" w="0.6in" h="0.8in">
   <value>
      <line hand="right" slope="/">
         <edge thickness="0.2in"/>
      </line>
```

```
   </value>
</draw>
```

Handedness of Borders and Rectangles

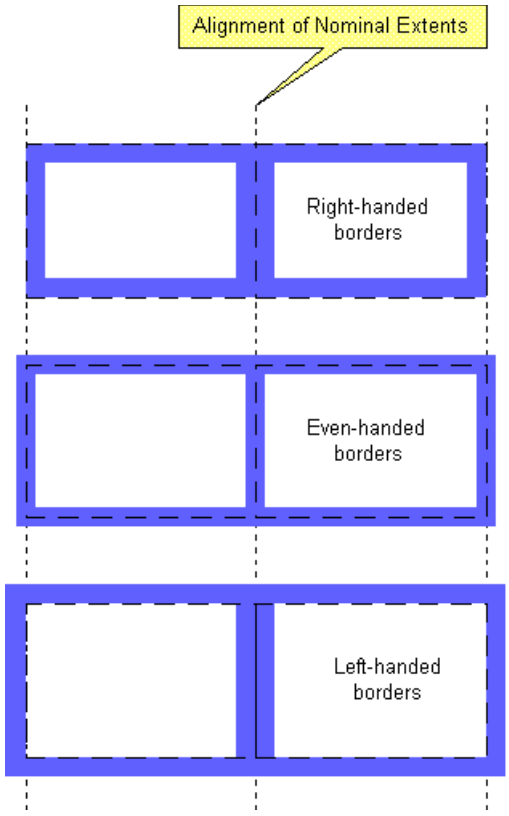
The border and rectangle elements are drawn from the top-left corner, in a clockwise direction. Therefore, a left-handed border will appear to draw immediately outside the border's path; a right-handed border will appear to draw immediately inside the border's path; and an even-handed border will appear to straddle the border's path. Each one of these options has some value to the form designer, who typically designs forms with both container and border margin insets of zero:

- Left-handed borders draw just outside the nominal extent, thereby graphically freeing up the entire nominal extent for content
- Right-handed borders fit within the nominal extent, ensuring that the container element's graphical footprint doesn't exceed its nominal extent
- Even-handed borders allow for alignment of container elements by nominal extent, without unusually thick lines where they join

It is this last point that is of greatest use to a forms designer. If the stroked edges of a border are even-handed, the edges will appear to spread outside the container's nominal extent by half the edge thickness. Placing two objects with this type of border adjacent to each other will result in the common border edge between the two objects, appearing to have the same width as all the other edges — this is very common in traditional form composition.

If the border had been right-handed causing the stroked edges to be rendered completely inside the nominal extent, or left-handed causing the stroked edges to be rendered completely outside the nominal extent, there would appear to be a doubly thick border between the two objects.

This effect of handedness on adjacent bordered objects is illustrated by the following diagram:



HAND-2 — Border handedness

In the above diagram, note how even-handed borders avoid the doubly thick line between the two bordered objects.

The elements that produce the above diagram are as follows.

Example 7.5 Field borders displaying different handedness

```
<field name="field1" x="1in" y="1in" w="1.5in" h="1in">
   <border hand="right">
      <edge thickness="0.125in">
         <color value="128,128,255"/>
      </edge>
   </border>
   <value>
      <text/>
   </value>
</field>
<field name="field2" x="2.5in" y="1in" w="1.5in" h="1in">
   <border hand="right">
      <edge thickness="0.125in">
         <color value="128,128,255"/>
      </edge>
   </border>
   <value>
      <text>Right-handed borders</text>
   </value>
</field>
```

```
<field name="field3" x="1in" y="2.5in" w="1.5in" h="1in">
   <border hand="even">
      <edge thickness="0.125in">
         <color value="128,128,255"/>
      </edge>
   </border>
   <value>
      <text/>
   </value>
</field>
<field name="field4" x="2.5in" y="2.5in" w="1.5in" h="1in">
   <border hand="even">
      <edge thickness="0.125in">
         <color value="128,128,255"/>
      </edge>
   </border>
   <value>
      <text>Even-handed borders</text>
   </value>
</field>
<field name="field5" x="1in" y="4in" w="1.5in" h="1in">
   <border hand="left">
      <edge thickness="0.125in">
         <color value="128,128,255"/>
      </edge>
   </border>
   <value>
      <text/>
   </value>
</field>
<field name="field6" x="2.5in" y="4in" w="1.5in" h="1in">
   <border hand="left">
      <edge thickness="0.125in">
         <color value="128,128,255"/>
      </edge>
   </border>
   <value>
      <text>Left-handed borders</text>
   </value>
</field>
```

Content Types

The representation of fixed content using the draw element is similar to the representation of default content in a field element. The main difference is that fixed content can contain different types of content. Fixed content can be plain text, rich text, an image, or a geometric figure.

Caution: Fixed content can *not* be any of the types that are subject to localization and/or validation. Thus it can not be a `date` element, a `time` element, a `dateTime` element, a `boolean` element, an `integer` element, a `decimal` element, or a `float` element.

Example 7.6 Fixed content that describes a red-filled semicircle, enclosed in a border

```
<draw y="10mm" x="10mm" w="10mm" h="10mm">
```

```
   <border/>
   <value>
      <arc sweepAngle="180">
         <fill>
            <color value="255,0,0"/>
         </fill>
      </arc>
   </value>
</draw>
```

Those content types which differ from variable content are described below.

Lines, Rectangles, and Arcs

The line, rectangle, and arc content type elements enclose geometric shapes. Such content types are meaningful only within draw value elements. In contrast, field value elements should not contain line, rectangle, or arc elements because XFA does not specify a UI widget that would allow a user to provide such geometric shapes.

Example 7.6provides an example of the `arc` content type element.

Images

The image content type element may enclose an image. XFA forms may provide images as fixed content.

Note: The image formats supported by an XFA processing is application dependent.

Background (draw) Images

Template draw elements may contain background images. Each image is embedded in the `image` element as `pcdata` (Example 7.7) or referenced with a URI (Example 7.8).

Example 7.7 Embedded background image

```
<draw name="StaticImage1" y="3.3969mm" x="0mm" w="63.6586mm" h="17.5246mm">
   <value>
      <image contentType="image/tif">
SUkqAAgAAAAVAP4ABAABAAAAAAAAAAAABAwABAAAA0wIAAAEBAwABAAAA6QAAAAIBAwADAAAACgEA
AAMBAwABAAAABQAAAAYBAwABAAAAAgAAABEBBAACAAAAEAEAABUBAwABAAAAAwAAABYBAwABAAAA
…
      </image>
   </value>
</draw>
```

Example 7.8 Referenced background image

```
<draw name="StaticImage2" w="79.12mm" h="28.84mm">
   <value>
      <image
         href="http://www.example.org/pathname/Picture.jpg"
         contentType="image/jpg"/>
   </value>
</draw>
```

Icon buttons

A dynamic form may specify an image and also a text legend for a button. The image is supplied as the field default value and the legend as the field caption. In addition if the button's `highlight` mode is `push` the button may specify different images and legends for each of the up, down, and rollover states. See "Button" on page 406 for more information.

Scaling the Image to Fit the Container (aspect)

Image elements contain a property (`aspect`) that specifies how the image should be adjusted to fit within the dimension of the draw. See "Images" on page 47.

Formatting Text in Dynamic Forms

Within dynamic forms the container into which the text (whether fixed or variable content) is being placed may itself be growable. This adds to the complexity of the text placement algorithms. "Flowing Text Within a Container" on page 52 describes text justification in growable containers.

Repeating Elements using Occurrence Limits

What makes dynamic forms dynamic is that certain containers can be copied into the Form DOM or the Layout DOM more than once. Each copy is known as an *occurrence*. Those containers which can have multiple occurrences take a child property which sets occurrence limits. This property is represented by the `occur` element.

The `occur` element has three attributes, as follows:

- The `min` attribute is used when processing a form that contains data. Regardless of the data at least this number of instances is included. It is permissible to set this value to zero, in which case the container is entirely excluded if there is no data for it. When this attribute is not supplied the internal `min` property defaults to 1.
- The `max` attribute is also used when processing a form that contains data. Regardless of the data no more than this number of instances may be included. It is permissible to set the `max` attribute to -1, in which case there is no limit to the number of instances. When this attribute is not supplied the internal `max` property duplicates the value of the internal `min` property.
- The `initial` attribute is used only when the XFA processor is printing or displaying a blank form. In this circumstance this attribute determines how many instances of the container should be used. When the attribute is not supplied the internal `initial` property duplicates the value of the internal `min` property.

Both `subform` and `subformSet` elements can contain `occur` elements. In these contexts the occurrence limits control how many times they are copied into the Form DOM during a merge operation. The influence of occurrence limits on merge operations is discussed in detail in "The Occur Element" on page 291.

By contrast `pageArea` and `pageSet` elements can also contain `occur` elements, but in these contexts the occurrence limits control how many times they are copied into the Layout DOM during layout. For these elements the `initial` occurrence attribute has no effect. The effects of occurrence limits on layout are described in detail in "Flowing Between ContentArea Objects" on page 254.

It is permitted to set the minimum and maximum occurrences to the same value. If the value is anything other than 1 the effect is to force the full dynamic logic to be invoked but to constrain it so that it operates in a pseudo-static manner. When used this way the result is equivalent to repeating the container element *N* times in the template. For more information see "Using Fixed Multiple Occurrences for Pseudo-Static Forms" on page 307.

Basic Layout in Dynamic Forms

This section describes the most common aspects of how objects are arranged and presented on the presentation medium.

It explains how the objects that appear on a page can be positioned relative to the page and to one another. It explains how contentArea, pageArea, and pageSet can be used to control where content is placed upon each page.

The Layout Processor

The job of the layout processor is to assign each layout object an absolute position on a page. When positioned layout is used for every subform the layout process is simple. Each contentArea is located at an absolute position on a particular page. Each subform is positioned relative to a particular contentArea. The layout objects within the subform are positioned relative to the subform. Hence to compute the absolute position and page of each object the layout processor only needs to trace up the tree of containers, integrating relative positions, until it reaches a contentArea which resolves the relative position to an absolute position and page.

Box Model

The layout processor uses the same box model that has already been described for static forms. There are some additional element types in dynamic forms which take part in the layout process. The box models for those additional element types are described below.

Area

An `area` element represents a grouping of related objects. Area objects grow to the minimum size required to hold the nominal extents of all the layout objects they contain. Area objects do not have margins or borders of their own. These rules make an `area` element largely transparent to the layout process. However when an object within the `area` object uses positioned layout the X and Y positions are specified relative to the `area` object. Hence `area` elements are convenient for bundling objects that are to be dropped into a form and moved around as a unit.

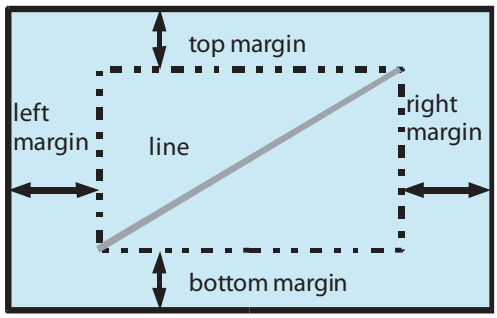
ContentArea Layout Object

A `contentArea` element represents a physical region of a `pageArea` into which content may be placed. The size of a `contentArea` is fixed, hence the `w` and `h` attributes are mandatory.

Geometric Figure

Geometric figures supported by XFA include straight lines, arcs, and rectangles. Arcs and rectangles can be filled. These figures are inherently scalable in the sense that the figure grows or shrinks to fit the content region of the container. However line width does not change with the figure size. The figure at right shows a straight line layout object within its container.

Image with different aspect settings



Line object and its container

PageSet Element

A `pageSet` element represents a set of display surfaces, such as a stack of sheets of paper. Width and height do not apply.

PageArea Element

A `pageArea` represents a display surface, such as one side of a page. Its actual size is unknown (and irrelevant) to the layout processor. A `pageArea` element may contain content such as subforms. Such content, which is placed directly in a `pageArea` element, represents page background. It is up to the creator of the template to ensure that page background and any `contentArea` elements contained in the pageArea do not extend beyond the usable area of the surface.

Size Requirement for Containers

For images, draws containing geometric figures, and contentAreas, the `w` and `h` attributes are mandatory. If either of these is not supplied the layout processor should either infer a size from context (for example from a `minH` attribute) or default the value to 0. Whichever strategy is taken the goal is to continue processing even if the result is unattractive.

Layout Strategies

Dynamic forms may use a *positioned layout strategy* in which every subform is directly linked to a named contentArea. The resulting form can still adapt to the data by including different pages as appropriate. Alternatively dynamic forms can use a *flowing layout strategy* in which content flows through a series of content areas. Both fixed and variable content flow so that the user data is surrounded by all of the appropriate boilerplate. Flowing layout allows a form that contains just those subsections that are needed for the data. In addition flowing layout can interpose such additional elements as headers and footers to enhance the appearance of the form. Finally, dynamic forms can lay out objects aligned in rows and columns to form *tables*. The tables adapt to the size of the individual cells making up the rows and columns.

Dynamic forms may contain a mixture of objects laid out using different layout strategies. In fact this is inevitable because flowing is the only layout strategy available for text inside any container, whereas positioned is the only layout strategy for any container placed inside a `contentArea` region. For all other containers the default layout strategy is positioned but flowing may be specified on a per-container basis.

Note that static forms, because they do not specify any layout strategy, implicitly use a flowing layout strategy for text but a positioned layout strategy for all other objects. The position of each field is specified relative to the top left corner of its containing subform, and the subform is implicitly positioned to the top left of the printable region of the page.

Flowed layout for the special case of text is described in "Alignment and Justification" on page 41. The following section describes the rules for positioned layout. Flowed layout for container objects is described in "Flowing Layout for Containers" on page 241.

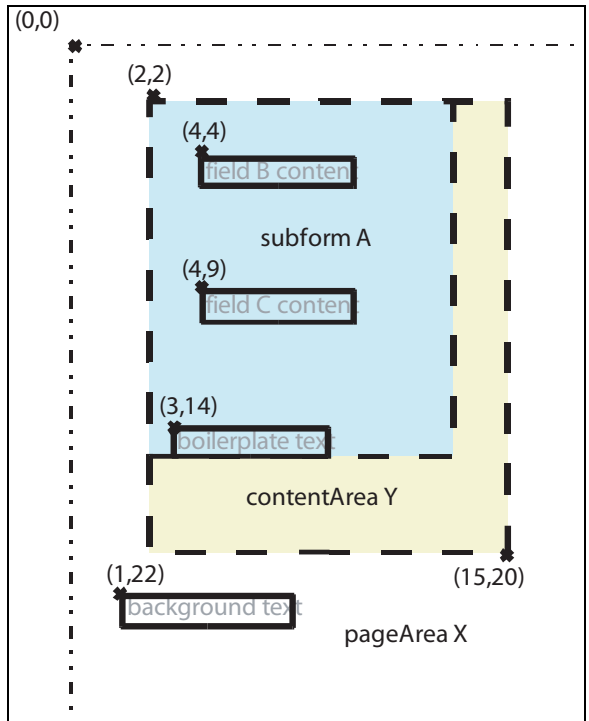
Positioned Layout

When using positioned layout each contained object has a fixed offset vector (an (x,y) pair) which determines the location of the contained object with respect to its container. By default (and typically) the vector gives the offset of the contained object's top-left corner from the container's top-left corner. The offset vector is supplied as properties named `x` and `y`. The `x` property is interpreted as a measurement to the right and the `y` property as a measurement down from the top-left-corner of the container. The values of these properties must be measurements as described in "Measurements" on page 33. If there is no `x` or `y` property for a contained object a value of 0 is assumed by the layout processor. For example, assume a template as follows.

Example 7.9 Template using positioned layout

```
<subform Name="A" layout="position" …>
<!-- root subform -->
   <pageSet …>
      <pageArea name="X">
         <draw name="XBackground" x="1cm" y="20cm" …>
            <text …>background text</text>
         </draw>
         <contentArea name="Y" x="2cm" y="2cm" w="13cm" h="15cm" … />
      </pageArea>
   </pageSet>
   <draw name="ABoilerplate" x="1cm" y="12cm" …>
      <text …>boilerplate text</text>
   </draw>
   <field name="B" x="2cm" y="2cm" …> … </field>
   <field name="C" x="2cm" y="7cm" …> … </field>
</subform>
```

Assume that the data binding process resulted in the text "`field B content`" being associated with field B and "`field C content`" with field `C`. The resulting layout is reproduced at right, with some coordinates labelled. (0,0) is the origin of the `pageArea`. (2,2) is the location of the top-left corner of the `contentArea`, while (15,17) is its lower-right corner. Field `B` is placed 2cm down and 2 cm to the right of the top-left corner of its container, the subform. Hence field `B`'s top-left corner is placed at (4,4). This form also includes background text. Background objects are explained in "Page Background" on page 233. Here the important things to observe are that positioned layout is used for this block of background text and that it is positioned relative to its own container, the `pageArea` itself.



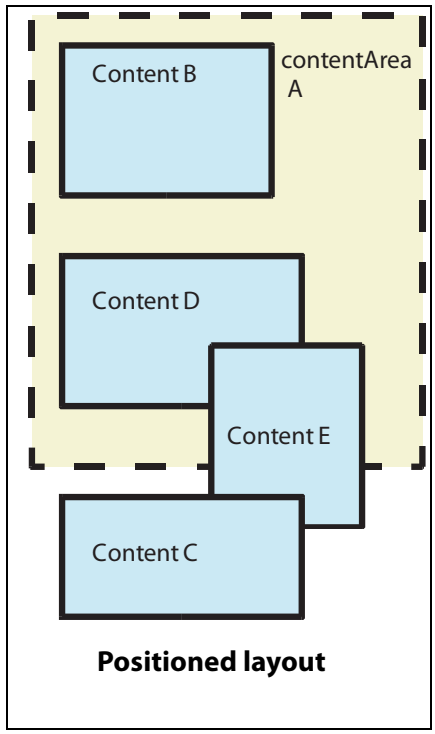
Simple layout example with coordinates

A contained object may specify an anchor point which is the reference point for its offset. The default is its top-left corner. However the offset is always specified with respect to the container's top-left corner regardless of the container's own anchor point. The anchor point is one of the following points within the object's nominal extent:

- top left corner
- middle of the top edge
- top right corner
- middle of the left edge
- center
- middle of the right edge
- bottom left corner
- middle of the bottom edge
- bottom right corner

The anchor point affects the direction in which the contained object grows if its width and height are increased. For example, if the template is modified to increase the height of field `C`, and the anchor point of C is at the top-left, the field extends farther down the page but its top-left corner stays at (4,9). However if its lower-right corner was the anchor point then increasing its height would cause its top-left corner to move up the page while the lower-right corner stayed fixed.

The template may specify offset vectors that cause contained objects to be placed partly or entirely beyond the right or bottom limits of their containers, or to overlap each other. It is not the layout processor's job to second-guess the offset vectors. However, this is no guarantee that overlapped objects will render properly on all output devices. No objects may include `x` or `y` values that are negative or that resolve to negative numbers after conversion from the given anchor point to the top-left corner. In other words, the nominal extents of objects must not extend beyond the top and left limits of their containers. The figure at right shows an example of permitted practice, in which the container's size is fixed and the objects contained within it both overlap each other and extend beyond the nominal extent of the container, but in an allowed direction.



Positioned layout

The layout processor employs positioned layout within any `area`, `pageArea`, or `contentArea` object. It also employs positioned layout within any `draw` object containing an arc, line, or rectangle. And, it employs positioned layout within any subform that has no layout attribute or has a layout property with the value of `positioned`. Exclusion groups are transparent to layout strategy, that is, they inherit their layout strategies from their parents.

Forbidden Condition: Negative Coordinates

The `x` and `y` properties of an object, its anchor point, and its width and height, must not conspire to result in all or part of the object's nominal extent being above or to the left of its container. The result is unspecified. The layout processor may make a best effort to deal with the resulting negative coordinates, but even if it copes that does not guarantee that the renderer will be able to.

Clipping

When a container has a fixed size, the content does not fit into the container, and the layout strategy is positioned, the excess content may either extend beyond the region of the container or be clipped. The permissible range of actions varies according to the type of container and the context (interactive or non-interactive).

When the container is a field and the context is interactive, the content of the field may be clipped. However some means must be provided to access the entire content. For example, the XFA application might arrange that when a field gains focus a widget pops up. The widget could be dynamically sized or it could support scrolling.

When the container is a field and the context is non-interactive (for example printing to paper) the content must not be clipped. The content may be allowed to extend beyond the field or it may be shrunk to fit the field.

When the container is a draw, in any context, the behavior is implementation-defined. It is the responsibility of the form creator to ensure that the region is big enough to hold the content.

Note that clipping does not have to be done by the layout processor. If it is done at all it can be done downstream in the renderer. However it may be advantageous to do partial clipping at the layout stage. For example, when justifying it is more efficient to stop adding text after the last line that is wholly or partly inside the content region.

Locating Containers Based on Data

In a static XFAF form the everything in the form except the contents of fields is fixed. When the template is bound to data (merged), some fields are filled in. Any fields left unfilled are present in the form but empty (or optionally given default data). These types of forms are uncomplicated and easy to design, and suffice for many applications.

Dynamic forms by contrast may be designed with containers that stretch or shrink to accomodate varying amounts of data within individual fields. Containers of this sort are called growable containers. "Layout for Growable Objects" on page 237explains how the content of forms with growable containers is laid out. The need to accomodate variable amounts of data in fixed-size pages imposes a significant burden of additional layout mechanism.

Dynamic forms may also be designed to change structure to accommodate changes in the structure of the data supplied to the form. For example, a section of the form may be omitted if there is no data for it. Such forms are called dynamic forms. From the perspective of the layout process such forms are almost the same as static forms that have growable containers. "Layout for Dynamic Forms" on page 310 describes the differences between layout for static forms with growable containers and layout for dynamic forms.

Page Selection

In static XFAF forms the physical sequence of pages is simply the order in which the pages appear in the PDF. However in dynamic forms the physical sequence of pages is controlled from within XFA using the `pageSet`, `pageArea`, and `contentArea` elements.

Each `pageArea` element along with its contained `contentArea` elements describes the physical layout of one display surface (for example one side of a printed sheet of paper). Any subform can use its `breakBefore` property to specify that it must appear on a display surface described by a particular `pageArea` object, or in content area described by a particular `contentArea` object. XFA provides additional ways to control pagination but for many forms this is all that is needed. The `breakBefore` property is discussed in more detail in "Break Conditions" on page 232.

When a subform asserts a `breakBefore` property the layout processor attempts to satisfy it within the current page. For example, if the subform being laid out specifies a break to a content area named `xyz`, the layout processor looks for an unused `contentArea` with the name `xyz` within the current `pageArea`. (Potentially the page could have many content areas with the same name). If it cannot satisfy the break request using the current page it performs whatever completion actions are required for the current page and then starts a new page and content area that satisfy the request. The new page may be based upon the same `pageArea` as the previous page or a different `pageArea`.

The mechanism by which the layout processor determines which `pageArea` to use next is described in detail in "Pagination Strategies" on page 254.

Break Conditions

In XFA pieces of displayable content are not tied directly to particular content areas or pages. Instead each subform has a `breakBefore` property which controls what the layout processor does when it is about to lay down that subform. It also has a `breakAfter` property which controls what it does after it lays down the subform.

Both the `breakBefore` and `breakAfter` properties have `targetType` subproperties which specify whether the target of the break is a pageArea or a contentArea. For many simple forms the `targetType` subproperty of the `breakBefore` property for each subform is set to `contentArea`. This causes the

layout processor to place each subform into the next available content area on the current page. If there are no more content areas on the page, the layout processor moves on to the next available page within the page set and uses the first content area on that page. There are several different ways in which the next available page can be determined; see "Pagination Strategies" on page 254 for more information. The layout processor maintains a count of instances used for each page set and, when the count is exhausted, ascends a level in the template looking for an unused page set. When it has exhausted the last page set, or if it runs out of displayable entities to lay down, it stops.

The `targetType` subproperty of a `breakBefore` property can also be set to the value `pageArea`. This causes the layout processor to treat the current page as though all of its `contentArea` regions have been used. It advances to the first content area of the next page without putting anything more on the current page. Using this a single `pageArea` can be used to accommodate more or fewer subforms, depending upon the type of subform.

The `breakBefore` or `breakAfter` property may also hold a script. If there is a script the layout processor executes the script to determine whether or not to perform the associated break action. The script must return a Boolean value. If the script returns True the break action is taken. If the script returns False the break action is not taken, that is the `breakBefore` or `breakAfter` has no effect. In the absence of a script the break action is always taken..

Page Background

A `pageArea` may contain subforms, fields, draw objects and area objects. Typically, this is used for letterhead, watermark, and/or identifying data such as a purchase order number. The layout processor makes no attempt to detect or prevent overlaps between background and foreground objects.

Consider the following example.

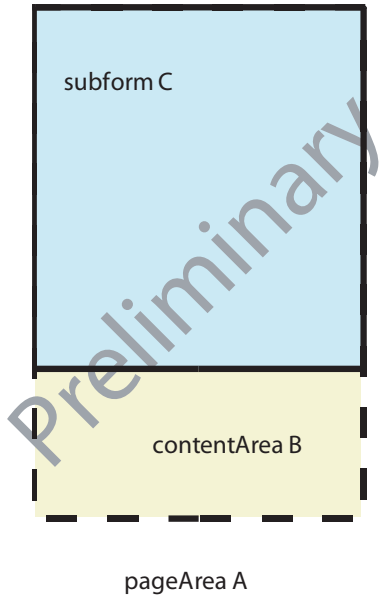
Example 7.10 Template using a background image

```
<template …>
   <subform name="X">
      <pageArea name="A">
         <draw …>
            <image href="Preliminary.jpg" … />
         </draw>
         <contentArea name="B" ID="A_ID" … />
         …
      </pageArea>
      <subform name="C" …>
         <breakBefore targetType="contentArea" target="#A_ID"/>
         <field …> … </field>
      </subform>
      …
   </subform>
</template>
```

The resulting layout incorporates the "`Preliminary.jpg`" image as background in each instance of `pageArea A`, as shown at right.

Structural layout objects used in background content may use either positioned or flowing layout.

Fields contained in page background are somewhat restricted compared to ordinary fields. A field in page background can be a container for data, but it has to be linked to the data by an explicit data reference. See "Explicit Data References" on page 177 for more information. A field in page background can have calculate and validate scripts just as ordinary fields can.



pageArea with background content

Appearance Order (Z-Order)

Z-order applies to fixed content such as draws just as it applies to fields. Draws can overlap fields and vice-versa. As with fields overlapping fields, XFA processors are not expected to do anything to prevent overlaps. It is up to the form creator to prevent overlap if that is what is desired.

Grammar Excluded from XFAF

The following table summarizes those aspects of the XFA Template grammar that are not available to XFAF forms but that are available to dynamic or old-style static forms. While XFAF forms cannot use these features they are not forbidden to express the grammar, however the grammar can only be used to express default behavior. In other words it can only be used when it doesn't do anything. For example the `occur` element can be used but only if it expresses the default values for its properties `initial`, `min`, and `max`. This comes about because in XFA grammars the absolute omission of an element or attribute is almost always equivalent to its inclusion with default values.

Element	Property	Note
any elements	use, usehref	Prototype references are not allowed.
area	x,y colSpan *various content*	No use allowed.
border	hand = "even","left"	Only right-hand borders (inside the widget) are handled by XFA. Even and left-hand borders are delegated to the PDF appearance stream.
caption	Acrobat only supports caption on buttons and barcodes, but other implementations may support captions on other fields.	
contentArea	No use allowed. Pagination is delegated to the PDF appearance stream.	
draw	No use allowed. Draws are delegated to the PDF appearance stream.	
field	border colSpans margin	Relation to the surrounding page is delegated to the PDF appearance stream.
medium	No use allowed. Print control is delegated to the PDF appearance stream.	
occur	No use allowed. Single occurrences only.	
pageArea	No use allowed. Pagination is delegated to the PDF appearance stream.	
pageSet	No use allowed. Pagination is delegated to the PDF appearance stream.	
proto	Prototypes are not allowed.	
script	*instanceManager object*	Scripts must not modify the number or order of objects in the Form DOM.

Element	Property	Note
subform	anchorType allowMacro bookend border break breakBefore, breakAfter colSpan, columnWidths h,w keep layout locale margin minH, maxH minW, maxW occur overflow pageSet para x,y	Allowed but only to express the form logic, not to vary the appearance or number of instances.
subformSet	No use allowed. XFAF forms can only contain static objects.	

8 Layout for Growable Objects

Various layout objects, including fields and subforms, can be made growable. A growable object starts off at a certain size but can stretch to accommodate more content. It is possible to impose a maximum size limit on a growable object, or it may be infinitely elastic.

Placing growable objects upon fixed-size pages presents a problem. Where should the objects be placed? What should happen if a growable object does not fit in the region where it is to be placed? This chapter discusses how these problems are addressed by an XFA layout processor.

Background and Goals

When an XFA processing application produces a document, the physical arrangement of the document may change depending on the supplied data. For example, the supplied text may overflow the boundaries of the associated region of the page into another region. This specification codifies the rules which govern the physical placement of displayable objects upon a page or display screen and the actions which are taken when a displayable entity does not fit in the space allocated for it. It does *not* deal with rendering of the laid-out page on the display device, which is handled downstream in a renderer. Rendering is outside the scope of XFA.

If these rules are followed any XFA layout processor, given the same Template DOM, Form DOM, locale settings and typefaces, will produce the same layout as any other XFA layout processor. This means that all glyphs and other displayed objects will be positioned at the same places and on the same pages. It does not guarantee identical appearance of the finished documents because there may be differences in rendering, for example, between a monochrome printer and a color printer. However, it is expected that on comparable display devices, with the same fonts and page sizes, the alignment of printed entities will be reproduced to within one pixel. This degree of reproducibility is important for some uses of electronic forms, notably when using forms whose appearance is prescribed by law. On the other hand, forms do not require the refined text processing that is used in publishing – kerning and so on. It is more important that the placement of text be fast (because it may be done by an application running on a multitasking server) and, above all, robust.

Some XFA applications do not include a layout processor. This may even be true of client applications that present a form for interactive filling and update. Instead the form may have been already laid out and rendered by an upstream process. In that case the client may (but is not required to) rely upon the pre-rendered image of the form rather than performing layout and rendering for itself. However this is only feasible with a subset of forms known as *static* forms. See "Static Forms Versus Dynamic Forms" on page 286 for a full discussion of the difference between static and dynamic forms. The discussion in this chapter applies to every form when it passes through the layout process, whether the form is static or dynamic.

The reader of this specification will learn how the XFA layout process proceeds, its inputs and outputs, its limitations and options. This information is valuable to implementors who wish to create an XFA-compliant layout processor or generate XFA-compliant templates, and to users of electronic forms who want to understand them in greater depth.

Growable Containers

An XFA template may contain growable container elements, which means the container's size may change. Growable objects are very useful in form processing environments. Growth may occur along either the X and Y axes as follows:

- Growable along both X- and Y-axes. An application of this feature is a region of freeform text on a form with no imposed width to force word-wrapping, and no limit on height.
- Growable along the Y-axis only. An application of this feature is the body text field of a memorandum form where many lines of input causes the field to grow vertically.
- Growable along the X-axis only. An application of this feature is a field that accommodates a product part number of indeterminate length.

The dimensions of non-growable container objects are determined from the outside-in. In contrast, the dimensions of a growable container are determined from the inside-out.

- Non-growable container dimensions. When a form-designer uses a GUI-based template design application to place an container object on the template, the software application typically works form the outside-in. That is, it starts with the rectangle drawn by the designer and applies the margins to determine the available nominal content region.
- Growable container dimensions. The growability of a container object becomes apparent during form fill-in. When a container object is growable, the XFA processing application typically works from the inside-out. It computes a content region from the container's contents and then applies the margins to determine a dynamic nominal extent.

A container object is growable if the following conditions are met:

- *Container supports growth*. The container's content type or ui supports growth. Most container elements with widget properties are growable, but container elements with arcs, lines, or rectangles properties are not. The Appendix "Layout Objects" on page 1048 specifies the content types and ui's that support growth.
- *Container omits one or both fixed size attributes*. The container omits a fixed dimension (`h` or `w`) along one or both axes. The presence of the `h` or `w` attributes with non-empty values fixes the height or width of a container. The absence of those attributes in a growable container indicates the axis/axes along which the container may grown, as described in the following table.

Non-null attribute		Axis along which container grows	Explanation
h	w		
✔	✔	None	If both h and w are non-null, the container is not growable on either axis. Any `minW`/`minH`/`maxW`/`MaxH` values are ignored. An example of a fixed-size container follows: `<field name="text_field" h="1in" w=".5in"/>` **Note:** The default value for `w` and `h` are null, the default value for `minH` and `minW` is 0, and the default value for `maxH` and `maxW` is infinity.
✔		X	If `h` is specified (non-null) and `w` is null, the container is horizontally growable and vertically fixed. Any `minH`/`maxH` values are ignored.

Non-null attribute		Axis along which container grows	Explanation
h	w		

	✔	Y	If w is specified (non-null) and h is null, the container is vertically growable and horizontally fixed. Any minW/maxW values are irrelevant.
			Examples of containers that are growable only in the X axis follows:
			`<field name="A" w="2in" minH=".5in" maxH="3in"/>`
			`<field name="A" w="2in" />`
			`<subform name="B" w="6in" minH="1in" maxH="5in"/>`

		X and Y	If neither h nor w is specified, the container is growable on both axes.
			Example of containers that are growable in both the X and Y axes follows:
			`<field name="A" minH=".5in" maxH="3in" minW="1in" maxW="3in"/>`
			`<field name="A"/>`

For those draws and fields that do not support the notion of growableness (such as arcs and rects) the minW and/or minH are used in the absence of a specified w or h attribute.

➤ **Forbidden condition: value of -1 not allowed for minH, maxH, minW, or maxW**

Specifying -1 as a value of w/h/minH/maxH/minW/maxW is undefined. It does not indicate growableness or infinity.

➤ **Forbidden condition: maxH/W must be null or greater than or equal to minH/W**

If both minimum and maximum widths are supplied for the same element the minimum width must be smaller than or equal to the maximum width. Similarly if both minimum and maximum heights are supplied for the same element the minimum height must be smaller than or equal to the maximum height. However it is anticipated that layout processors will encounter some templates that are not conforming in this way. In such a case the layout processor should emit a warning and swap the offending maximum and minimum.

➤ **Warning condition: maxW/H should be null or non-zero**

The following fragment declares a zero-width container.

Example 8.1 A container with an improper (zero) width

```
<field name="text_field" h="1in" maxW="0"/>
```

Growth and the Box Model

Typically, a growable object grows or shrinks when the geographical size of its content changes. The object responds to the size change by adjusting the size of its nominal content region to appropriately enclose the content. As a result, this triggers a change in the container's nominal extent, to keep the box model consistent. The change in nominal extent may cause the container's parent to adjust itself, possibly changing its own size as well. Note that some of a container's graphical content may be outside the

nominal extent both before and after the size changes. It's up to the object to manage this in a way that is intuitive for users of the object.

It may be helpful to think of transformations as occurring after growth. An object establishes its new nominal content region in the coordinates it is comfortable with. Next, applies the box model embellishment. Only then does it do transformations.

Growth in Growable Positioned Objects

Growth always occurs away from the anchor point in the directions defined by the anchor point type. For example, a topRight anchored object will grow only to the bottom and the left, while a middleCenter anchored object will grow evenly in all four directions.

When a positioned object grows, we say that the growth is visually destructive. That is, the growing object may become large enough to overlap and obscure other sibling objects that occur beneath it according to "Appearance Order (Z-Order)" on page 57. Note that the overlap is apparent only in the presentation of the objects, the obscured objects continue to exist in the form in their entirety.

Text Placement in Growable Containers

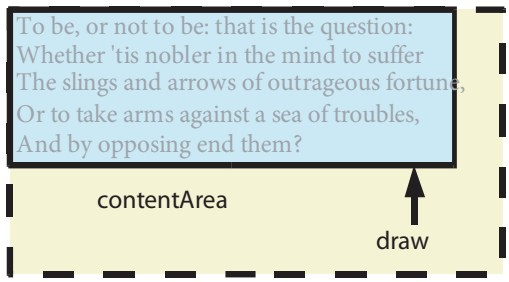
The rules for text placement discussed in "Flowing Text Within a Container" on page 52 are modified when the text is being placed within a growable container. The modified rules are described below. The examples below use the same text as that used in the earlier chapter, reproduced again here:

```
To be, or not to be: that is the question:&nl;Whether 'tis nobler in the mind
to suffer&nl;The slings and arrows of outrageous fortune,&nl;Or to take arms
against a sea of troubles,&nl;And by opposing end them?
```

The sequence "`&nl;`" stands for the newline indicator.

Text Layout with Growable Width

When placing text into a growable-width region, the text records are interpreted as lines. Printable characters in the text are placed sequentially in the flow direction until a newline character is encountered or a width limit is reached. The layout processor then starts a new line and places subsequent characters on the new line. The final width of the region is equal to the width of the longest line plus the width of the caption, if applicable. The caption width is applicable if the caption is placed either at the left or the right of the region.



Extent of a paragraph

The figure at right shows the *layout extent,* or bounding box, of a sample paragraph in solid black. The `draw` object containing the paragraph is being placed into a `contentArea` object. The extent of the `contentArea` is shown in dashed black.

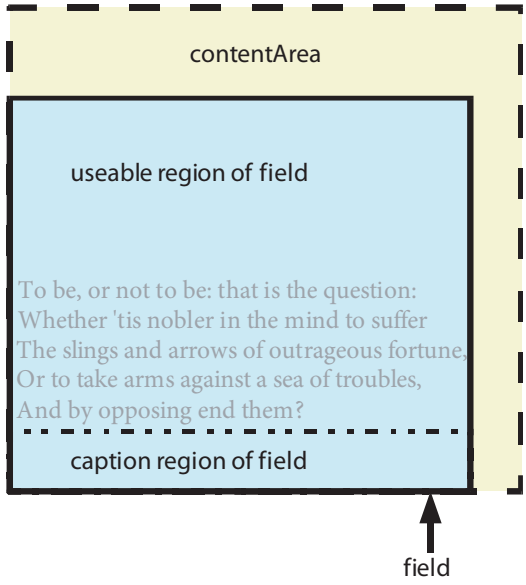
If the field is positioned using its upper-left corner, lower-left corner, or the middle of the left edge it grows to the right. That is, growth occurs by moving the left edge right on the page. If it is positioned using its upper-right corner, lower-right corner, or the middle of the right edge it grows to the left. If the field is positioned using either the middle of its top edge or the middle of its bottom edge it grows equally to the right and left.

The hAlign attribute governs the placement of text within the container. When the container grows to the left and hAlign has a value of left, or the container grows to the right and hAlign has a value of right, the hAlign setting forces the container to grow as much as it can within its own container, even when the text does not fill it. See the following section for an illustration of a similar effect produced in the vertical direction by vAlign.

Text Layout with Growable Height

When the container for the text is growable in the vertical direction, it increases in height as required to accommodate the text within its usable region. The usable region excludes the caption region, if any.

If the field is positioned using its upper-left corner, upper-right corner, or the middle of the top edge it grows downward. That is, growth occurs by moving the bottom edge down on the page. If it is positioned using its lower-left corner, lower-right corner, or the middle of the bottom edge it grows upward. If the field is positioned using either the middle of its left edge or the middle of its right edge it grows equally up and down.



Effect of the vAlign attribute

The vAlign attribute governs the placement of text within the container. When the container grows downward and vAlign has a value of bottom, or the container grows upward and vAlign has a value of top, the vAlign setting forces the container to grow as much as it can within its own container, even when the text does not fill it. In the illustration at right, the paragraph has a vAlign attribute of bottom, an hAlign attribute of left, and a region reserved for a caption. The field was placed within the contentArea using positioned layout, which set the position of the field's top-left corner. However as the field is growable and bottom-alignment was specified, the field has grown to the bottom of the contentArea. The user-supplied text in the field is positioned just above the caption region. There is not enough text to fill the usable region of the field.

Flowing Layout for Containers

Positioned layout, as discussed in "Layout Strategies" on page 40, works best when the size of each layout object is known in advance. This is not possible for growable objects. If positioned layout is used for a growable object then there is a danger of the object either being clipped or overlapping other objects. Hence forms using growable objects generally use flowing layout within the container for the growable object. Usually the container is itself growable, so its container in turn uses flowing layout, and so on.

In flowing layout the contained objects (or in some cases upper parts of objects) are placed sequentially in abutting positions until they have all been placed or the container cannot accommodate the next contained object. In this type of layout the contained object's x and y properties, as well as its anchor point, are ignored.

Flowing layout may not be used by all containers. Some containers (pageArea and contentArea objects) always employ positioned layout for their contents. A second group (areas, subforms and exclusion groups) inherit the layout strategies of their containers. Subforms, by contrast, have a layout

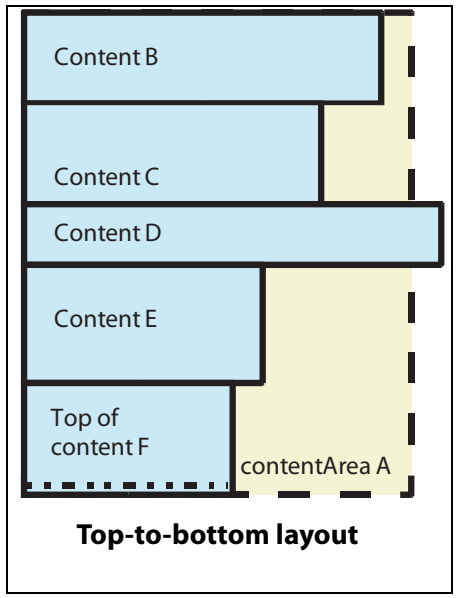
property which specifies the type of layout to be used for the content of the subform. The contained objects (children of the subform node), if they are themselves containers, may in turn specify some other layout strategy within themselves. If a subform element does not have a `layout` attribute it defaults to positioned layout. Finally, fields and draws containing text always use flowing layout.

It is important to distinguish the layout strategy used by a container for its own content and the strategy controlling the placement of the container itself within the object that contains it. For example the contents of a field always flow but the field as a whole may be positioned or it may flow.

Note that, because the root subform has no container, unlike all other layout object it is not controlled by the layout strategy of its container. Instead the root subform itself is treated as though it (but not necessarily its content) flows across the entire document. For example, if the root subform has visible borders, the borders are displayed on every page.

Top-to-Bottom Layout

In top-to-bottom layout the layout processor attempts to place the first contained object at the top-left of the container. If this succeeds it attempts to place the next contained object immediately below the nominal extent of the previous contained object and aligned with the left edge of the container, and repeats for the remaining contained objects. If it comes to an object that will not fit in the remaining height of the container, it attempts to split the object horizontally as discussed in "Content Splitting" on page 250. If it is able to split the object it places the top fragment of the split object into the current container immediately below the nominal extent of the previous object. In the example at right, content `F` has been split and the top fragment placed into `contentArea A`. Also note that content `D` extends past the right edge of the `contentArea`; splitting is *not* done in the vertical direction.



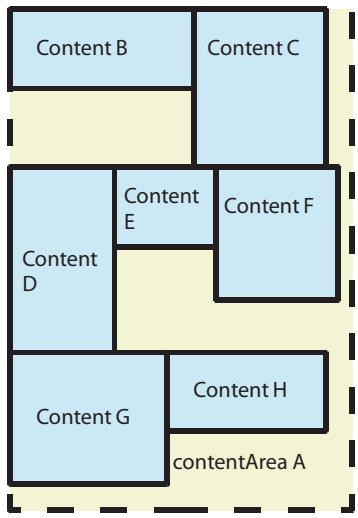
Top-to-bottom layout

The layout processor employs top-to-bottom layout within any subform having a `layout` property with a value of `tb`.

Left-to-Right Top-to-Bottom Tiled Layout

In left-to-right top-to-bottom tiled layout the layout processor attempts to place the first contained object at the top-left of the container. For each of the remaining contained objects, it attempts to place the object immediately to the right of the nominal extent of the previous object, or if this fails immediately below it aligned with the left edge of the container. If it comes to an object that will not fit in the remaining height of the container, it attempts to split the object horizontally as discussed in "Content Splitting" on page 250. If it can split the object, it places the top fragment of the split object into the container and then attempts to place the bottom fragment, treating each fragment as a contained object and following the same rules given earlier in this paragraph.

For example, the document at right has a single container into which seven layout objects have been placed using left-to-right top-to-bottom tiled layout. The layout objects were placed in order from `B` through `H`.



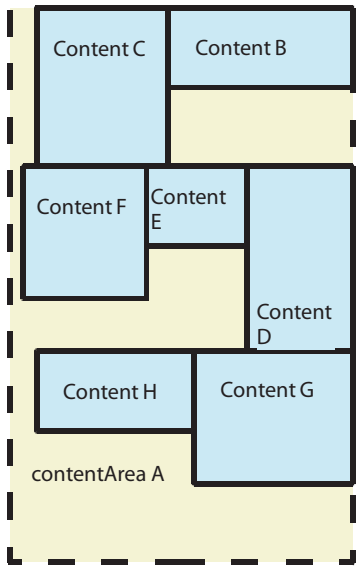
Left-to-right top-to-bottom layout

The layout processor employs left-to-right top-to-bottom tiled layout within any subform having a `layout` property with a value of `lr-tb`. It also defaults to this layout strategy when laying out text within a `draw` or `field` object, unless the `locale` property of the `draw` or `field` names a locale in which text normally flows right-to-left and top-to-bottom. See "Flowing Text Within a Container" on page 52 for more detail about text layout.

When used for text, this type of flowing layout suffices for languages that are written left-to-right, with successive lines stacked top-to-bottom in the European style. It does not suffice for languages written from right-to-left (such as Arabic), nor for languages written top-to-bottom (such as Chinese).

Right-to-Left Top-to-Bottom Tiled Layout

Right-to-left top-to-bottom tiled layout is just like left-to-right top-to-bottom tiled layout, above, except that objects are placed from right to left instead of left to right. For example, the document at right has a single container into which seven layout objects have been placed using right-to-left top-to-bottom tiled layout. The layout objects were placed in order from `B` through `H`.



Right-to-left top-to-bottom layout

When used for text this type of flowing layout suffices for languages that are written right-to-left, with successive lines stacked top-to-bottom in the style of Arabic and Hebrew. However within such languages it is common for sections of included text to run left to right. For example, numbers using Western-style digits are placed left-to-right within the overall right-to-left flow. See for more detail about text layout in right-to-left top-to-bottom locales see "Flowing Text Within a Container" on page 52.

The layout processor employs right-to-left top-to-bottom tiled layout within any subform having a `layout` property with a value of `rl-tb`. It also defaults to this layout strategy when laying out text within a `draw` or `field` object provided the `locale` property of the `draw` or `field` names a locale in which text normally flows right-to-left and top-to-bottom.

➤ Error Condition: Inappropriate Layout Strategy

If a layout container has a `layout` property with a value that is a valid keyword but is not applicable to that container's contents, the layout processor should issue a warning and ignore the offending property. It is likely that some of the restrictions will be relaxed in future versions of this specification. The recommended behavior assures that the form can still be processed, although probably not with the expected appearance. The set of inappropriate layout strategies consists of:

- Positioned layout applied to text
- Flowing layout applied to a geometric figure inside a draw that does not assert `w` and `h` properties (because such a geometric figure has no natural size)
- Flowing layout applied to a subform that is a leader or trailer or is contained within a leader or trailer subform (because the leader or trailer size is fixed)

Interaction Between Growable Objects and Flowed Content

Growable containers can have a dynamic size, which can affect the layout of flowed content simply by changing the size of the container. Whether a growable container is allowed to change depends on whether the form is being used interactively or not. "Growable Containers" on page 238 describes the characteristics of growable containers.

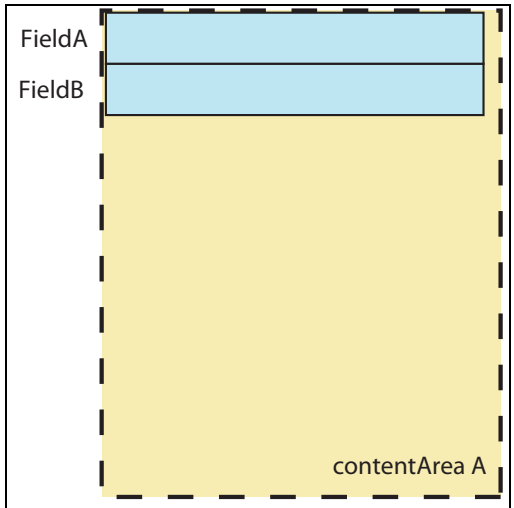
Non-Interactive Form Layout

For initial layout and non-interactive layout, growable containers are re-sized to accommodate the data they contain. The new size of the container is used to determine the containers position, as described "Flowing Layout for Containers" on page 241, with the exception that the new container size is used rather than the original container size.

The following example describes vertically growable fields that have no default value. These fields are contained in a subform with that uses flowed layout (`layout="tb"`). After initial layout, those fields have the appearance shown at right.

Example 8.2 Vertically growable fields

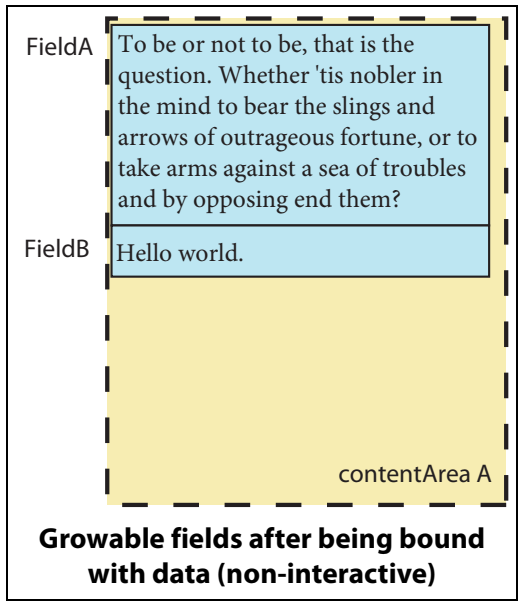
```
<subform name="MySubform" layout="tb">
   <field name="FieldA"
      w="100mm" minH="10mm" maxH="100mm"/>
   <field name="FieldB"
      w="100mm" minH="10mm" maxH="100mm"/>
</subform>
```



Growable fields after initial layout

If a non-interactive bind process associates `FieldA` with data that exceeds the minimum vertical height specified for the field (`minH`), the layout processor grows the field vertically to accommodate the text and rearranges the subsequent fields. The illustration at right shows the vertical growth of `FieldA` and the relocation of `FieldB`, and the following depicts the Form DOM after new data is bound to `FieldA` and `FieldB`.

```
[subform (MySubform) layout="tb"]
   [field (FieldA) = "To be or not to be …"
     w="100mm" minH="10mm" maxH="100mm"]
   [field (FieldB) = "Hello world."
     w="100mm" minH="10mm" maxH="100mm"]
```



Growable fields after being bound with data (non-interactive)

Interactive Form Fill-In

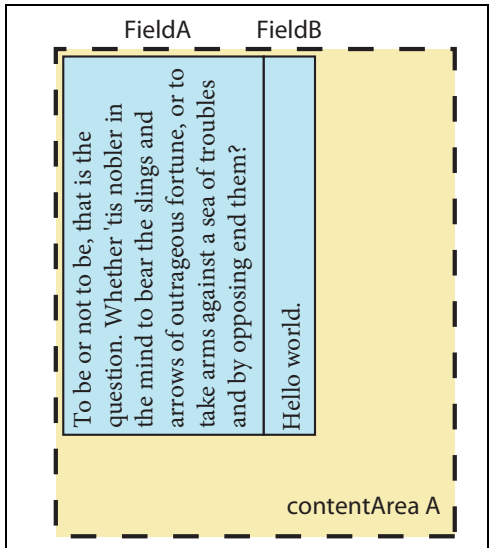
If a person filling out a form interactively provides new data to the growable fields illustrated above, the field is not required to change shape. Rather, the layout processor may clip the new data, as described in "Clipping" on page 50. However if it does so then whenever the field has input focus the user interface provides scroll bars or some other mechanism to make available the full content of the field.

Effect of Container Rotation on Flowed Layout

Flowing layout accommodates rotated containers by using the adjusted axes of the container to determine layout. If both fields in the illustration "Growable fields after being bound with data (non-interactive)" are rotated 90 degrees (counterclockwise), the subform is displayed as shown at right.

Example 8.3 Vertically growable fields rotated

```
<subform name="MySubform" layout="tb">
   <field name="FieldA" rotate="90"
     w="100mm" minH="10mm" maxH="100mm"/>
   <field name="FieldB" rotate="90"
     w="100mm" minH="10mm" maxH="100mm"/>
</subform>
```



Rotated growable fields after being bound with data (non-interactive)

The Layout DOM

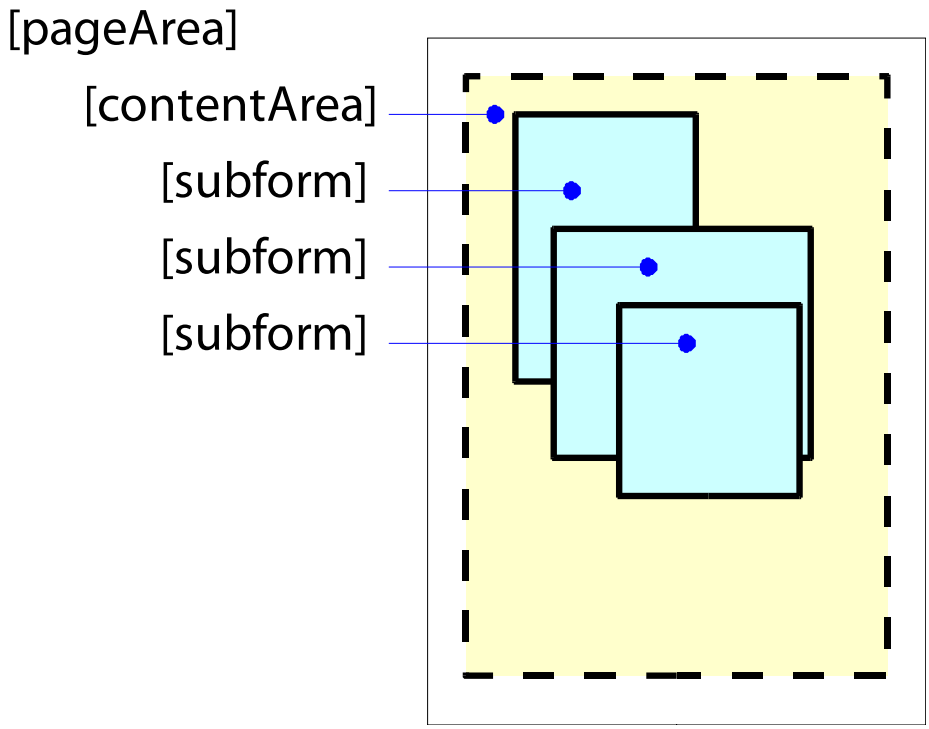
The relationship between layout objects is embodied in the Layout DOM. This is a tree graph representing the result of the layout operation. This includes positioning information for all displayable layout objects. Within the Layout DOM descent stands for relative positioning, so that the child's position on the page is established relative to its parent. Usually the child's displayed representation is also contained within the parent's region of the page, however this is not always the case. The parent is often described as the *container* of the child, even though the child may be rendered outside the parent.

Note: The Layout DOM is an internal DOM used by the layout processor. It is not visible to scripts. The `$layout` object which is accessible via XFA-SOM expressions does not correspond to the Layout DOM but to the layout processor itself. Hence `$layout` is used to control or query the layout processor. For more information about the `$layout` object see the Adobe XML Form Object Reference [FOM].

A node within the Layout DOM is called a layout node. Layout nodes other than the root have the same types as nodes in the Form DOM or in the Template DOM, from which they are copied. Sometimes multiple layout nodes are copied from the same form node or template node; this happens when a single layout content object is split into fragments which are spread across multiple `contentArea` objects.

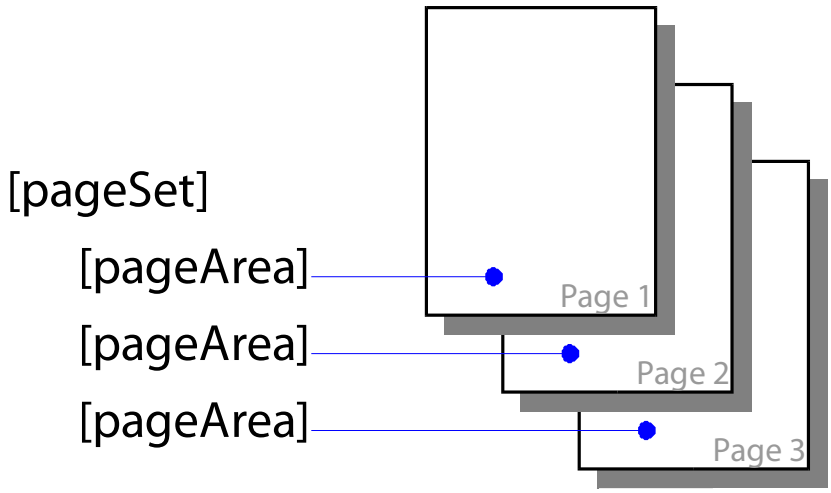
Layout is performed by adding layout nodes to the Layout DOM. The top level below the root holds a single `pageSet`. The next level holds `pageArea` objects. Each new `pageArea` added to the Layout DOM is another display surface added to the document. The next level holds `contentArea` objects. Lower levels of the Layout DOM represent layout content. Each new node of layout content represents a layout object or fragment of a layout object that has been placed upon a display surface. The document order of the Layout DOM is also the Z-order of the layout content; later in document order is closer to the front. You can think of this as the later displayable objects being printed over top of the earlier ones. This is illustrated in the following diagram.

Order in which objects are drawn on page



Among sibling `pageArea` objects, the document order determines the page order; the leftmost (eldest) sibling is the first (lowest page number) in the document. This is illustrated below.

Order in which pages are produced (printed)



Later pages are, by convention, placed toward the back of the document. However printers actually print the lower-numbered pages first. Hence the same principle is observed as for displayable objects; pages are printed and objects are rendered in document order.

The Layout Algorithm

The layout processor is described below and throughout this specification in terms of an algorithm. Note that, notwithstanding the algorithmic description in this specification, conforming implementations are not required to follow the same algorithm internally. They are only required to produce the same results given the same inputs. Similarly, although this specification uses object-oriented terms and concepts, conforming implementations may be written in any language and use any internal structure, as long as they produce the same results given the same inputs. The node structure described here is not necessarily the same structure used by any particular implementation.

The layout processor is content-driven. It makes a single traversal of the content subtree of the Form DOM, adding nodes to the Layout DOM as it goes until all of the content has been processed (or some other termination condition is reached). The layout processor keeps track of a current content node and a current container. The current content node is the node in the Form DOM representing the content currently being processed. A new layout content node is added to the Layout DOM to represent the newly-placed content. The current container is the node in the Layout DOM which will be the parent of the new layout content node.

The starting point in the content is the root subform of the Template DOM. All content is descended from the root subform. The root subform may also specify the starting point in the hierarchy of containers (the starting container). If it does not specify a starting point the starting point defaults to the first `contentArea` of the first `pageSet` which is a child of the root subform. The layout processor recursively copies the nodes from the root subform down to the starting container into the Layout DOM, reproducing the parent-child relationships.

There is a hierarchy of containers. The outermost container is the `pageSet`, which contains `pageArea` objects. `pageArea` objects contain background material such as a letterhead or watermark, as well as `contentArea` objects. `contentArea` objects contain foreground material, including both boilerplate and user-supplied data. `contentArea` objects may directly contain draws, areas, fields, subforms, and subform sets. From this level of the hierarchy down the schema is identical to that defined for the Template DOM. Subforms and areas may directly contain draws and fields. A layout object of any other type must be contained in a draw or a field. In addition areas, subforms, and subform sets may contain lower-level areas, subforms, and/or subform sets nested to any depth.

The layout processor places background ahead of any other content of the same `pageArea` within the Layout DOM. This gives background objects a position within the Z-order such that overlapping foreground objects should be drawn on top. However, there is no guarantee that overlapped foreground and background objects will render properly on all output devices.

As each new content node is encountered it is placed within the current container, unless and until the current container fills up. When the current container fills up, the Layout DOM is extended by adding nodes as necessary and the layout processor traverses to a new node to find the new current container. For example, a template contains the following declarations.

Example 8.4 A simple template using positioned layout

```
<template …>
   <subform Name="A" layout="position" …>
      <!-- root subform -->
      <pageSet …>
         <pageArea name="X">
            <draw name="XBackground" x="1cm" y="22cm" …>
               <text …>background text</text>
            </draw>
            <contentArea name="Y" x="2cm" y="2cm" w="13cm" h="18cm" … />
```

```
      </pageArea>
    </pageSet>
    <draw name="ABoilerplate" x="1cm" y="12cm" …>
      <text …>boilerplate text</text>
    </draw>
    <field name="B" x="2cm" y="2cm" …> … </field>
    <field name="C" x="2cm" y="7cm" …> … </field>
  </subform>
</template>
```

After merging the Form DOM contains:

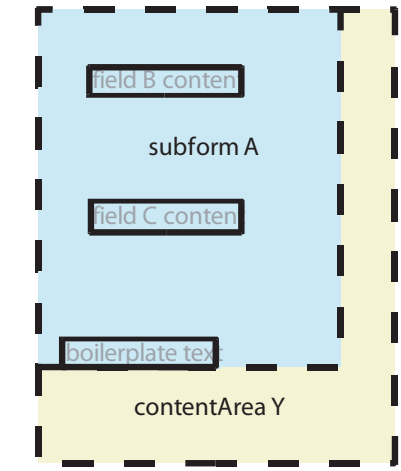
```
[subform (A)]
  [field (B) = "field B content" x="2cm" y="2cm"]
  [field (C) = "field C content" x="2cm" y="7cm"]
  [draw (ABoilerplate) x="1cm" y="12cm"]
    [text = "boilerplate text"]
```

The layout processor starts by copying the `pageArea` into the Layout DOM and then adding a copy of the background text. At this point it is ready to begin inserting foreground objects. It copies the `contentArea` object into the Layout DOM. It initializes the current content node as the root subform, which is subform A. It adds a copy the root subform as a child of the `contentArea`. Then it looks for the next content node, which is field B, so it adds field B into the Layout DOM as a child of the subform. Continuing with the children of the subform, it places field C and the boilerplate into the Layout DOM. The resulting Layout DOM contains:

```
[root]
  [pageSet]
    [pageArea (X)]
      [draw (XBackground) x="1cm" y="22cm"]
        [drawText = "background text"]
      [contentArea (Y) layout="position"]
        [subform (A)]
          [field (B) = "field B content" x="2cm" y="2cm"]
          [field (C) = "field C content" x="2cm" y="7cm"]
          [draw (ABoilerplate) x="1cm" y="12cm"]
            [text = "boilerplate text"]
```

In this case the Layout DOM is not much more than a reordering of the Form DOM. This example is simple because all of the content fits into its assigned region of the page. The reordering is done to give each displayable entity its proper Z-order. The background text is placed ahead of its peers in the Layout DOM so that when rendered it will be on the bottom, overlaid by anything else that may happen to overlap it. When putting children of a `pageArea` into the Layout DOM the layout processor places displayable entities that are immediate children of a `pageArea` ahead of any other children. In all other cases the layout processor retains the document order of the DOM from which it is copying so that the Z-ordering is preserved.

The resulting page from the example would be rendered as shown at right.



Simple layout example

Error Handling

A layout processor may encounter a template which does not conform to this specification. The template may simply be defective, or it might have been constructed using a newer version of the specification. In the event of template non-conformity it is desirable for the layout processor to emit a warning but keep on processing and produce output, even if the visual appearance is degraded. This fail-soft capability ensures that data can still be viewed and operated on if at all possible. This specification describes recommended behaviors for coping with non-conforming templates.

Content Overflow

When a layout object is too tall to fit into the remaining vertical portion of a content region, any of several things can happen. The layout processor may place the object into the current region even though it does not fit, either clipping it or allowing it to extend past the limits of the region. It may decide not to place the object into that content region at all, deferring it to the next content region. Or it may split the object, putting the top of it into the current region and the rest into the next available region.

Caution: Deferral and splitting are characteristics of a flowing layout strategy. However a layout object may be a candidate for deferral or splitting even if its immediate container practices positioned layout. This comes about when the container is itself positioned by its container using flowing layout. Indeed it can happen when any container of the object (any ancestor in the Layout DOM) uses flowing layout. The individual object has a single position relative to the other positioned contents of its immediate container, but the entire container may be split as it flows. Splitting the container may split the contents.

Clipping, including shrinking the content to fit into the region, does not have any explicit controls. The rules governing clipping are described in "Clipping" on page 50.

Deferral can be controlled via the `breakBefore` or `BreakAfter` property of an object. The constraints specified by this property can cause the object to be directed to a particular content area. See "Break Conditions" on page 232 for more information. Deferral can also be constrained by a requirement that adjacent subforms be kept together. See"Adhesion" on page 265 for more information.

An object can be protected from splitting by placing an explicit constraint upon it. In addition, different types of objects are splittable only in certain places or not splittable at all. The rules for splitting are described in "Content Splitting" (below).

Content Splitting

Splitting is not trivial. Splitting within a top or bottom margin is not allowed because it would defeat the purpose of declaring the margin. A simple multiline block of text cannot split at just any height, because

most split lines would cut through characters of text. The multiline block of text can only split at discrete locations that fall between the lines of text. Some other objects (such as images) cannot split at all. The details of where and how various objects can split are described below.

Split Restrictions

Splitting is always forbidden within top and bottom margins. In addition, some types of content can not be split. The restrictions applying to different types of content follow.

`Barcode`

No split is allowed.

`Geometric figure`

No split is allowed.

`Image`

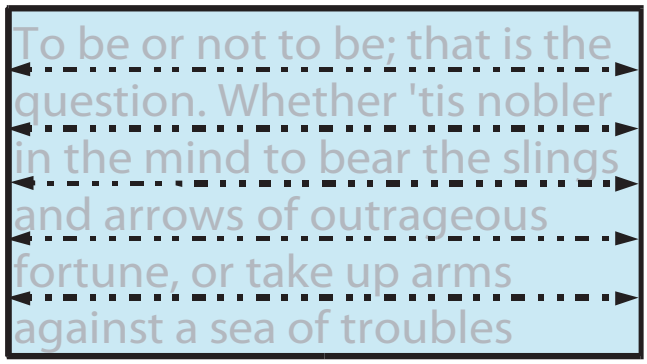
No split is allowed.

`Text`

Text includes anything consisting of printable characters and (optionally) formatting information. Hence it includes the content of numeric fields and of date, time, and date-time fields as well as text fields. Editable fields may take on a different appearance temporarily while they have the focus. For example, a date field may appear as a calendar widget with clickable dates. Layout is not concerned with this temporary appearance. Layout deals only with the non-focus appearance, which is also the appearance in non-interactive contexts such as when the form is printed on paper.

Variable text (i.e. text residing in the Data DOM) is splittable below any line, at the position of the lowest descender from the line. In the figure at right, the dashed lines represent possible split points between lines.

Note: Text within rotated containers cannot be split.



Split lines within text

`Widget`

Widgets include buttons, check boxes, radio buttons, choice lists, and signature widgets. Widgets may take on a different appearance temporarily while they have the focus. Layout is only concerned with the non-focus appearance, which is also the appearance in non-interactive contexts such as when a form is printed on paper. No split of the non-focus appearance of a widget is allowed.

In addition to the above inherent constraints, an explicit constraint may be placed upon an individual subform restricting when it can split. A subform object possesses a `keep` property. The `keep` property has

an `intact` sub-property which controls splitting. This can have any of three settings. `none` means that the layout processor is free to split the subform wherever it can. `contentArea` means that the layout processor is not allowed split the subform. Instead the subform is placed in a single `contentArea`. `pageArea` means that the layout processor may split the subform but not across pages.

Note that the default value for the `intact` property varies depending upon context. If the subform is a row in a table ("Tables" on page 281), the default is `contentArea`. This is also the default when the subform's container's layout strategy is positioned. Otherwise the default is `none`.

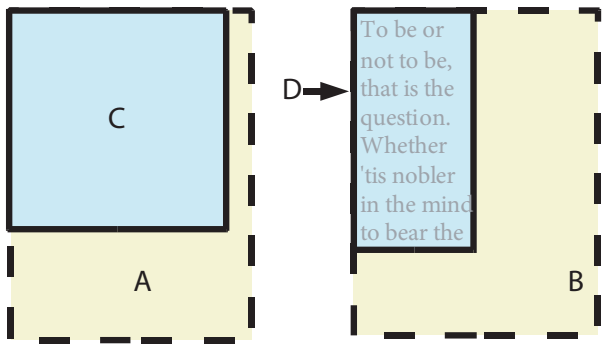
When the layout processor encounters a keep-intact constraint for the current subform it simply treats the subform as unsplittable. The current subform's container is prevented from splitting anywhere within the layout extent of the current subform.

For example, the following template declares a subform that is to be kept intact.

Example 8.5 Template with a keep-intact requirement

```
<template …>
   <subform name="root" layout="tb" …>
      <pageSet …>
         <pageArea …>
            <contentArea name="A" … />
            <contentArea name="B" … />
         </pageArea>
      </pageSet>
      <subform name="C" … />
      <subform name="D" …>
         <keep intact="contentArea" />
         …
      </subform>
   </subform>
</template>
```

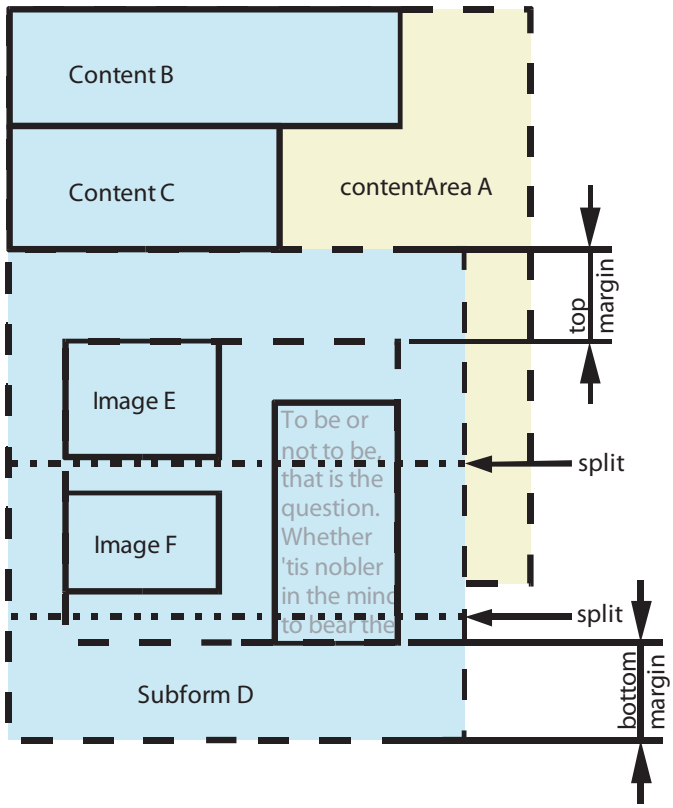
Assume that the subform `D` contains a field holding a multiline block of text and that the layout processor is attempting to place it into the remaining portion of `contentArea A`, but it is too tall to fit. Without the `intact` property the layout processor would have split `D` in between lines and placed the top part into `contentArea A` and the remainder into `contentArea B`. Instead it treats subform `D` as unsplittable and places the whole subform into `contentArea B`, as shown at right.



Effect of keep-intact property

Splitting a Container Having Child Containers

In addition to the constraints upon splitting individual objects, the layout process may be trying to split a contained object which itself contains a mixture of other objects. For example, the object to be split may contain a mixture of blocks of text and images as shown at right.



Splitting a container

In the figure at right, subform `D` cannot split within its margins, because splitting within margins is not allowed. It cannot split within image `E` or image `F` because images are not splittable. Neither can it split below the bottom of `contentArea A` because then the top fragment of subform `D` would extend outside the `contentArea`. The only places it can legally split are between the images at one of the text area's legal split points. The dotted lines with arrowheads show the two legal split positions.

When the object to be split contains other objects, the layout processor finds the most efficient (lowest) split location that is acceptable to all of the contained objects. In other words it finds the optimum consensus. The optimum consensus may be found by the following procedure (written as pseudocode):

```
Start with the current split location set to the desired split location.
While any object in the container cannot split at the current split
location, do the following:
   Set the current split location to the lowest permissible split location
   for that object that is above the current split location
```

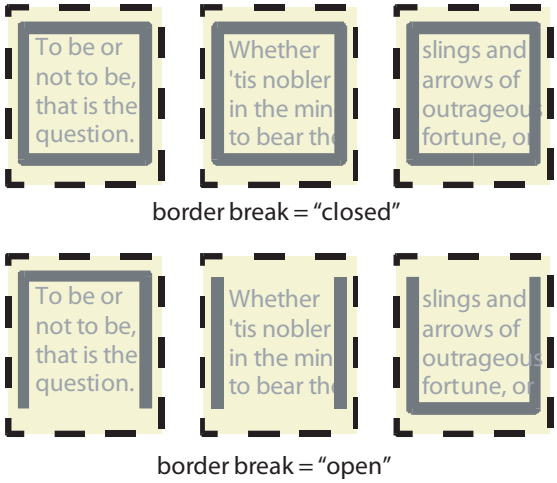
Thus the split location creeps upward until a location is found that is acceptable to all the contained objects. This location may be right at the top of the container (Y offset of zero) in which case the object can not split.

Split consensus is employed when splitting subforms, areas, and exclusion groups.

Borders and Splitting

Border objects have a `break` property that determines what the layout processor does when the object to which the border belongs splits. When `break` is set to `close` the layout processor draws all sides of the border around each fragment of the object. By contrast when `break` is set to `open`, the top border is drawn only over the first fragment and the bottom border only under the last fragment. The figure at right shows the effect of the `break` property.

Recall that borders have no effect upon the nominal extent of a layout object, so although borders are affected by the layout process they have no effect upon anything else.



Effect of the border break property

Flowing Between ContentArea Objects

During flowing layout, when a `contentArea` becomes full, the layout processor moves to a new `contentArea`. It traverses from the current `contentArea` to the new `contentArea` in document order (depth-first, right-to-left). Hence it pours content into the current `contentArea`, then into its next sibling, and so on until all the `contentArea` objects within the `pageArea` are full. At this point the `pageArea` itself is considered full.

When the current `pageArea` is full the layout processor inserts another `pageArea` into the Layout DOM. In order to do this it has to decide what `pageArea` in the template to use as the prototype for the new `pageArea` in the Layout DOM. The method by which it selects the prototype `pageArea` is called its *pagination strategy*. The types of pagination strategies supported by XFA are described in "Pagination Strategies" on page 254.

Some layout objects may assert that they have to be kept together with the preceding and/or following layout object. This is described in "Adhesion" on page 265.

XFA supports the automatic insertion of several kinds of leaders and trailers. Among these are overflow leaders and trailers, which are described in "Overflow Leaders and Trailers" on page 271.

Pagination Strategies

Printers come in two types, simplex (printing on one side of the paper only) and duplex (printing on both sides). Furthermore, when displayed on a monitor a form may be displayed as a series of individual pages (simplex) or as a series of side-by-side pairs of pages (duplex). Often the form creator can not control whether the form is presented in simplex or duplex. XFA 2.5 provides means for the form to adapt to simplex or duplex presentation as required through the specification of a pagination strategy.

The pagination strategy is controlled by the `relation` property of a `pageSet` object. This property takes three values representing the three different strategies.

When the value of relation is `orderedOccurrence` the original pre-XFA 2.5 strategy is selected. In this *ordered pagination* strategy the current content node starts at the root of the content subtree of the Form DOM and traverses the content subtree depth-first and left-to-right (oldest-to-newest). This order is also

known as *document order*, although not all DOMs are ever expressed as XML documents. When using this pagination strategy the same number and sequence of surfaces is printed regardless of printer type. For backwards compatability this is the default pagination strategy.

When the value of `relation` is either `simplexPaginated` or `duplexPaginated` a different approach is taken. When the layout processor chooses the next `pageArea` object it takes into account a number of qualifications specified for the `pageArea`. For example the `pageArea` can be limited to being a front surface or back surface, or to being at a certain position within a contiguous series of pages from the same enclosing pageSet. (This approach duplicates the pagination logic and much of the syntax of XSL-FO [XSL-FO].) The logic for `simplexPaginated` and `duplexPaginated` is very similar and they are both explained in "Qualified pagination strategies" on page 261.

When the current `pageArea` is full, the layout processor moves on to the next `pageArea` object in accordance with the selected pagination strategy. By default, when it has filled the last `pageArea` object, it stops and no more content is laid out. However, it is possible for individual `pageArea` objects, and the `pageSet` object, to be copied multiple times into the Layout DOM. This is controlled by the maximum occurrence property of the `pageArea` or `pageSet`.

It is also possible to make `subform` elements and `subformSet` elements sensitive to evenness or oddness of the page count using the `break` property. This is independent of the pagination strategy.

Determining the start point

Regardless of the pagination strategy there is a standard way for the layout processor to determine which `pageArea` and `contentArea` to start with when it begins processing. The rule is as follows:

1. If the root subform asserts `breakBefore`, start with the `pageArea` and `contentArea` that satisfy its break condition. If the break target is a `pageArea` use the first `contentArea` within it.

2. Else, look at the first subform child of the root subform. If it asserts `breakBefore`, start with the `pageArea` and `contentArea` that satisfy its break condition. If the break target is a `pageArea` use the first `contentArea` within it.

   Note: This rule was added in XFA 2.5.

3. Else, use the the first appropriate `pageArea` within the first `pageSet` child of the root subform. If the layout strategy for the `pageSet` is ordered the appropriate `pageArea` is the first one in document order. If the layout strategy is qualified it is the first qualified one in document order. Start with the first `contentArea` within the selected `pageArea`.

If this procedure arrives at a `pageArea` that does not contain any `contentArea` (a page of pure boilerplate) the layout processor emits the page and then selects the subsequent page using the normal algorithm appropriate to the pagination strategy of the `pageSet`. If that page is also pure boilerplate this step reiterates until a page with a `contentArea` is reached or until all eligible page occurrences are exhausted.

It is a fatal error if this procedure does not yield an appropriate `pageArea`, for example if the root subform asserts `breakBefore` but there is no `pageArea` or `contentArea` matching the supplied target name.

Occurrence Limits in Ordered Pagination

The `pageSet` and `pageArea` elements, like `subform` elements, have `occur` properties. These properties may be used to modify the layout when the pagination strategy is `orderedOccurrence`.

Caution: The `occur` properties are ignored when the pagination strategy is either `simplexPaginated` or `duplexPaginated`.

In the simplest case every page of the form is identical, except for the variable data within fields. In this case the `occur` property of the `pageArea` element can simply be set to the required number of pages. As with subforms the same value is supplied for the `min` and `initial` attributes, and the `max` attribute defaults to the same value. For example in the following template fragment the subform has a `breakBefore` property which causes each instance of the subform to be laid out on a new page. The `breakBefore` property is discussed in "Break Conditions" on page 232.

Example 8.6 A repeating subform with a layout directive

```
<subform ...>
   <pageSet relation="orderedOccurrence" ...>
      <pageArea>
         <occur min="5" initial="5"/>
      </pageArea>
   </pageSet>
   <subform ...>
      <occur min="5" initial="5"/>
      <breakBefore targetType="pageArea"/>
      <draw .../>
      <field .../>
   </subform>
</subform>
```

A sequence of pages can also repeat. This is accomplished using the `occur` property of the `pageSet` element. For example, in the following template fragment each of subform A and B is given a page (and corresponding `pageArea` object) by itself. The pattern of alternating pages is repeated five times, for a total of ten pages.

Example 8.7 A multiply-occurring page set

```
<subform ...>
   <pageSet relation="orderedOccurrence" ...>
      <occur min="5" initial="5"/>
      <pageArea ...>
         <occur min="1" max="1"/>
      </pageArea>
      <pageArea ...>
         <occur min="1" max="1"/>
      </pageArea>
   </pageSet>
   <subform ...>
      <occur min="5" initial="5"/>
      <subform name="A"...>
         <breakBefore targetType="pageArea"/>
         <draw .../>
         <field .../>
      </subform>
      <subform name="B"...>
         <breakBefore targetType="pageArea"/>
         <draw .../>
         <field .../>
      </subform>
   </subform>
```

```
</subform>
```

It is legal to set the occurrence count for a page or set of pages higher than the number of pages used by subforms in the template. When the layout processor runs out of content, it continues appending blank pages as necessary to fulfill the minimum occurrence requirements.

Note that the default occurrence behavior of `pageSet` and `pageArea` objects differs from the default behavior of subform objects. When no maximum or minimum occurrence is supplied for a subform the subform limits itself to exactly one instance. However when no maximum or minimum occurrence is supplied for a `pageArea` or `pageSet` object, the `pageArea` or `pageSet` object allows itself to replicate without limit. In this case the layout processor stops adding new pages or sequences of pages only when it runs out of subforms to lay down.

A pure boilerplate `pageArea` is a `pageArea` element that does not contain any `contentArea` elements. A pure boilerplate `pageArea` must not have a maximum occurrence limit of -1 (whether by defaulting or supplied explicitly). This is because, should the layout processor find its way into such a `pageArea`, it logically should execute an infinite loop emitting pages. This is anti-social behavior and templates are forbidden to do this. For the same reason, a `pageSet` element that contains only pure boilerplate `pageArea` elements must not have a value of -1 for its maximum occurrence property. However it is anticipated that layout processors will encounter some templates that are not conforming in one of these ways. It is recommended that in such a case the layout processor emit a warning and proceed as though the value of the offending `max` attribute was 1. In subsequent processing this could lead to the layout processor using up all allowed occurrences and quitting prematurely, which is annoying but safe behavior.

One might ask why `contentArea` objects do not have `occur` properties. There would be no point, because each instance of the `contentArea` would occupy the same position upon the page. By contrast each instance of a `pageArea` represents a unique display surface and each instance of a `pageSet` represents a unique set of display surfaces.

Algorithm for Maximum Occurrence Limits

When the layout processor finishes filling the last `contentArea` on a page and it is following an ordered pagination strategy, it ascends from the current node in the Template DOM until it comes to a node with a maximum occurrence limit that has not yet been exhausted. This may involve ascending one level to the parent `pageArea` or two levels to the grandparent `pageSet`. When it reaches a node with a maximum occurrence limit that has not yet been exhausted, the layout processor adds a new node of the same type to the Layout DOM in the corresponding position. For example, suppose a template contains the following declarations.

Example 8.8 Page set with maximum occurrence limits

```
<pageSet relation="orderedOccurrence" name="A">
   <occur max="-1"/>
   <pageArea name="B">
      <occur max="1"/>
      <contentArea name="C" … />
      <contentArea name="D" … />
   </pageArea>
   <pageArea name="E">
      <occur max="2"/>
      <contentArea name="F" … />
   </pageArea>
</pageSet>
```

By default the layout processor starts putting content into the first `contentArea` (`C`) on the first `pageArea` (`B`) if the first `pageSet` (`A`). At this point the Layout DOM contains:

```
[root]
   [pageSet (A)]
      [pageArea (B)]
         [contentArea (C)]
```

When `C` is full the layout processor moves to `contentArea` `D`, adding a corresponding node to the Layout DOM. When `D` is full it ascends to `pageArea` `B` and consults its maximum occurrence property. This is set to 1, so it can't create a sibling for `pageArea`. Instead it ascends once more and finds the next `pageArea`, `E`. It adds a node to the Layout DOM corresponding to `E` and descends into `contentArea` `F`. It adds a node corresponding to `contentArea` `F` and begins pouring content into it. At this point the Layout DOM contains:

```
[root]
   [pageSet (A)]
      [pageArea (B)]
         [contentArea (C)]
            … content …
         [contentArea (D)]
            … content …
      [pageArea (E)]
         [contentArea (F)]
            … content …
```

When `F` is full, the layout processor ascends to `E` and finds that its maximum occurrence limit has not yet been exhausted, so it adds another instance of it to the Layout DOM as a sibling of the previous instance. Then it descends once again to `contentArea` `F`, adding another instance of it to the Layout DOM. At this point the Layout DOM contains:

```
[root]
   [pageSet (A) relation="orderedOccurrence"]
      [pageArea (B)]
         [contentArea (C)]
         [contentArea (D)]
      [pageArea (E[0])]
         [contentArea (F)]
      [pageArea (E[1])]
         [contentArea (F)]
```

When `F` fills up the layout processor once again ascends to `E`. This time the maximum occurrence limit has been exhausted, so it ascends once again. There are no more `pageArea` objects to descend into, so it considers adding another instance of `pageSet` `A`. This has a maximum occurrence limit of -1. A maximum occurrence property of -1 is interpreted by the layout processor as meaning no limit. Hence it may duplicate the `pageSet` without limit. It adds an instance of `pageSet` `A` to the Layout DOM and descends as before. At this point the Layout DOM contains:

```
[root]
   [pageSet (A[0]) relation="orderedOccurrence"]
      [pageArea (B)]
         [contentArea (C)]
            … content …
         [contentArea (D)]
            … content …
      [pageArea (E[0])]
         [contentArea (F)]
```

```
        … content …
    [pageArea (E[1])]
      [contentArea (F)]
        … content …
[pageSet (A[1])]
  [pageArea (B)]
    [contentArea (C)]
      … content …
```

Assuming that the last content is used up without filling this latest `contentArea` (which could be called `A[1].B.C`), the resulting document would consist of four display surfaces. (If rendered and printed single-sided at this point it would come out of the printer as four sheets of paper, with each sheet having printing on one side.)

In the example above only the amount of data limits the number of surfaces. However had the maximum occurrence limit for `pageSet A` been a positive number, the layout processor could have exhausted it. When this occurs the layout processor stops adding content, and it is recommended to issue a warning message. It does not traverse to another `pageSet`, even if there is one. The template syntax allows other `pageSet` objects to exist but they may not be used for this purpose.

The maximum occurrence limit on the `pageSet` is likely to be used as a safety-valve to prevent the accidental generation of huge print runs. However it may also be used to intentionally extract just the first portion of a document. For that reason, when the limit is reached, the layout processor should preserve the Layout DOM so that the content laid out to that point can be rendered.

The value of the maximum occurrence limit for a `pageSet` or `pageArea` must be either -1, which signifies no limit, or a positive (i.e. one or greater) decimal integer. If not supplied it defaults to -1.

Note that subforms may also have maximum occurrence values, but those are used only in the data binding (merge) process; they have no effect on the layout processor. See the "Basic Data Binding to Produce the XFA Form DOM" on page 155 for more information about minimum and maximum occurrence values for subforms.

➤ **Error Condition: Exhaustion of pageArea Occurrences**

If all available `pageSet` and `pageArea` objects have maximum occurrence properties that are not equal to -1, there is a limit to how many `pageArea` objects can be included in the layout. When the last `pageArea` within this limit has been laid out, the layout processor stops processing. If there is more content that has not yet been laid out, the additional content is discarded. However the layout processor has no way of knowing whether the situation arose deliberately or as a result of an accidental mismatch between the template and the user data. Hence the layout processor should issue a warning but retain the pages laid out up to that point in the Layout DOM for rendering.

Algorithm for Minimum Occurrence Limits

When the layout processor is following an ordered pagination strategy, minimum occurrence properties on `pageSet` and `pageArea` objects force it to incorporate one or more copies of the associated object into the Layout DOM when descending through the node, even if no content is put into it. The default minimum occurrence property is 0. When the minimum occurrence property is greater than 1 the layout processor creates the specified number of siblings and then descends into the leftmost (eldest) of the new sibling nodes. The other siblings are used later if an empty container is needed, rather than creating another sibling. For example, suppose a template contains the following declarations.

Example 8.9 Page set with minimum occurrence limits

```
<pageSet relation="orderedOccurrence" name="A">
   <occur min="2"/>
   <pageArea name="B">
      <occur min ="0"/>
      <contentArea name="C" … />
      <contentArea name="D" … />
   </pageArea>
   <pageArea name="E">
      <occur min="2"/>
      <contentArea name="F" … />
   </pageArea>
</pageSet>
```

Assume that there is no `breakBefore` on either the root subform or its first child subform. At startup the layout processor descends from the first `pageSet` of the root subform into its first `pageArea` child, and thence into its first `contentArea` child. However the minimum occurrence property of `pageSet A` forces the layout processor to include two instances of `A` into the Layout DOM. Furthermore the minimum occurrence limit of `pageArea E` forces the layout processor to include two instances of `E` under each instance of `pageSet A`. The result is:

```
[root]
   [pageSet (A[0]) relation="orderedOccurrence"]
      [pageArea (B)]
         [contentArea (C)]
      [pageArea (E[0])]
      [pageArea (E[1])]
   [pageSet (A[1])]
      [pageArea (E[0])]
      [pageArea (E[1])]
```

Hence, the document already includes five `pageArea` objects, even though it does not yet have any content. (If rendered and printed single-sided at this point it would come out of the printer as five blank sheets of paper.) As content is poured into pre-existing `contentArea` objects, and more `contentArea` objects are added, the `pageArea` objects are gradually consumed. No more `pageArea` or `pageSet` objects are added until the existing ones are used up. At that point the Layout DOM contains:

```
[root]
   [pageSet (A[0]) relation="orderedOccurrence"]
      [pageArea (B)]
         [contentArea (C)]
            … content …
         [contentArea (D)]
            … content …
      [pageArea (E[0])]
         [contentArea (F)]
            … content …
      [pageArea (E[1])]
         [contentArea (F)]
            … content …
   [pageSet (A[1]) relation="orderedOccurrence"]
      [pageArea (B)]
         [contentArea (C)]
            … content …
         [contentArea (D)]
            … content …
      [pageArea (E[0])]
         [contentArea (F)]
            … content …
      [pageArea (E[1])]
         [contentArea (F)]
            … content …
```

The layout processor is data-driven; it lays down `pageArea` objects in order to use the `contentArea` objects on them. However it is possible for a `pageArea` to contain boilerplate but no `contentArea`. The minimum occurrence limit makes it possible to force the layout processor to lay down an instance of such a page, despite its lack of a `contentArea`.

The value of a minimum occurrence limit for a `pageSet` or `pageArea` must be a non-negative (i.e. zero or larger) decimal integer.

Minimum and maximum occurrence limits may be combined. If the same `pageSet` or `pageArea` has both minimum and maximum occurrence limits the maximum must be either -1 or larger than the minimum.

Note that subforms may also have minimum occurrence values, but those are used only in the data binding (merge) process; they have no effect on the layout processor. See the "Basic Data Binding to Produce the XFA Form DOM" on page 155 for more information about minimum and maximum occurrence values for subforms.

Qualified pagination strategies

When the `relation` property of a `pageSet` is either `simplexPaginated` or `duplexPaginated` the page set employs a qualified pagination strategy. When the layout processor needs to transition to a new page it looks at the qualifications of each `pageArea` in the set to determine which `pageArea` to use. It uses the first `pageArea` (in document order) that has not already been exhausted and that has suitable qualifications.

There are three page qualifications supported in XFA. They are described below. The set of page qualifications and the way they are interpreted is very similar to a subset of the pagination properties used

in the World-Wide Web Consortium's Extensible Stylesheet Language [XSL-FO]. To make it easier for readers familiar with XSL-FO the following discussion explicitly shows the parallels.

All of these page qualifications are ignored if the `relation` property of the parent `pageSet` object is `orderedOccurrence`.

The pagePosition property

The `pagePosition` property qualifies the position of the page with respect to a contiguous sequence of pages *from the same page set*. The possible values are:

- `first` - The page must be the first in the set.
- `last` - The page must be the last in the set.
- `rest` - The page must not be the first or last in the set but can be in any other position.
- `only` - The page must be the only one in the set.
- `any` - No qualification is asserted. This is the default.

This property corresponds to the `page-position` property in XSL-FO.

A pagePosition of `first`, `last`, or `only` inherently limits the `pageArea` to being used only once within the page set. By contrast a `pagePosition` of `any` or `rest` implies that the `pageArea` can be used any number of times within the `pageSet`. Because of these implied occurrence limits `pageArea` objects that use qualified pagination ignore their `occur` properties.

Note: In order to support `only` and `last` the layout processor may have to redo the layout of the current page under some circumstances. For example, it cannot know that the page set will contain only one page until it encounters the break or end of processing that terminates the page set. Form authors using `only` and/or `last` should expect an increase in CPU overhead.

The oddOrEven property

The `oddOrEven` property qualifies the page with respect to whether the physical surface count is odd or even (hence, when printing in duplex, whether the page is on the front or back surface). The possible values are:

- `odd` - The physical surface count must be odd-numbered (front surface).
- `even` - The physical surface count must be even-numbered (back surface).
- `any` - No qualification is asserted. This is the default.

This property approximates the `odd-or-even` property in XSL-FO. However in XSL-FO the controlling variable is not the physical surface count but the folio number. The XSL-FO folio number can be set by the stylesheet wherease the XFA physical surface count cannot be altered by the application. The physical surface count always starts at zero and increments by one for each surface printed or displayed. Also XSL-FO allows a value `inherit` which is not supported by XFA.

Note: When the pagination strategy is `simplexPaginated` the page is always assumed to be the front surface of a sheet, hence always odd.

The blankOrNotBlank property

The `blankOrNotBlank` property qualifies the page with respect to the reason why it was included. The possible values are:

- `blank` - The page was included merely to satisfy a break-to-even or break-to-odd requirement, not to hold any content. The break requirement is asserted by a `subform` or `subformSet` object. The break

requirement could have been asserted using the `breakBefore` property, the `breakAfter` property, or the deprecated `break` property. However it may have been asserted, it specifies a `targetType` of either `pageEven` or `pageOdd`. This context corresponds to the assertion of a `force-page-count` property in XSL-FO.

- `notBlank` - The page was included either to hold content or to satisfy a minimum occurrence requirement.
- `any` - No qualification is asserted. This is the default.

Note: A page may be included to hold content yet the content may not include anything visible to the eye. This qualification is based upon the context, not the content of the page.

Page selection algorithm

A page set is entered at the start of processing as follows. The layout processor examines the root subform to see if it asserts `break` or `breakBefore`. If so it enters the specified `pageSet`. If not it examines the first subform child of the root subform and in the same way enters the specified `pageSet` if one is specified. If that also fails then it enters the first `pageSet` in document order.

If the selected `pageSet` asserts a qualified pagination strategy then the layout processor has to decide which `pageArea` to use. This also happens during layout whenever the layout processor overflows the current page. The algorithm is simply to searches the children of the current `pageSet` in document order for a `pageArea` object that matches the current layout state. The layout state is described by the following variables:

- page position - either first (before the first page is placed) or not-first (all remaining pages using the page set)
- odd or even - may be odd (front surface) or even (back surface)

When applying the `simplexPaginated` strategy the layout processor stays in the odd (front surface) state. When applying the `duplexPaginated` strategy the state is determined by the evenness or oddness of the physical surface count.

During processing the layout processor may break from an even page to a page that asserts `even`, or from an even page to a page that asserts `odd`. When this happens, if the strategy is `duplexPaginated`, the layout processor emits an extra page which is blank. This page must be qualified as `odd` (if breaking to an even page), `even` (if breaking to an odd page), or `any`. It must also be qualified with a `blankOrNotBlank` setting of `blank` or `any`. As with all qualified pages it must be a child of the current `pageSet`.

If during processing the layout processor cannot find a `pageArea` that is a child of the current `pageSet` and that matches its requirements a fatal error occurs.

At the end of processing, when it has exhausted the Form DOM, the layout processor performs termination processing for the page set. If there is a `pageArea` that is specified as `last` the layout processor backtracks and tries to redo the layout of the current page using that `pageArea`. However it does not do this is the current page does not match, that is if it has a different number of content areas or is itself qualified as `first` or `last`. If the `pageArea` object does match and the content fits onto it then the resulting page replaces the current page. However if the `pageArea` objects don't match or the content does not fit onto the designated `last` page then the layout processor keeps the current page and in addition emits a blank instance of the designated `last` page.

Similarly if a `pageArea` is specified as `only` and the current page is the only one in the page set, the layout processor tries to redo the layout of the current page using the designated `only` `pageArea`. If the content fits then the resulting page replaces the current page, otherwise the current page is retained.

The following table illustrates the layout state for a 3-page run using a simplexPaginated strategy.

surface	page position	odd or even
1	First	Odd
2	Not first	Odd
3	Not first	Odd

The following table illustrates the layout state for the same 3-page run using a duplexPaginated strategy:

surface	page position	odd or even
1	First	Odd
2	Not first	Even
3	Not first	Odd

Subforms and subform sets may specify breaks that direct the layout to a different page set. When a change of page set occurs the current page set is terminated before layout transitions to the new page set. There is no limit to the number of times a `pageSet` using qualified pagination may be used. Its `occur` property (like the `occur` properties of its `pageArea` children) is ignored.

Generally explicit breaks are permitted to go to any page or page set, however there is one restriction. If a `pageArea` has its `blankOrNotBlank` property set to `blank`, it is forbidden to break to that `pageArea`.

Combining multiple pagination strategies

Often it is necessary to print the same form sometimes on simplex printers and other times on duplex printers. To support this XFA allows the inclusion of multiple `pageSet` objects as children of the root subform. The `relevant` property is used to signify which `pageSet` is for duplex printers and which is for simplex printers.

XFA 2.5 augments the existing `print` predefined system view with the new ones `simplex` and `duplex`. The `simplex` and `duplex` views are mutually exclusive. In addition `print` must be asserted whenever `simplex` or `duplex` is asserted.

The `simplex` or `duplex` view can be asserted in the Configuration DOM by including it among the views listed in the `relevant` property. This could be used to force a form to be printed simplex even if the printer is capable of duplex printing. Alternatively it could be used to print all the surfaces of a duplex form even when the printer is simplex, which may be useful in photocopying.

Alternatively `simplex` or `duplex` can be asserted in an implementation-defined manner. Ideally the application knows what printer it is using, can find out from the operating system whether that printer is simplex or duplex, and sets the view accordingly.

Odered page sets can be combined by nesting them, using occurrence limits to force the return from a child page set to its parent. Qualified page sets can also be nested with each other, and it is legal to combine simplex with duplex in any order. However it is forbidden for a qualified page set to be nested inside an ordered page set or vice-versa.

Adhesion

Generally when using a flowing layout strategy the layout processor puts as much content as possible into each `contentArea` before moving on to the next `contentArea`, splitting layout objects where possible to pack them more efficiently into the `contentArea`. However sometimes the densest packing is not desired. For example, it may be desired to keep sequential layout objects together in the same `contentArea`, similar to widow and orphan control in word processors. The `keep` property of a subform has sub-properties which control exceptions to the default packing.

Adhesion of the current subform to an adjacent subform is controlled by the `next` and `previous` sub-properties of the `keep` property. These sub-properties accept three values.

- `none` means the subform does not adhere to the adjacent subform. This is the default.
- `contentArea` means the adjacent parts of the two subforms must be placed in the same content region.
- `pageArea` means the adjacent parts of the two subforms must be placed on the same page.

When the layout processor encounters a keep-next constraint for the current subform or a keep-previous constraint for the next subform, it holds off laying down the current subform until it reaches a `contentArea` big enough to hold both the bottom part of the current subform and the top part of the next subform. The next subform may have an adhesion constraint that similarly binds it to the next subform, and so on. If consecutive adhering subforms are not splittable then the layout processor holds off laying down all of them until they can be laid down together. The unused content region is left blank.

The default value for `next` and `previous` is always `none`, regardless of context.

Note that there is overlapping functionality. Two adjacent subforms adhere if the first one declares that it adheres to the next or if the second one declares that it adheres to the previous. It is also permissible for them both to declare that they adhere to each other. In all three cases the effect is the same.

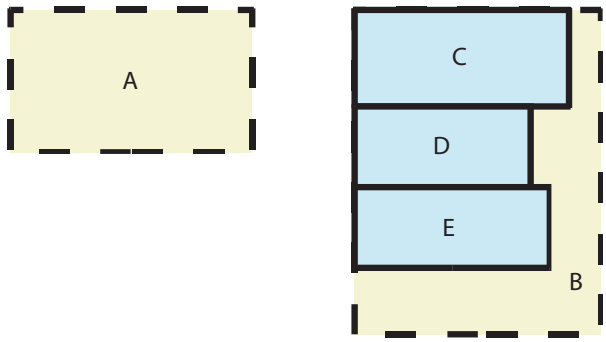
For example, the following template declares two subforms that each adhere to the next subform. The result is that three adjacent subforms adhere together.

Example 8.10 Template using adhesion

```
<template …>
   <subform name="root" layout="tb" …>
      <pageSet …>
         <pageArea …>
            <contentArea name="A" … />
            <contentArea name="B" … />
         </pageArea>
      </pageSet>
      <subform name="C" …>
         <keep next="contentArea" intact="contentArea" />
            …
      </subform>
      <subform name="D" …>
         <keep next="contentArea" intact="contentArea" />
            …
      </subform>
      <subform name="E" …>
         <keep intact="contentArea" />
            …
      </subform>
```

```
   </subform>
</template>
```

In this case all three of the subforms have been declared unsplittable using the `intact` property. The result is as shown at right. Because all three adhering subforms can not fit in the remaining region of `contentArea A`, they are placed together in `contentArea B`.



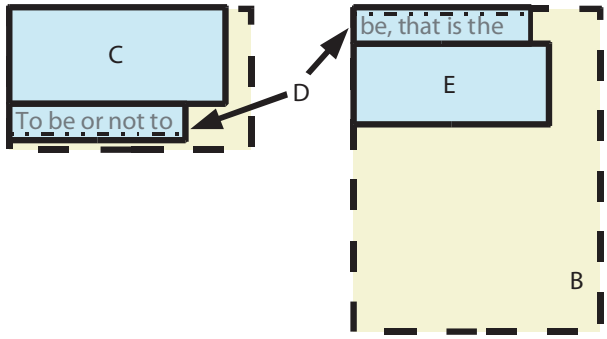
Adhesion of unsplittable subforms

The result would have been different if the subforms had been splittable. When an adhering subform is splittable only the adhering edge and the first fragment of content (not including the border) adhere to the adjacent subform. However if the smallest permissible fragment does not fit in the available space then the layout processor holds off laying down both subforms. Consider what happens if the previous example is modified so that subform `D` is splittable.

Example 8.11 Previous example modifed with a splittable subform

```
<template …>
   <subform name="root" layout="tb" …>
      <pageSet …>
         <pageArea …>
            <contentArea name="A" … />
            <contentArea name="B" … />
         </pageArea>
      </pageSet>
      <subform name="C" …>
         <keep next="contentArea" intact="contentArea" />
         …
      </subform>
      <subform name="D" …>
         <keep next="contentArea" />
         <draw …>
            <value …>
               <text> … </text>
            </value>
         </draw>
      </subform>
      <subform name="E" …>
         <keep intact="contentArea" />
         …
      </subform>
   </subform>
</template>
```

In this case subform C and the top part of D fit in contentArea A, while the remainder of D and subform E are placed in contentArea B, as shown at right. The adhesion requirements are still satisfied because C adheres to a piece of D and a piece of D adheres to E.



Adhesion of splittable subforms

Note: Adhesion is restricted to adjacent subforms that are siblings in the Form DOM. If they do not share the same parent they do not adhere. The reason for this is that not being siblings in the Form DOM implies that they are not logically grouped. This rule is particularly useful in dynamic forms, as discussed in "Adhesion in Dynamic Forms" on page 310.

For example, in the following template subform D does not adhere either to subform C or subform X because they are not siblings with D.

Example 8.12 Template in which adhesion has no effect

```
<template …>
   <subform name="root" layout="tb" …>
      <pageSet …>
         …
      </pageSet>
      <subform name="C" … />
      <subform name="X" …>
         <subform name="D" …>
            <keep prev="contentArea" />
            …
         </subform>
      </subform>
   </subform>
</template>
```

Adhesion is modified by the presence of a subform set. Subform sets have not been introduced yet. They are used with dynamic forms. The effect of a subform set upon adhesion is discussed "Adhesion in Dynamic Forms" on page 310.

Leaders and Trailers

A subform or subform set may be associated with leaders and/or trailers that are placed before and after objects in a flowing layout. Leaders and trailers must be subforms, however although a leader or trailer is a single subform it may have an arbitrary number of child subforms. Leader and trailer subforms and all their children must use positioned layout.

Break Leaders and Trailers

A subform or subform set may specify that layout is to transition to a new content region before, after, or both before and after the object is placed. This is described in "Break Conditions" on page 232. In addition the object may nominate a leader and/or trailer in association with the break condition.

The words *leader* and *trailer* have slightly different meanings in the context of a break condition than they do in other contexts. The leader and trailer do not surround the content of the object. Rather they surround the break itself. For example, suppose a before break is specified and it includes a leader and a trailer as follows.

Example 8.13 Template using a before break leader and trailer

```
<template>
   <subform name="W">
      <pageSet …>
         <pageArea …>
            <contentArea name="A" ID="A_ID" … />
            <contentArea name="B" ID="B_ID" … />
            <contentArea name="C" ID="C_ID" … />
         </pageArea>
      </pageSet>
      <subform name="D" layout="tb">
         <breakBefore
            targetType="contentArea"
            target="#B_ID"
            leader="#Leader_ID"
            trailer="#Trailer_ID"/>
      …
      </subform>
   </subform>
   <proto …>
      <subform name="Leader" ID="Leader_ID">
         <draw …>
            <text …> … </text>
         </draw>
      </subform>
      <subform name="Trailer" ID="Trailer_ID">
         <draw …>
            <text …> … </text>
         </draw>
      </subform>
   </proto>
</template>
```

When it is ready to start processing subform D, the layout processor carries out the following steps:

1. Places the leader into the current layout region, which in this case is content area A.

2. Moves, if necessary, to a new layout region to satisfy the target specification and target type. In this case it moves to content area B.

3. Places the trailer into the new layout region, content area B.

4. Begins placing the content of the object into the new layout region, content area B.

For comparison, suppose an after break is specified and it includes a leader and a trailer as follows.

Example 8.14 Template using an after break leader and trailer

```
<template>
   <subform name="W">
      <pageSet …>
         <pageArea …>
            <contentArea name="A" ID="A_ID" … />
            <contentArea name="B" ID="B_ID" … />
            <contentArea name="C" ID="C_ID" … />
         </pageArea>
      </pageSet>
      <subform name="D" layout="tb">
         <breakBefore targetType="contentArea" target="#B_ID"/>
         <breakAfter
            targetType="contentArea"
            target="#C_ID"
            leader="#Leader_ID"
            trailer="#Trailer_ID"/>
      …
      </subform>
   </subform>
   <proto …>
      <subform name="Leader" ID="Leader_ID">
         <draw …>
            <text …> … </text>
         </draw>
      </subform>
      <subform name="Trailer" ID="Trailer_ID">
         <draw …>
            <text …> … </text>
         </draw>
      </subform>
   </proto>
</template>
```

When it is about to finish processing subform D, the layout processor carries out the following steps:

1. Finishes placing the content of the object into the current layout region. Because of the before break this is content area B.

2. Places the leader into the current layout region, content area B.

3. Moves, if necessary, to a new layout region to satisy the target specification and target type. In this case it moves to content area C.

4. Places the trailer into the new layout region, content area C.

5. Procedes with the next object.

Note that the trailer is placed even if there is no subsequent object to lay down. In the example this is the case, because D is the last displayable object inside the root subform W.

Bookend Leaders and Trailers

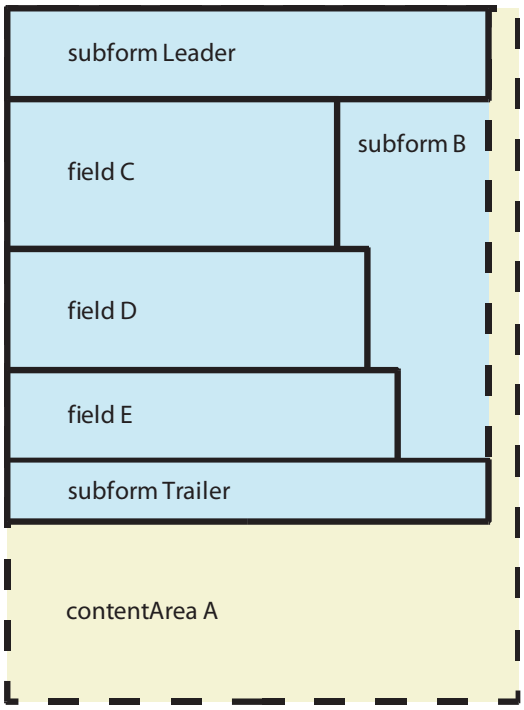
If a subform has a bookend leader specified, the layout processor inserts the leader into the Layout DOM as a child of the subform ahead of any other content. A bookend trailer is similar except it is placed after all

other content of the subform. Bookend leaders and trailers are controlled by the `bookend` property of the flowing subform. For example, a template includes the following declarations.

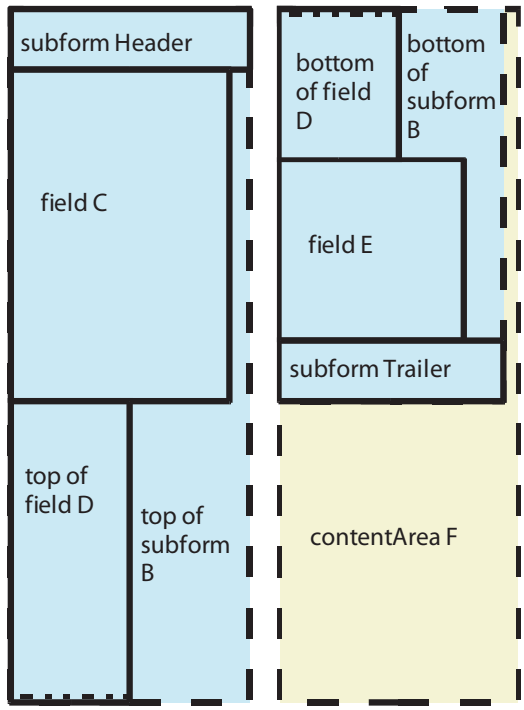
Example 8.15 Template using bookend leaders and trailers

```
<template>
   <subform name="W">
      <pageSet …>
         <pageArea …>
            <contentArea name="A" ID="A_ID" … />
         </pageArea>
      </pageSet>
      <subform name="B" layout="tb">
         <breakBefore
            targetType="contentArea"
            target="#A_ID"/>
         <bookend
            leader="#Leader_ID"
            trailer="#Trailer_ID"/>
         <field name="C" …> … </field>
         <field name="D" …> … </field>
         <field name="E" …> … </field>
      </subform>
   </subform>
   <proto …>
      <subform name="Leader" ID="Leader_ID">
         <draw …>
            <text …> … </text>
         </draw>
      </subform>
      <subform name="Trailer" ID="Trailer_ID">
         <draw …>
            <text …> … </text>
         </draw>
      </subform>
   </proto>
</template>
```

When flowing content into subform `B`, the layout processor starts by placing subform `Leader` at the top, then fields `C`, `D`, and `E` in that order, then subform `Trailer` at the end. The result is shown below at left.



Bookend subforms



Effect of bookend subforms when flowing across contentArea boundaries

A subform with a bookend leader and/or trailer may be split across `contentArea` boundaries. As shown above at right, fields `C`, `D`, and `E`, plus subforms `Leader` and `Trailer`, taken together, are too tall to fit in `contentArea A` and overflow into `contentArea F`. The layout processor places the bookend header as the first layout object inside `contentArea A` and the bookend trailer as the last layout object inside `contentArea F`.

The root subform may specify a bookend leader and/or trailer. These are incorporated at the beginning and/or end of the entire document.

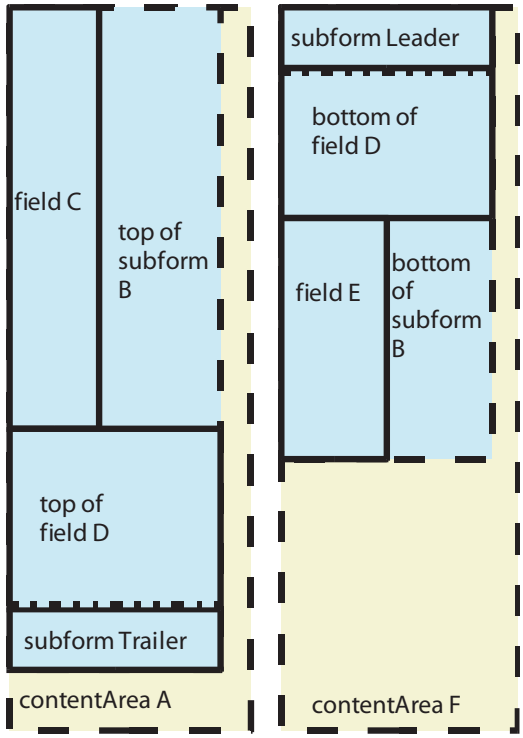
Overflow Leaders and Trailers

An overflow trailer is a subform that is placed as the last content of the top fragment of the subform, if the subform overflows from one `contentArea` to another. Similarly an overflow header is a subform that is placed as the first content in the bottom fragment of the subform. Overflow leaders and trailers are controlled by the `overflow` property of the flowing subform. For example, a template includes the following declarations.

Example 8.16 Template using overflow leader and trailer

```
<template>
   <subform name="W">
      <pageSet …>
         <pageArea …>
            <contentArea name="A" ID="A_ID" … />
            <contentArea name="F" ID="B_ID" … />
         </pageArea>
      </pageSet>
      <subform name="B" layout="tb" …>
         <overflow
            leader="#Leader_ID"
            trailer="#Trailer_ID"/>
         <field name="C" …> … </field>
         <field name="D" …> … </field>
         <field name="E" …> … </field>
      </subform>
   </subform>
   <proto …>
      <subform name="Leader" ID="Leader_ID">
         <draw …>
            <text …> … </text>
         </draw>
      </subform>
      <subform name="Trailer" ID="Trailer_ID">
         <draw …>
            <text …> … </text>
         </draw>
      </subform>
   </proto>
</subform>
```

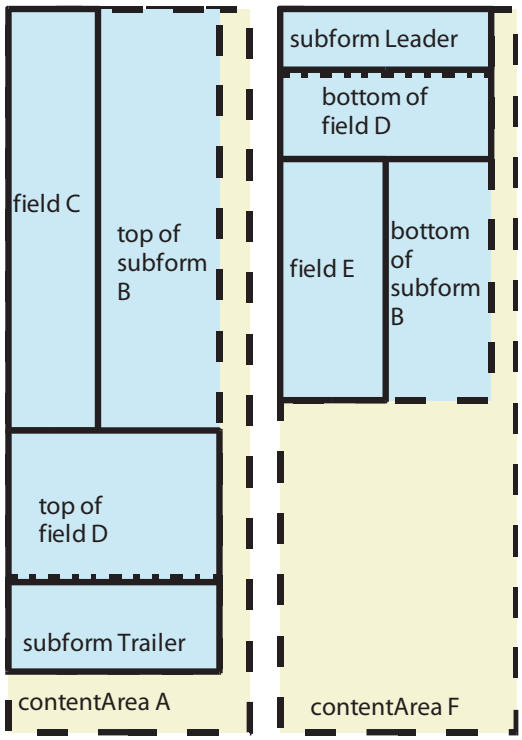
Assume that the total height of fields C, D, and E is greater than the height of contentArea A. The layout processor places subform B into contentArea A, and starts placing the fields into subform B. While placing the fields into B it reserves space for Trailer. Field D overflows the available space. The layout processor splits field D, then it places the top of D and Trailer into subform B. It splits subform B at the bottom of Trailer, completing the first fragment of B. Then it begins to place the second fragment of subform B into contentArea F. Into this it places the bottom of D, all of E, and Trailer. The result is shown at right.



Overflow leader and trailer subforms

In the example D could not split in the ideal location (exactly at the bottom of contentArea A), so its top fragment is a little shorter than it could have been. Subform Trailer is placed immediately after the top fragment of D, leaving a little space between Trailer and the bottom of contentArea A.

Note: The layout processor must reserve space in advance for the overflow trailer. This reservation of space sometimes forces an overflow to happen which would not have happened otherwise. In the figure below, which is like the previous example but with subform Trailer taller and field D shorter, D would have fit into the available space in contentArea A if some of that space had not been reserved for the overflow trailer.



Reserving space causes non-optimum packing

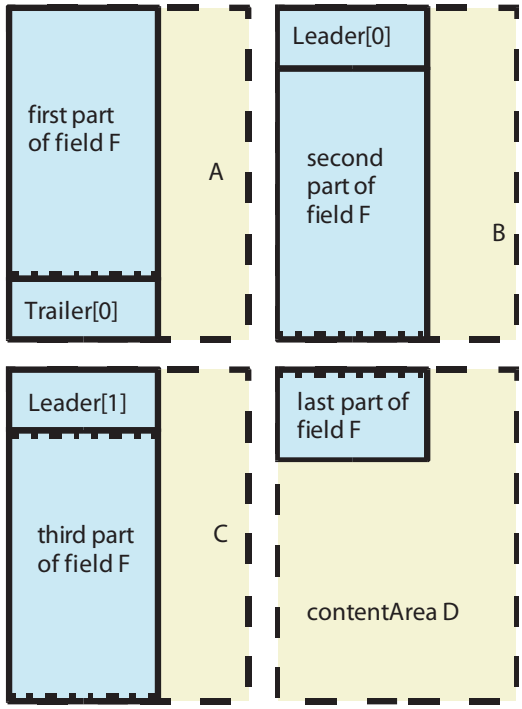
When a field overflows the overflow leader and trailer is supplied by the field's containing subform, because a field does not have an `overflow` property. However when a subform overflows it may supply its own overflow leader and trailer. If a subform overflows and it specifies its own overflow leader then that overflow leader is used, otherwise it uses the inherited one. The overflow trailer behaves the same way.

The layout processor respects maximum occurrence properties of leader and trailer subforms. Within a particular subform (in the example above subform `B`), the layout processor stops laying down leader or trailer subforms when the leader or trailer subform's maximum occurrence is exhausted. For example, suppose that the template contained the following declarations.

Example 8.17 Template with maximum occurrence properties on leader and trailer

```
<template>
   <subform name="W">
      <pageSet …>
         <pageArea …>
            <contentArea name="A" … />
            <contentArea name="B" … />
            <contentArea name="C" … />
            <contentArea name="D" … />
         </pageArea>
      </pageSet>
      <subform name="E" layout="tb" …>
         <overflow leader="#Leader_ID" trailer="#Trailer_ID"/>
         <field name="F"…> … </field>
      </subform>
   </subform>
   <proto …>
      <subform name="Leader" ID="Leader_ID">
         <occur max="2"/>
         <draw …>
            <text …> … </text>
         </draw>
      </subform>
      <subform name="Trailer" ID="Trailer_ID">
         <occur max="1"/>
         <draw …>
            <text …> … </text>
         </draw>
      </subform>
   </proto>
</template>
```

Assume that field F is very tall compared to the contentArea objects. Subform E overflows from A and B, then from B and C, and finally from C to D. Subform Trailer's occurrences are used up after the first overflow, so it only appears at the bottom of contentArea A below the first fragment of E. Leader's occurrences are used up after the second overflow, so it appears at the top of contentArea B (Leader[0]) and at the top of contentArea C (Leader[1]). The result is shown at right.



Effect of occurrence limits on leader and trailer subforms

Often leader and trailer subforms are placed in the proto section of the template (rather than under the root subform) to prevent them from taking part in the merge process. Alternatively leader and trailer subforms may be made nameless or given a scope of none, either of which also prevent them from participating in the merge process. However if none of these things are done then the leader or trailer subform may also appear in the Form DOM bound to a node in the Data DOM. To accommodate this the layout processor maintains its own occurrence counts for leaders and trailers, separate from occurrence counts used by the merge process. On the other hand if the same subform is used both as a leader and a trailer, its occurrence limit applies to the total of its appearances as leader and as trailer.

Overflow Leader/Trailer Lists

Both overflowLeader and overflowTrailer properties may have values which are space-separated lists of target specifications. Each target specification that is an XML ID must start with the '#' character. All other target specifications are interpreted as SOM expressions. The separator must be a single SPACE (U0020) character.

Caution: When the target specification is a SOM expression the expression must not include a SPACE (U0020) character.

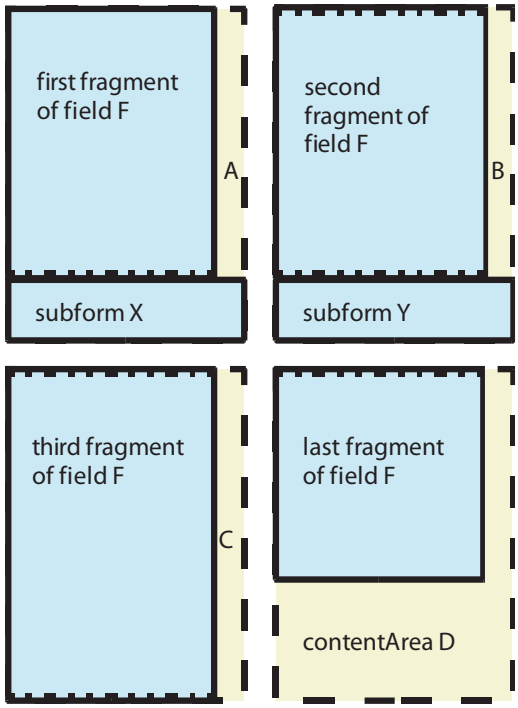
(Of course a target specification which is an XML ID must not contain a space either but this is already forbidden by the XML specification [XML1.0].)

Each target specification is (re)used as required until its maximum occurrence limit is reached, after which the layout processor goes on to the next target specification in the list. A target specification may appear in both lists; each use from either list counts towards its occurrence limit. It is pointless to put a target specification more than once in the same list because for the second and subsequent appearances its maximum occurrence limit will already have been exhausted. For example, a template includes the following declarations.

Example 8.18 Template using an overflow trailer list

```
<template>
   <subform name="W">
      <pageSet …>
         <pageArea …>
            <contentArea name="A" … />
            <contentArea name="B" … />
            <contentArea name="C" … />
            <contentArea name="D" … />
         </pageArea>
      </pageSet>
      <subform name="E" layout="tb" …>
         <overflow trailer="#X_ID #Y_ID #X_ID"/>
         <field name="F"…> … </field>
      </subform>
   </subform>
   <proto …>
      <subform name="X" ID="X_ID">
         <occur max="1"/>
         <draw …>
            <text …> … </text>
         </draw>
      </subform>
      <subform name="Y" ID="Y_ID">
         <occur max="1"/>
         <draw …>
            <text …> … </text>
         </draw>
      </subform>
   </proto>
</template>
```

The figure at right shows the result of laying out this form. Assume that the merge results in field F holding a large amount of text. Subform X is used as an overflow trailer once, exhausting its maximum occurrence limit. The layout processor moves on to the next object in the list, which is subform Y again. After subform Y has been used the layout processor goes on to the next overflow trailer subform, which is subform X again. However X's limit is still exhausted, so the layout processor passes over it. The end of the list has been reached so the layout processor stops laying down overflow trailers.



Effect of a trailer subform list

➤ **Warning: Invalid Leader/Trailer Target**

If a leader or trailer target is not valid (for example if it does not exist or is not an appropriate object), the layout processor issues a warning message and continues processing without laying down the leader or trailer.

Inheritance of Overflow Leaders and Trailers

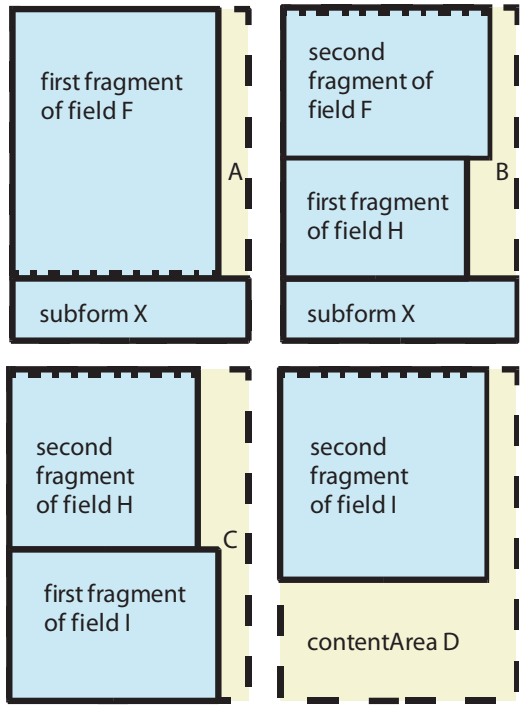
When a subform does not specify an overflow leader or trailer, it inherits the overflow leader or trailer specified by its containing subform. Along with the leader or trailer subform (or list of subforms) it inherits the count(s) of maximum occurrences. In other words, the inherited leaders and trailers that are laid down by the child subform count towards the maximum occurrence limits for the parent subform. On the other hand, when a subform asserts a leader or trailer of its own, it acquires its own set of occurrence counts. Even if the same leader or trailer subform is used by some other subform, the occurrence count(s) start at zero for each asserting subform.

For example, a template includes the following declarations.

Example 8.19 Template showing inheritance of overflow trailer

```
<template>
   <subform "W" …>
      <pageSet …>
         <pageArea …>
            <contentArea name="A" … />
            <contentArea name="B" … />
            <contentArea name="C" … />
            <contentArea name="D" … />
         </pageArea>
      </pageSet>
      <subform name="E" layout="tb" …>
         <breakBefore
            targetType="contentArea"
            target="A"/>
         <overflow trailer="#X_ID"/>
         <field name="F" … > … </field>
         <subform name="G" layout="tb" …>
            <overflow trailer="#X_ID"/>
            <field name="H" … > … </field>
         </subform>
         <field name="I" …> … </field>
      </subform>
   </subform>
   <proto>
      <subform name="X" ID="X_ID" …>
         <occur max="1"/>
         …
      </subform>
   </proto>
</template>
```

Assuming the fields F and H each contain moderate amounts of text, the layout processor puts the first fragment of field F into contentArea A, laying down one instance of subform X as an overflow trailer at the bottom. This exhausts the maximum occurrence limit for subform X. The layout processor finished processing field F by placing the second fragment of it into contentArea B. At this point it encounters subform G. At this point, because G declares an overflow trailer for itself, the layout processor starts a separate count of instances of subform X. It is able to place an instance of subform X at the bottom of contentArea B as an overflow trailer because the new count of instances has not yet reached the limit. Upon finishing with subform G the layout processor returns to subform E in order to process field I. Subform G's occurrence count for subform X is still set to one, so it does not lay down an overflow trailer when field I overflows contentArea C. The result is shown at right.



Effect of inherited occurrence counts

Inheritance need not be direct. Objects other than subforms are transparent to inheritance of overflow leaders and trailers. For example, a subform A contains an area B which in turn contains a subform C. If subform A asserts an overflow leader but subform C does not, subform C inherits the overflow leader from A. In addition, inheritance can chain through any number of intermediate subforms that do not assert the leader or trailer. However the chain of inheritance can be stopped at a particular subform by asserting an overflow leader or trailer with the name "" (the empty string).

A subform may also inherit an overflow leader or trailer once it has exhausted the occurrence limit(s) for its own overflow leader or trailer subform(s). When this happens the layout processor resumes spending inherited leader or trailer subform(s). When these inherited occurrences are exhausted the layout processor moves up the chain of inheritance and resumes spending occurrences at the next higher level, and so on. Only when all inheritable overflow leaders or trailers have been exhausted does it stop inserting overflow leaders or trailers.

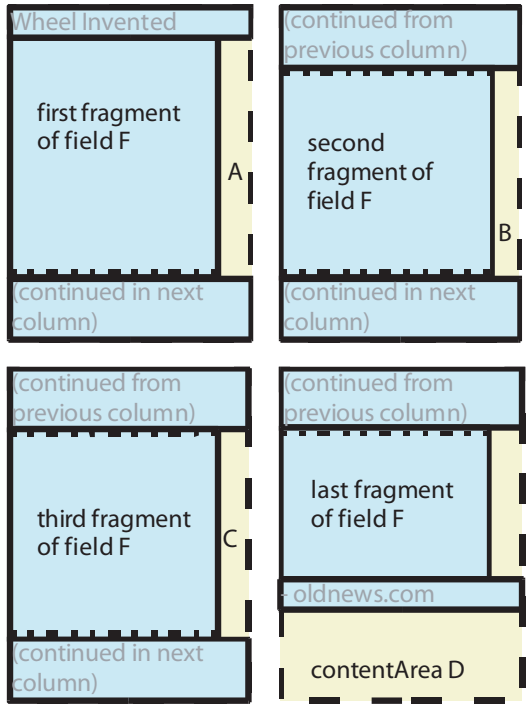
Combined Leaders and Trailers

Leaders and trailers of all types may be combined in the same context. For example, a template includes the following declarations.

Example 8.20 Template combining different types of leaders and trailers

```
<template>
   <subform name="W">
      <pageSet …>
         <pageArea …>
            <contentArea name="A" … />
            <contentArea name="B" … />
            <contentArea name="C" … />
            <contentArea name="D" … />
         </pageArea>
      </pageSet>
      <subform name="E" layout="tb" …>
         <bookend
            leader="#Title_ID"
            trailer="#Source_ID"/>
         <overflow
            leader="#X_ID"
            trailer="#Y_ID"/>
         <field name="F" …> … </field>
      </subform>
   </subform>
   <proto>
      <subform name="Title" ID="Title_ID">
         <draw …>
            <text>Wheel Invented</text>
         </draw>
      </subform>
      <subform name="Source" ID="Source_ID">
         <draw …>
            <text>oldnews.com</text>
         </draw>
      </subform>
      <subform name="X" ID="X_ID">
         <draw …>
            <text …>(continued from previous column)</text>
         </draw>
      </subform>
      <subform name="Y" ID="Y_ID">
         <draw …>
            <text …>(continued in next column)</text>
         </draw>
      </subform>
   </proto>
</template>
```

After merging field F holds a large amount of text. The result is shown at right. The bookend header named Title, containing "Wheel Invented", has been laid down before the first fragment of F. The bookend trailer named Source, containing "oldnews.com", has been laid down after the last fragment of F. In addition the overflow trailer Y "(continued in next column)" and the overflow leader X "(continued from previous column)" have been inserted wherever F has flowed across containers.



Combined bookend and overflow leaders and trailers

Tables

The layout process can automatically arrange layout objects into aligned rows and columns. This is accomplished by marking subforms in the template as table or row subforms using the layout property. A table subform represents an entire table and contains everything in the table. A row subform represents one row of a table and contains everything in the row. A row subform can only exist inside a table subform, although it may not be a direct child of a table subform (for example it may be a child of a subform set which is itself a child of a table subform).

The table subform may optionally supply a list of column widths. If the list of column widths is supplied, each width must be either a measurement or -1. A column width of -1 tells the layout processor to fit the column to the natural width of the widest object in the column. If no list of column widths is supplied, all column widths default to -1. Similarly the widths for any columns that are not present in the list (that is, beyond the length of the list) default to -1.

The following example shows the structure of a table in the template.

Example 8.21 Subforms using table layout

```
<subform name="T" layout="table" columnWidths="1in -1 25mm">
   <subform name="P" layout="row">
      <field name="A" …/>
      <draw name="B" …/>
      <subform name="C" …/>
      <subform name="D" …/>
   </subform>
   <subform name="Q" layout="row">
      <subform name="J" …/>
      <field name="K" …/>
      <draw name="L" …/>
      <subform name="M" …/>
   </subform>
</subform>
```

In the above example the first column is set to one inch wide, the second is unspecified, the third column is set to 25 millimeters wide, and the fourth is unspecified. As usual in layout when a fixed size is allotted for an object, the visible representation of the object may extend beyond the allotted region.

The layout processor regards each layout object inside a row subform as a cell in the table. First it lays out the cells in each row in order from left to right with their natural sizes. Then it adjusts the cell sizes to align the table. For each row it expands the cells vertically to the height of the tallest cell in the row. This results in each row being vertically aligned. Next it lays out the rows sequentially from top to bottom. Then the layout processor aligns the columns. It expands the cells in each column horizontally to the designated width, or if the width is not specified to the width of the widest cell in the column. If a row does not have as many cells as other rows then it leaves an empty region on the right of that row.

The following figure shows the above example before and after table alignment.

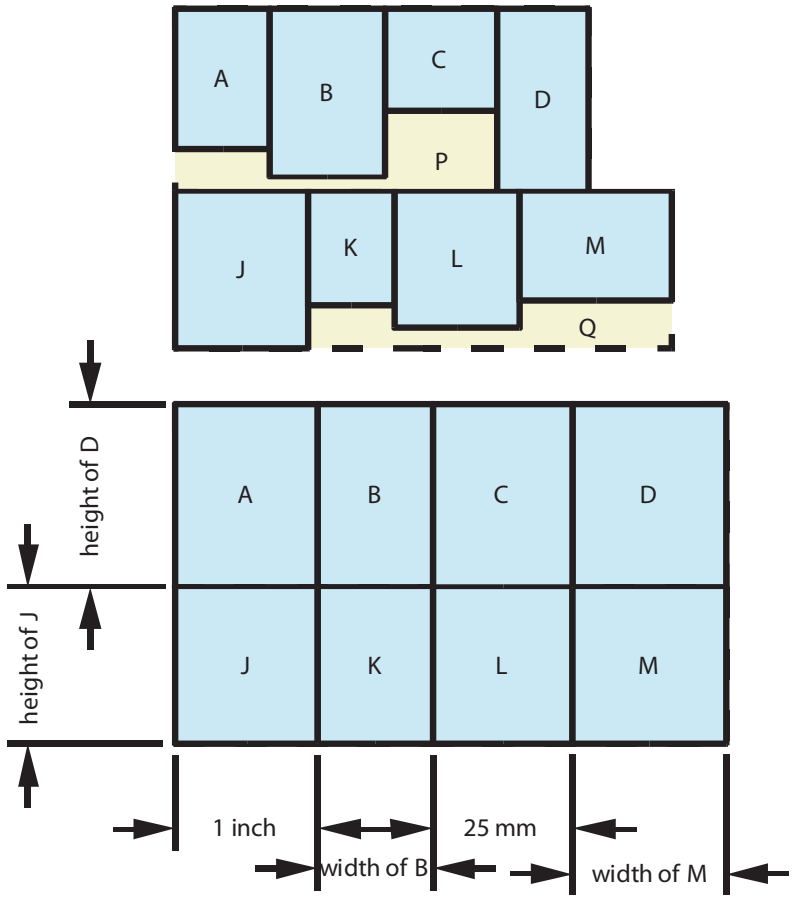


Table before and after alignment

A column is normally the set of corresponding cells from different rows, in row order. For example, the second column consists normally of the second cell from each row. However it is possible to make an individual cell span more than one column using the colSpan attribute of the draw, field, subform, and area elements. If colSpan is set to a positive integer the cell spans that many columns, but if colSpan is -1 the cell spans *all* the remaining columns. If a row has a cell with colspan set to -1 and additional cells after that, the extra cells are not displayed. If colSpan is not supplied the value defaults to 1. Note that colSpan must not be set to zero.

Consider the following example.

Example 8.22 Subforms using the colSpan property in a table layout

```
<subform name="T" layout="table" columnWidths="0.5in 0.5in 0.5in 25mm 0.6in ">
   <subform name="P" layout="row">
      <field name="A" …/>
      <draw name="B" colSpan="2"…/>
   <subform name="C" …/>
   <subform name="D" …/>
</subform>
<subform name="Q" layout="row">
   <subform name="J" colSpan="2" …/>
      <field name="K" …/>
      <draw name="L" colspan="-1" …/>
      <subform name="M" …/>
   </subform>
</subform>
```

The figure at right shows this example before and after table alignment. The first column contains A and the left side of J. The second column contains the left side of B and the right side of J. The third column contains the right side of B and all of K. The fourth column contains all of C and the left side of L. The fifth column contains the all of C and the right side of L. M does not appear because it is preceded by a cell (L) with a colSpan of -1.

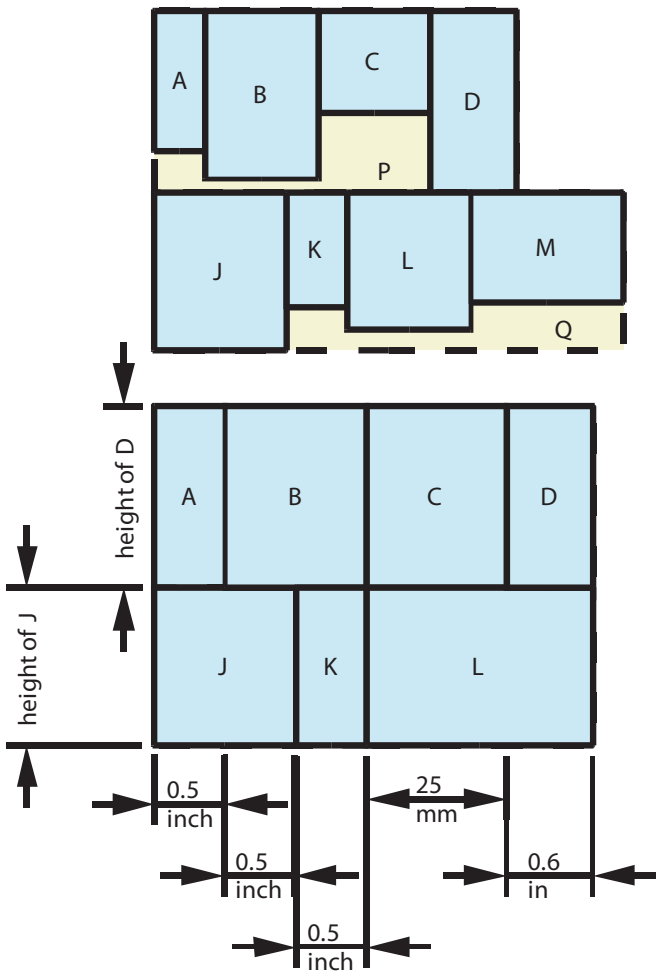


Table with cells spanning multiple columns, before and after alignment

In this example all the columns have constrained widths. It is possible for a table to contain cells spanning columns with unconstrained widths. As long as at least one cell in each unconstrained column does not span multiple columns the table is well-defined. However if any given unconstrained column contains only cells that span multiple columns the table is not well-defined and the resulting layout is up to the implementation. Most tables have one title cell per column so this situation does not usually arise.

Note that, in contrast to cells spanning columns, XFA does not provide support for cells spanning more than one row.

The examples above show uniquely-named cells and rows but neither cells nor rows have to be uniquely named. It is also normal and expected for cells and rows to be subforms or subform sets that have

multiple and/or variable (dynamic) occurrences. The layout algorithm as described here is only affected by the presence of objects in the Form DOM, not by their names or how they got there.

The examples above do not show margins or borders for the table or row subforms but it is normal and expected for them to be used. In addition the cell objects may have their own margins and/or borders.

Typically all the direct children of a table subform are row subforms. However a table subform may have direct children consisting of any mixture of row subforms and ordinary subforms or other layout objects (although row subforms must not appear as descendents at a deeper level). The non-row child is laid out in the same place where a row would appear, but it and its contents are not adjusted for alignment in height or width.

A table subform may descend from a table subform, causing tables to be nested. Tables may nest to any level.

Tables can be freely combined with leaders and/or trailers. A table subform may employ a row subform as a leader or trailer, but it may also employ an ordinary subform.

9 Dynamic Forms

This chapter explains how XFA processing applications support binding and layout in dynamic forms. A dynamic forms differ from static forms in their ability to dynamically add containers and rearrange layout depending on the data being entered.

"Basic Data Binding to Produce the XFA Form DOM" on page 155 describes data binding for static forms and "Basic Layout" on page 45 describes layout for static forms.

Static Forms Versus Dynamic Forms

In a static form the template is laid out exactly as the form will be presented. When the template is merged with data, some fields are filled in. Any fields left unfilled are present in the form but empty (or optionally given default data). These types of forms are uncomplicated and easy to design, though not as capable as dynamic forms. XFA supports both static and dynamic forms.

In a dynamic form the number of occurrences of form components is determined by the data. For example, if the data contains enough entries to fill a particular subform 7 times, then the Form DOM incorporates 7 copies of the subform. Depending on the template, subforms may be omitted entirely or rearranged, or one subform out of a set selected by the data. Dynamic forms are more difficult to design than static forms but they do not have to be redesigned as often when the data changes. In addition dynamic forms can provide an enhanced visual presentation to the user because unused portions of the form are omitted rather than simply left blank. When printed, dynamic forms save paper and toner. When displayed on glass, dynamic forms eliminate unnecessary scrolling.

Whether a form is static or dynamic is determined when it is designed. In addition a form may be partly dynamic and partly static. The mechanism is controllable at the level of individual subforms. When a subform has no `occur` sub-element, or its minimum, maximum, and initial occurrence properties are all set to 1, it is static. When the values are fixed and equal but not 1 (for example if they are all 7), the subform is also static. In such a subform the occurrence values merely take the place of replicating the subform so many times. But if the values are unequal or if the maximum occurrence is unlimited (indicated by a value of -1), the subform is dynamic. For example, if the maximum occurrence is unlimited the data binding process will keep adding more copies of the subform to the Form DOM until it has used up all the matching data from the Data DOM.

In addition, a form is dynamic if it includes subform sets. Subform sets allow for subforms to be grouped into sets with certain logical relationships. For example, one out of a set of subforms can be incorporated depending upon what element is present in the data. Any given subform set can express an ordered set, an unordered set, or a choice set, corresponding to the set relationships defined in [XML Schema].

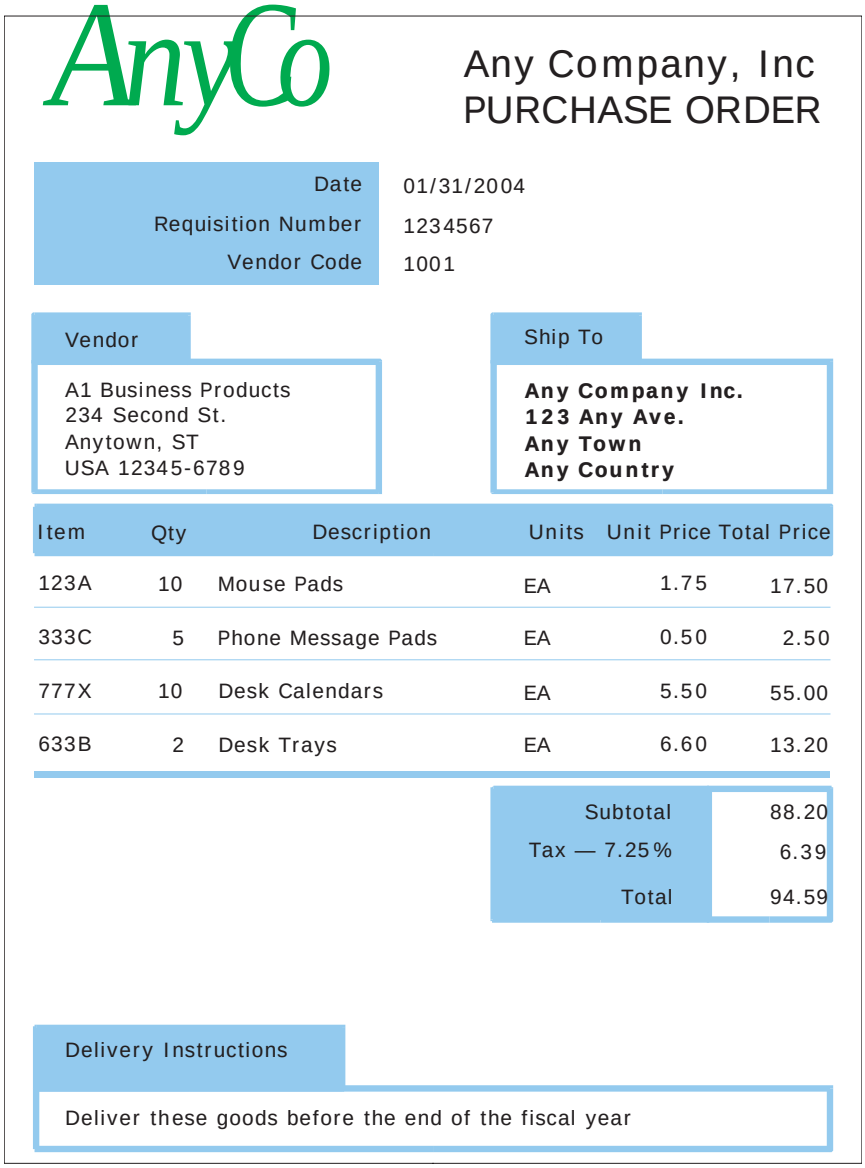
The same data that is used with a static form can also be used with a dynamic form.

Data Binding for Dynamic Forms

As explained in "Static Forms Versus Dynamic Forms" on page 286, dynamic forms are data-driven To the end user this means that unused portions of the form are omitted. This makes the filled form less cluttered and more convenient to view in a display of finite size. The number of occurrences can also be limited to a range bounded by the `max` and `min` properties of the `occur` property. For example, consider the same membership list described under "Forms with Repeated Fields or Subforms" on page 202, converted to a dynamic form. The `Member` subform is set to repeat just as many times as the data requires. The following figure shows the result when the Form DOM is filled with the same data as before and printed or displayed. Note that the detail line is present exactly twice, once for each supplied detail record. If there had been 17 members there would have been 17 detail lines and no more.

Dynamic membership form after merge with data

The membership list example is highly simplified compared to forms used in business. Consider the requirements for a dynamic purchase order form. This form must grow to as many detail lines (purchased items) as required by the data. In addition, there must be fields holding the subtotal, taxes, and the grand total, which must move down the page as the list of detail lines grows. Also, there must be a subform containing delivery instructions which must only be included if there are delivery instructions in the data. The following figure shows the result of merging the template with typical data.



Dynamic purchase order form after merge with data

This example still does not illustrate the full capabilities of dynamic forms. Dynamic forms can span columns and pages, with the number of columns and/or pages determined by the amount of data. However columnization and pagination are not parts of the data binding process. They are done downstream in a stage referred to as the layout process. The layout process handles all physical layout issues such as flowing text across content regions and pages. The layout process can also insert leaders, trailers, and bookends. See "Layout for Dynamic Forms" on page 310 for more information about the layout process. Addressing these issues of presentation is *not* the job of the data binding process. The job of data binding is simply to build the correct logical association between data nodes and template nodes, and encapsulate that association in document order in the Form DOM.

Variable Number of Subforms

What makes a dynamic subform dynamic is that it has values other than 1 for its minimum and maximum occurrences. The `min` attribute of the `occur` property of the subform determines its minimum occurrences. When the subform is copied into the Form DOM this number of copies are created to start with. A value of 0 makes the subform optional. If and when all of these copies are bound to data, and more data remains that could bind to additional copies, the `max` attribute of the `occur` property limits how many more copies can be added. If the value of `max` is -1 the only limit is the amount of data available. The `occur` property for a subform set works exactly the same way.

The following example shows the dynamic membership list template corresponding to the figure on page 287, omitting decorative elements. The attributes that make it dynamic have been highlighted in **bold**.

Example 9.1 Dynamic membership list template

```
<template ……>
   <subform name="Members">
      <pageSet …>…</pageSet>
      <field name="Date" …> … </field>
      <subform name="Member">
         <occur min="1" max="20"/>
         <field name="First" …> … </field>
         <field name="Last" …> … </field>
      </subform>
   </subform>
</template>
```

In this example the minimum number of detail lines (member's names) that will be included when merging with data is one, from the `min` attribute. The maximum is twenty, from the `max` attribute.

Note that when any of the attributes is omitted from the `occur` element, the value of the corresponding property defaults to 1. In the absence of an `occur` sub-element all of its properties default to 1. Hence the default behavior is for a subform to be incorporated exactly once into the Form DOM whether or not there is data, that is, to behave as a static subform.

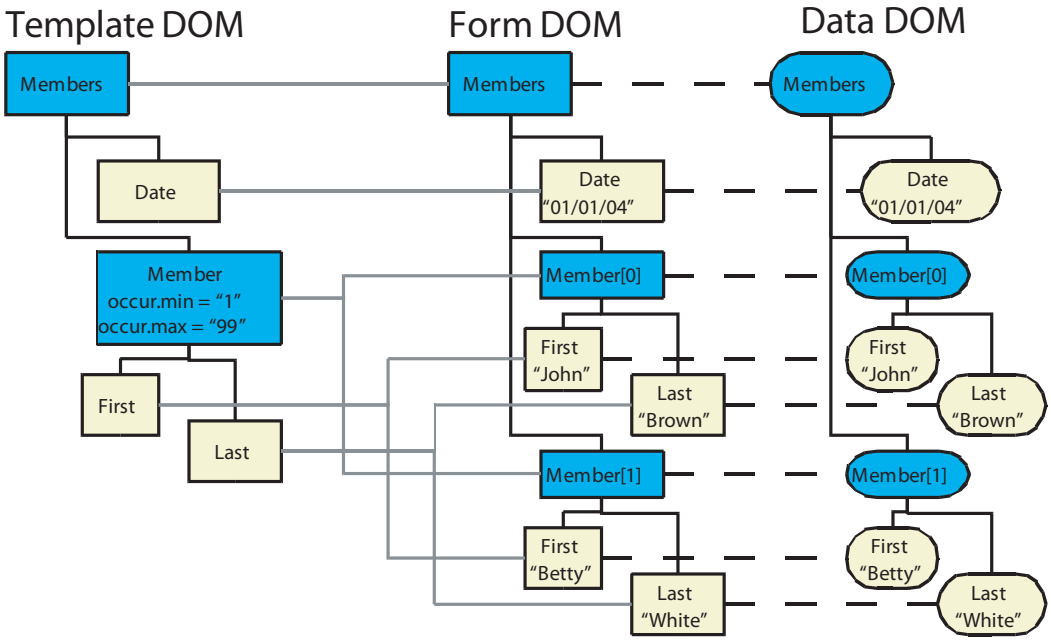
In the membership list example the minimum occurrence for the `Detail` subform defaults to 1 and the maximum is 20. The minimum of 1 means that the subform must be copied at least once into the Form DOM, even if there is none of the data matches it. The maximum of 20 means that it can be copied at most twenty times into the Form DOM. If the data file contained a twenty-first `Member` data group, it would if possible bind to some other subform. In this case there would be no other subform for it to bind to, so it would simply be ignored.

Normally, for the template to be valid, the maximum occurrence value must be an integer greater than or equal to the minimum occurrence value. However a value of -1 for the maximum occurrence is special. It means that the number of occurrences is unlimited. When the maximum occurrence is -1 the minimum occurrence can have any value greater than or equal to zero.

A maximum occurrence of -1 is very commonly used for dynamic subforms. When the form is to be displayed on a graphics display the unlimited scrolling length of the virtual page suits the unlimited length of the sequence of subforms. However when printed to paper the sequence of subforms must be broken up into properly paginated units. This is performed downstream by the layout process, as described in "Layout for Dynamic Forms" on page 310 and has no effect on data binding.

Note that the minimum occurrence must be an integer greater than or equal to zero. In addition, it must be less than or equal to the maximum occurrence value unless the maximum occurrence value is -1. If either of these conditions is violated the template is invalid.

When a subform has a variable number of occurrences, the data binding process starts by creating the specified minimum number of copies of the subform in the Form DOM. Then it seeks matching data objects for each of these in turn. If it finds matches for all of them, and there is still another potential match, it adds another copy and binds this to the next match. It continues adding more copies and binding them as long as the total number of copies is less than the maximum and there is at least one more match. In the example, it starts with one copy (`$form.Members.Member[0]`), because this is the minimum, and binds it to the first `Member` data group (`$data.Members.Member[0]`). Proceeding in template document order, it descends into the subform and the data group and binds the fields to the data values. Returning to the `$form.Members` level, it finds that it is allowed to add another copy of the same subform and also there is a match for it, so it adds `$form.Members.Member[1]`, binding it to `$data.Member.Members[1]`, then descends into these and binds fields to data values. After this the data binding process finds that, although it is allowed to add more copies of the same subform, there would be no matches for the copies. Hence it stops adding copies of `$template.Members.Member` and returns to the next higher level in the template (`$template.Members`) where it looks for the next child of `Members` to copy into the Form DOM – but there isn't one, so it is finished. The effect is the same as if the `Members` subform was declared twice in the template – just as many times as the data requires – and each `Members` subform along with its contents was processed in document order. The following figure shows the resulting relationship between the DOMs.



DOMs resulting from dynamic membership list example

The template for the dynamic purchase order on page 288, omitting decorative elements, is as follows.

Example 9.2 Dynamic purchase order template

```
<template …>
   <subform name="PO">
      <pageSet …>…</pageSet>
      <field name="PO_Date" …> … </field>
      <field name="ReqNo" …> … </field>
      <field name="Vendor_Code" …> … </field>
      <field name="Vendor_Name" …> … </field>
      <field name="VendAddr1" …> … </field>
      <field name="VendAddr2" …> … </field>
      <field name="VendAddr3" …> … </field>
      <subform name="Detail">
         <occur max="-1"/>
         <field name="Item"…> … </field>
         <field name="Quantity"…> … </field>
         <field name="Units"…> … </field>
         <field name="Unit_Price"…> … </field>
         <field name="Total_Price"…> … </field>
      </subform>
      <field name="Sub_Total"…> … </field>
      <field name="Tax"…> … </field>
      <field name="Total"…> … </field>
      <subform name="Delivery">
         <occur min="0"/>
         <field name="Del_Instrctn"…> … </field>
      </subform>
   </subform>
</template>
```

The `Detail` subform has no maximum occurrences and defaults to a minimum of one. This is typical for a subform corresponding to one in a list of records. The `Delivery` subform has no minimum occurrences and defaults to a maximum of one. This is typical for a record which is optional.

The DOM relationships for this example are not shown here because the drawing would not fit in the space available.

Note that fields do not have `occur` properties, hence can not automatically repeat. It is common to wrap a field in a subform simply to provide a way to associate an `occur` property indirectly with the field. In such cases it may be convenient to leave the subform nameless so it does not alter the SOM expression used to refer to the field in scripts. Alternatively, setting its `scope` property to `none` causes it to appear in SOM expressions but to be transparent to the data binding process so it has no effect on the data hierarchy.

The Occur Element

Each subform and subform set object has an *occur* property which has three sub-properties, *min*, *max*, and *initial*. It governs how many iterations of the subform or subform set are required and how many are permitted.

In a template the occur property is expressed as an `occur` element. It can be the child of a subform or subform set. When the `occur` element is missing, all of the sub-properties default to 1. If the element is present but any of its attributes is missing, the sub-property associated with that attribute defaults to 1.

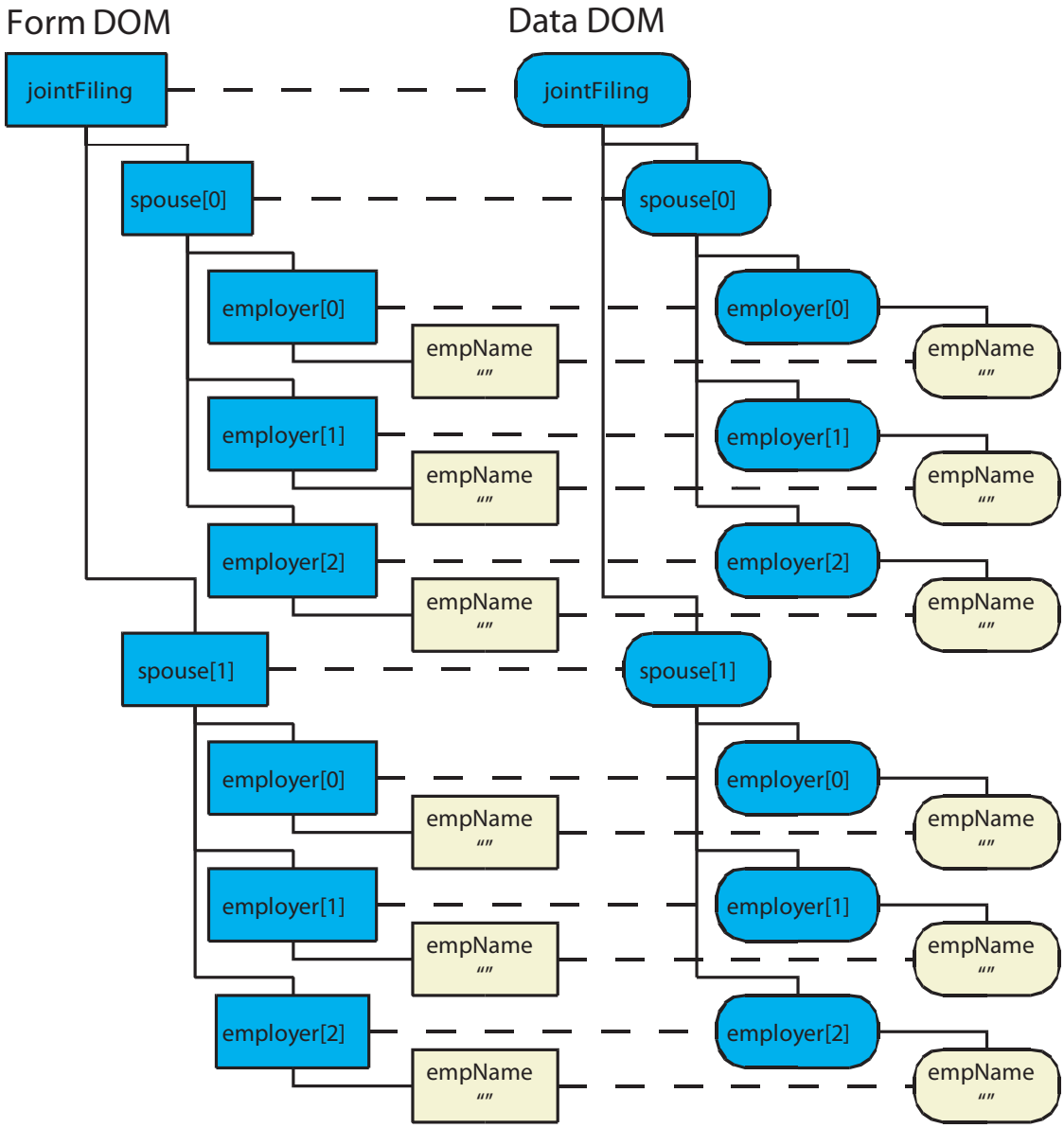
The initial property

The initial property determines how many copies of the subform or subform set are included in the Form DOM as siblings during an empty merge. The value of this property must be a non-negative integer. For example, consider the following template fragment.

Example 9.3 Tax template using the initial occurrence property

```
<template>
   <subform name="jointFiling">
      <subform name="spouse">
         <occur initial="2"/>
         <subform name="employer">
            <occur initial="3"/>
            <field name="empName">...</field>
         </subform>
      </subform>
   </subform>
</template>
```

The following figure shows the Form and Data DOMs that result when the above template is processed through the data binding process without data. The Template DOM has been omitted to save space.



Effect of the initial properties of nested subforms

When the `initial` attribute is not supplied, this property defaults to the value of the `min` property. Note that the root (outermost) subform in a template must have its `initial` property explicitly or by default set to 1.

This property is ignored when merging with a non-empty data document. It is also ignored when the object to which it applies is the child of a subform set and the subform set enforces a `choice` between children.

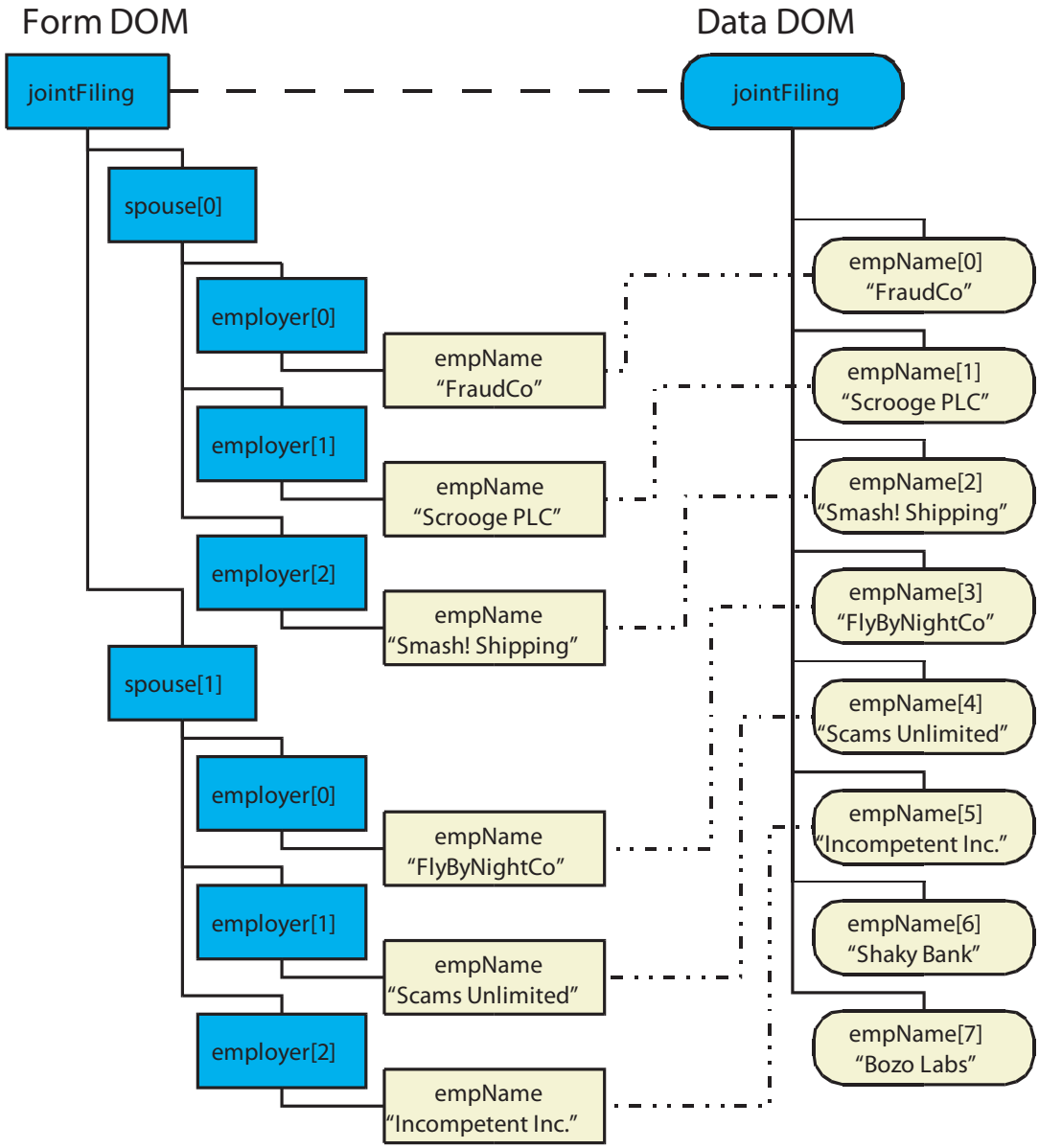
The max property

The `max` property determines the maximum number of copies of the subform or subform set that may be included in the Form DOM as siblings during a non-empty merge. Once this number has been reached the subform or subform set is considered exhausted and no more siblings may be inserted at that location. However if a subform or subform set that is higher in the chain of ancestors is not yet exhausted, the data binding process may insert another copy of that higher-level node, then add descendants to that node including a new set of siblings copied from this same subform or subform set. For example, the following template fragment includes a higher-level subform (`spouse`) with a maximum occurrence of 2 and a lower-level subform (`employer`) with a maximum occurrence of 3.

Example 9.4 Tax template using the maximum occurrence property

```
<template>
   <subform name="jointFiling">
      <subform name="spouse">
         <occur max="2"/>
         <subform name="employer">
            <occur max="3"/>
            <field name="empName">...</field>
         </subform>
      </subform>
   </subform>
</template>
```

Given a flat data document with eight `empName` elements, the resulting Form and Data DOMs, before renormalization, are shown in the following figure. Data values after the first six are left unbound because there are no unbound `empName` fields left for them to bind with.



Effect of the max properties of nested subforms

When the `max` attribute is not supplied this property defaults to the value of the `min` property. Note that the root (outermost) subform of a template must have this property defaulted or explicitly set to 1.

This property is ignored during an empty merge.

The min property

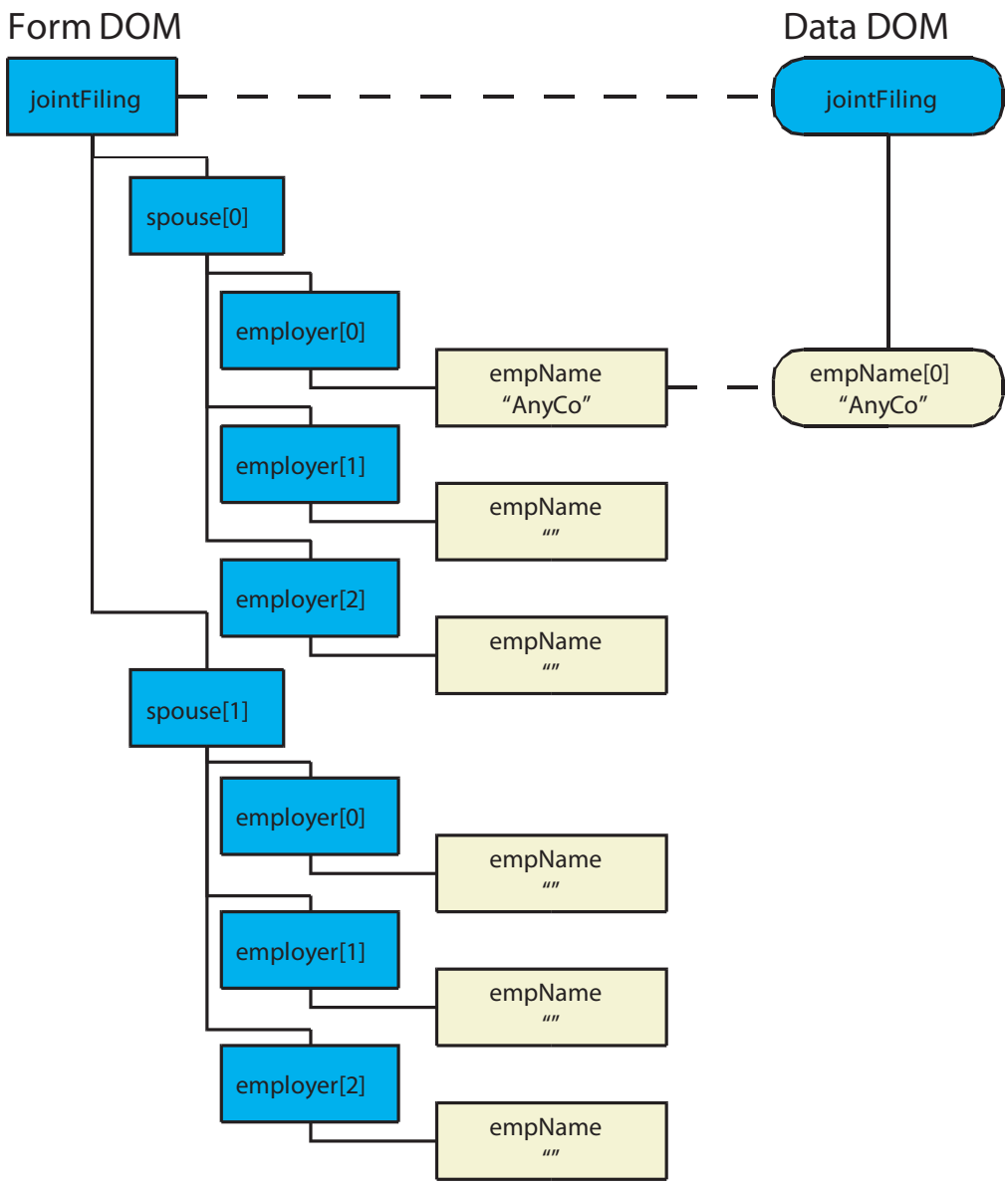
The min property determines the starting number of copies of the subform or subform set that are included in the Form DOM as siblings during a non-empty merge. This number of siblings is created whenever the subform or subform set is used as a prototype for a node in a new location in the Form DOM,

where there were no siblings copied from it before. If the same subform or subform set is subsequently used as a prototype somewhere else in the Form DOM the same starting number of siblings is created there too. For example, the following template fragment has a higher-level subform (`spouse`) with a minimum occurrence of 2 and a lower-level subform (`employer`) with a minimum occurrence of 3.

Example 9.5 Tax template using the minimum occurrence property

```
<template>
   <subform name="jointFiling">
      <subform name="spouse">
         <occur min="2"/>
         <subform name="employer">
            <occur min="3"/>
            <field name="empName">...</field>
         </subform>
      </subform>
   </subform>
</template>
```

Given a data document with a single data value named `empName`, the resulting Form DOM is shown in the following figure.



Effect of the min properties of nested subforms

When the `min` attribute is not supplied this property defaults to 1. Note that the root (outermost) subform of a template must have this property defaulted or explicitly set to 1.

This property is ignored during an empty merge. It is also ignored when the object to which it applies is the child of a subform set and the subform set enforces a `choice` between children.

Blank Form

When a dynamic subforms or subform sets is merged with data, the data determines (at least partly) the number of times the subform or subform set is copied into the Form DOM. But what is to be done when there is no data, that is during an empty merge? A separate attribute (`initial`) is defined which controls

how many copies of the subform or subform set are incorporated into the Form DOM during an empty merge. Generally `initial` will be equal to 1 or to the value of `min` or `max`. It does not make much sense to set `initial` to a value larger than `max` but it is not forbidden to do so. In fact `initial` is always used during an empty merge, even for static subforms and subform sets, so one could perversely set `min` and `max` to the same value but `initial` to some other value. Doing so is not recommended.

The garden club diagram on page 287 shows the membership list as printed after merging with data. Compare this to the figure at right, which shows the same dynamic form after an empty merge.

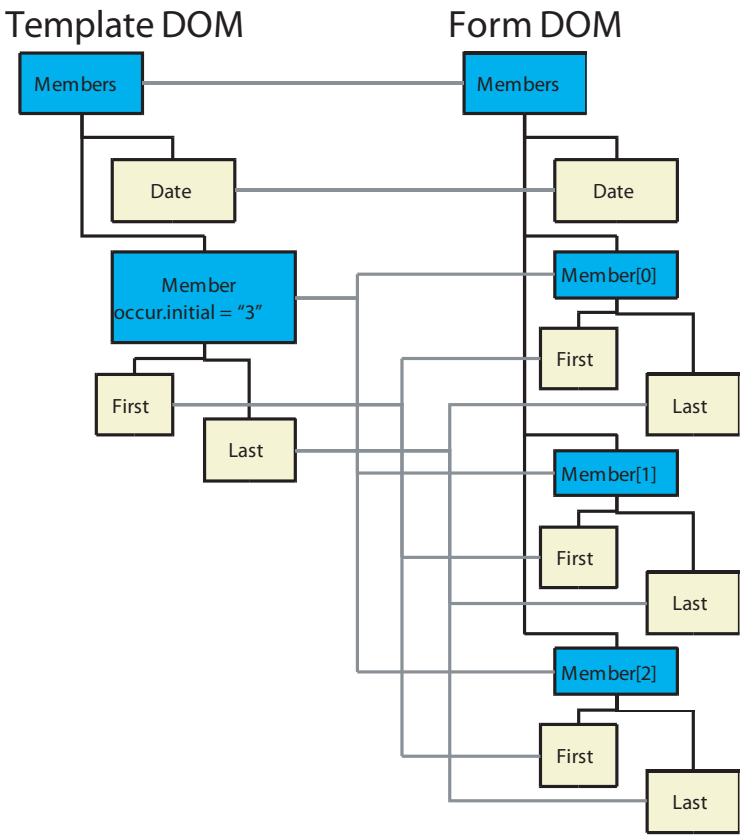
The template was defined in Example 9.1. For convenience it is repeated below.

Dynamic membership form after an empty merge

Example 9.6 Dynamic membership list template

```
<template …>
   <subform name="Members">
      <field name="Date" …> … </field>
      <subform name="Member">
         <occur min="1" max="20"/>
         <field name="First" …>
          … </field>
         <field name="Last" …>
          … </field>
      </subform>
   </subform>
</template>
```

Since no value was supplied for `initial`, it defaulted to one. Hence the data binding process placed a single copy of the `Member` subform into the Form DOM. The resulting relationship between the DOMs is shown in the following figure.



Relationship between the DOMs of the membership form after an empty merge

Greedy Matching

Once the data binding process has introduced a subform into the Form DOM, and the number of occurrences is variable, the data binding process tries to match the full permitted number of siblings to the data. This is referred to as *greedy matching*. But some of the matches may be indirect matches. These indirect matches sometimes lead to non-intuitive results. For example, consider the following template fragment from a passport application.

Example 9.7 Passport application template

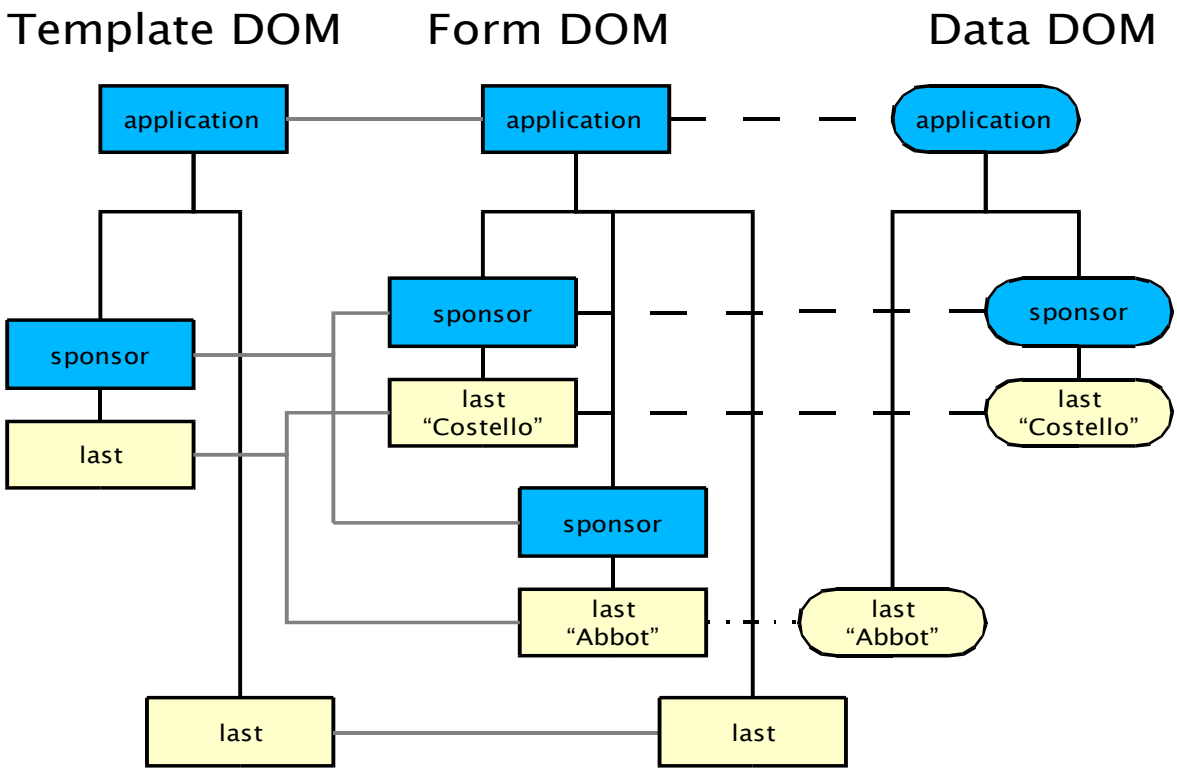
```
<template>
   <subform name="application">
      <subform name="sponsor">
         <occur max="7"/> <!-- up to seven sponsors -->
         <field name="last"> … </field> <!-- sponsor's last name -->
         …
      </subform>
      <field name="last"> … </field> <!-- applicant's last name -->
      …
   </subform>
</template>
```

This template is merged with the following data.

Example 9.8 Passport application data

```
<application>
   <last>Abbot</last>
   …
   <sponsor>
      <last>Costello</last>
   </sponsor>
</application>
```

The result is shown in the following figure.



Scope-matching leads to dynamic subform greedily devouring data it shouldn't

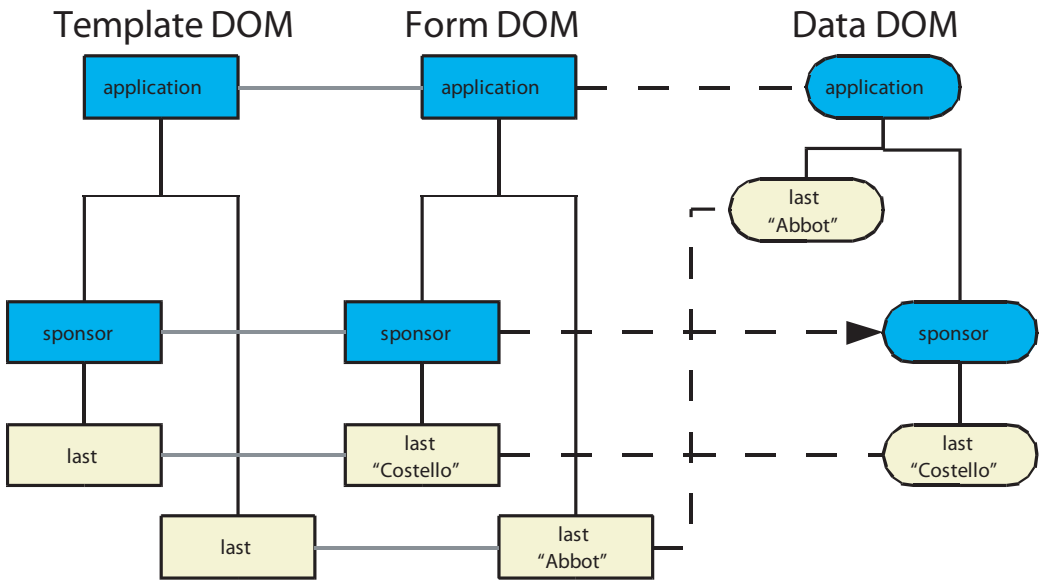
At first glance this is a surprising result. Recall that data binding traverses the Template DOM in document order. In the Template DOM the `sponsor` subform precedes its sibling `last` field, which is intended for the applicant's last name. So, the XFA processor adds a `sponsor` subform to the Form DOM and binds it to the `sponsor` data group in the Data DOM. The `last` field beneath this subform correctly binds to the data value containing "`Costello`". However the `sponsor` subform is allowed to occur up to 7 times. Due to greedy matching the XFA processor replicates the `sponsor` subform and the `last` field within it in order to bind the field to the applicant's last name (`"Abbot"`). This is a valid binding because `$data.application.last` scope-matches to `$form.application.sponsor.last`. By the time the field for the applicant's last name is processed all the data has already been bound, so this field is left unbound and when the form is displayed it remains blank.

The same remedies apply in this case as apply generally when scope-matching produces an undesirable result. The various remedies are listed in Matching Hierarchy. Here is the same template fragment with an explicit data reference added to fix the problem.

Example 9.9 Passport application template fixed to prevent greedy matching problem

```
<template>
   <subform name="application">
      <subform name="sponsor">
         <occur max="7"/> <!-- up to seven sponsors -->
         <bind match="dataRef" ref="sponsor"/>
         <field name="last"> … </field> <!-- sponsor's last name -->
         …
      </subform>
      <field name="last"> … </field> <!-- applicant's last name -->
      …
   </subform>
</template>
```

The result using this template fragment and the same data is shown in the following figure. This is the desired result.



Explicit data reference prevents unwanted scope-matching

Globals

A field or exclusion group can bind to global data, as described in "Globals" on page 214. However globals play a passive role in data binding. That is, the binding process does not drag in a subform into the Form DOM just because the subform contains a field or exclusion group that matches to a global. However once a subform has been dragged into the Form DOM, any field or exclusion group within that subform that does not match non-global data may fall back to binding with global data.

Explicit Data References

An explicit data reference (using the `dataRef` sub-property) may cause a subform, field, or exclusion group node to bind to data which would not otherwise match it, as described in "Explicit Data References" on page 177. During dynamic binding this is treated as a special case. If the data has not already been bound to a node in the Form DOM then the subform, field, or exclusion group declaring the explicit data reference is dragged into the Form DOM and a binding is established. However if the data node is already

bound to some other node then the declaring subform, field, or exclusion group is not dragged into the Form DOM. One way of looking at it is that the referenced data is treated as ordinary non-global data as long as it has not yet been bound, but once bound it is treated like global data.

Subform Set

An individual dynamic subform can be omitted or included in response to the presence or absence of data in the Data DOM. A subform set imposes additional constraints upon the inclusion or omission of the set of subforms and/or subform sets which it encloses.

There are three types of subform sets, distinguished by the value of the `relation` attribute. The relation attribute can have any of the values `choice`, `ordered`, and `unordered`.

A choice subform set encloses a set of mutually-exclusive subforms and/or subform sets. Even if the Data DOM contains matches for more than one of the members of the set, only one will be copied into the Form DOM. The one chosen is the first matching one encountered in the Data DOM, when descending it in data order, that is, width-first and oldest to newest (left to right). If there is no match none of the members are included, leaving the subform set node in the Form DOM without any children.

An unordered subform set encloses subforms and/or subform sets that have no special ordering in the template. The whole set is copied into the Form DOM, however the ones (if any) that match data groups are copied first, in data order. The rest are copied in template order.

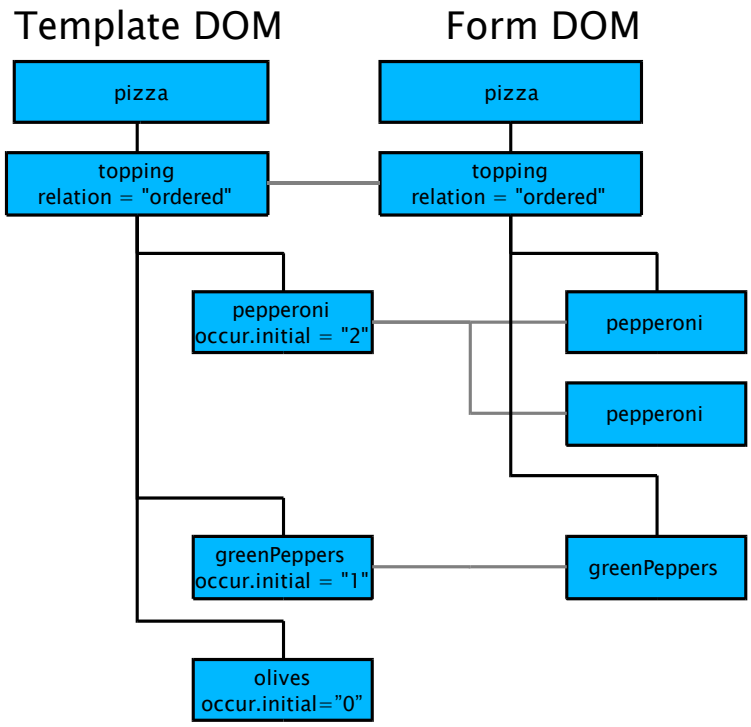
An ordered subform set encloses subforms and/or subform sets that have a special ordering in the template. The whole set is copied into the Form DOM in template order, and then matching data nodes (if any) are bound to them. An ordered subform set is functionally equivalent to a subform with no name.

Subform sets have initial, min, and max occurrence attributes just like the subforms that belong to the set. During an empty merge the initial attribute of the subform set determines how many copies of it are added to the Form DOM, and then the initial attributes of the subforms determine how many copies of each are added to the Form DOM under each copy of the subform set, except for choice subform sets. When a choice subform set is added to the Form DOM only the first of its subforms is copied to the Form DOM regardless of the occurrence attributes of the rest. For example, the following shows a portion of a template for a pizza order. There is a separate subform for each type of pizza topping because each type has different options. Pepperoni can be mild or hot, green peppers can be sliced or chopped, and olives can be green or black. The pizza toppings are contained by an ordered subform set.

Example 9.10 Subform set for pizza toppings

```
<subformSet name="topping" relation="ordered">
   <subform name="pepperoni">
      <occur initial="2"/>
      …
   </subform>
   <subform name="greenPeppers">
      <occur initial="1"/>
      …
   </subform>
   <subform name="olives">
      <occur initial=”0”/>
    … </subform>
</subformSet>
```

The following figure shows part of a Template DOM and the corresponding part of the Form DOM after an empty merge of the ordered subform set. For clarity the fields within the subforms are omitted from the drawing.



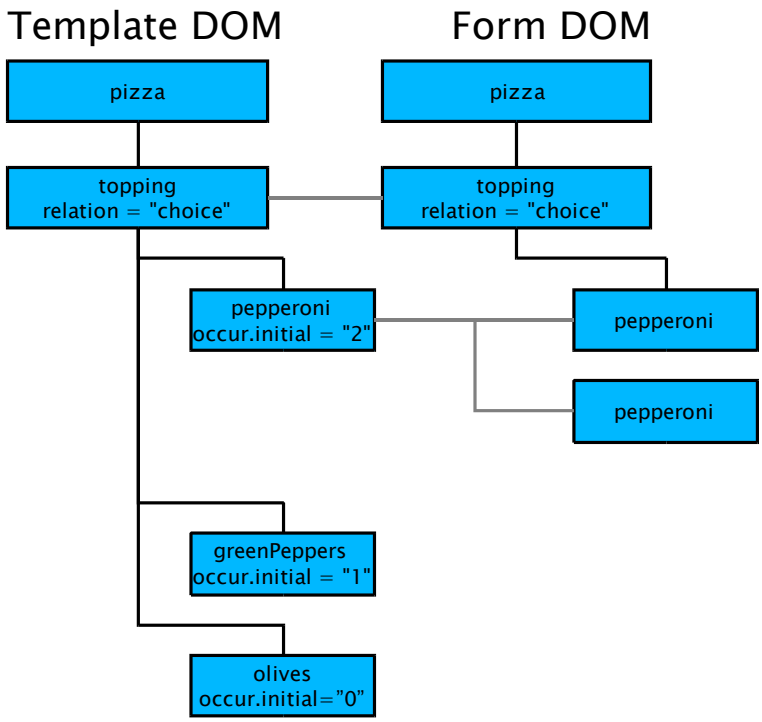
Empty merge of an ordered subform set

Now consider the same fragment modified by changing the subform set from `unordered` to `choice`.

Example 9.11 Previous example changed to use a choice subform set

```
<subformSet name="topping" relation="choice">
   <subform name="pepperoni">
      <occur initial="2">
         …
   </subform>
   <subform name="greenPeppers">
      <occur initial="1">
         …
   </subform>
   <subform name="olives">
      <occur initial=”0”/>
    … </subform>
</subformSet>
```

The following figure shows the result of an empty merge using this subform set. Only the `pepperoni` child is used and the other child subforms are ignored.



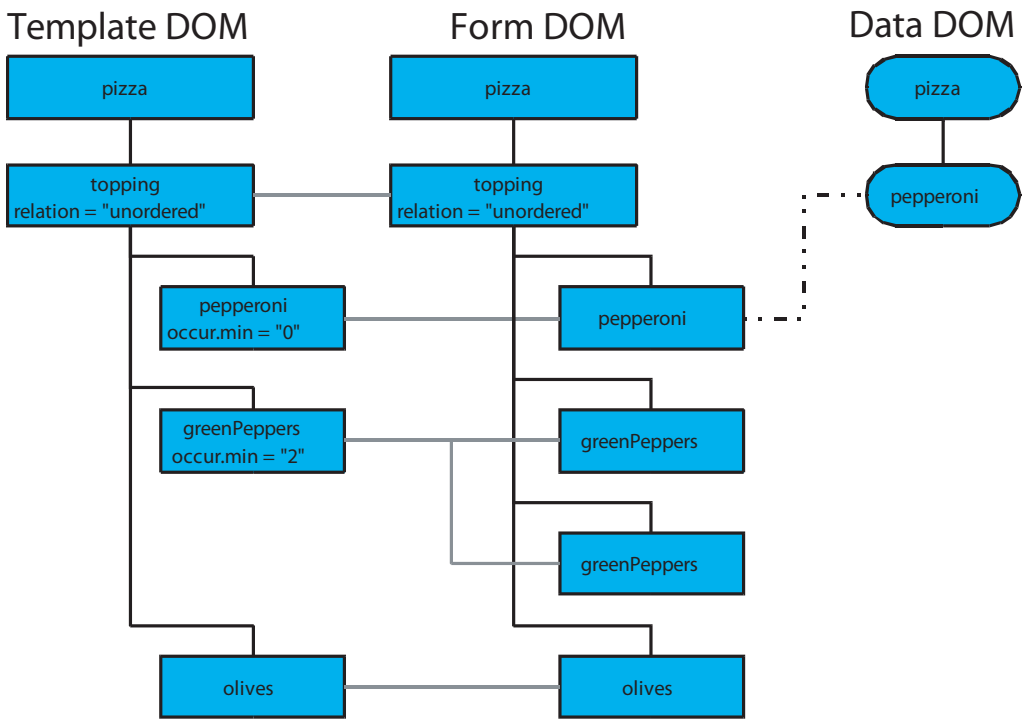
Empty merge of a choice subform set

During a non-empty merge a subform set can be pushed into the form by its own minimum occurrence attribute or drawn into the Form DOM by an indirect match with one of its child subforms. In this regard it is like a subform except that a subform can also be pulled in by a direct match to a data group, whereas a subform set cannot directly match data. For example, consider the following template fragment.

Example 9.12 Subform set for pizza topping with minimum occurrence limits

```
<subformSet name="topping" relation="unordered">
   <subform name="pepperoni">
      <occur min="0">
         …
   </subform>
   <subform name="greenPeppers">
      <occur min="2">
         …
   </subform>
   <subform name="olives"> … </subform>
</subformSet>
```

The following figure shows a the result of a non-empty merge to this fragment, leaving out fields and the data values to which they match. The `toppings` subformSet is dragged into the Form DOM by its `pepperoni` child, which matches a data group. Then, their minimum occurrence attributes force the inclusion of `greenPeppers` and `olives` subforms even though they do not match any data. Note that the `olives` subform merely defaults to a minimum occurrence of 1.



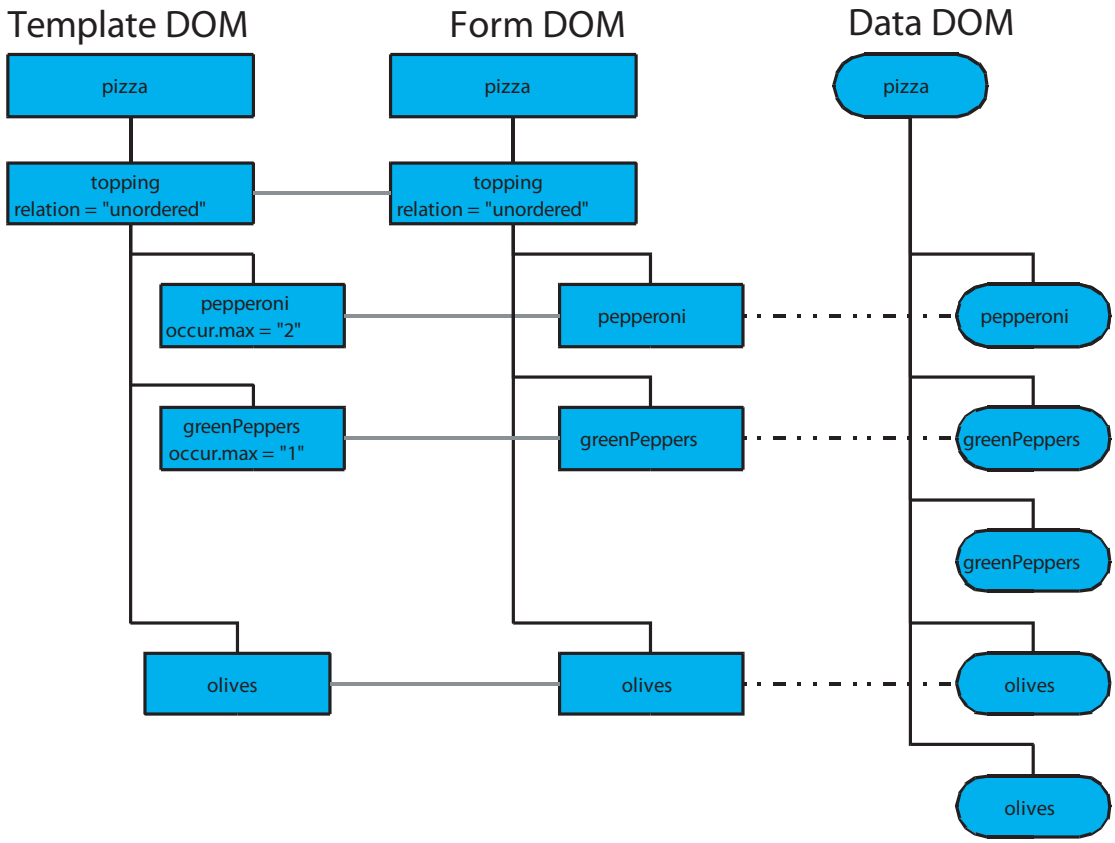
The min attribute forces inclusion of template siblings

Once the first copy of the subform set has been placed in the Form DOM, its maximum occurrence attribute limits how many siblings may be given to the copy. For example, compare the previous example to the following template fragment in which some of the subforms assert maximum rather than minimum occurrence limits.

Example 9.13 Subform set for pizza topping with maximum occurrence limits

```
<subformSet name="topping" relation="unordered">
   <subform name="pepperoni">
      <occur max="2">
         …
   </subform>
   <subform name="greenPeppers">
      <occur max="1">
         …
   </subform>
   <subform name="olives"> … </subform>
</subformSet>
```

The following figure shows a non-empty merge to this fragment, in which the maximum occurrence attributes limit the number of `pepperoni` and `olive` subforms even while some data remains unmatched. Note that the `olives` subform merely defaults to a maximum occurrence of 1.



The max attribute forces exclusion of data siblings

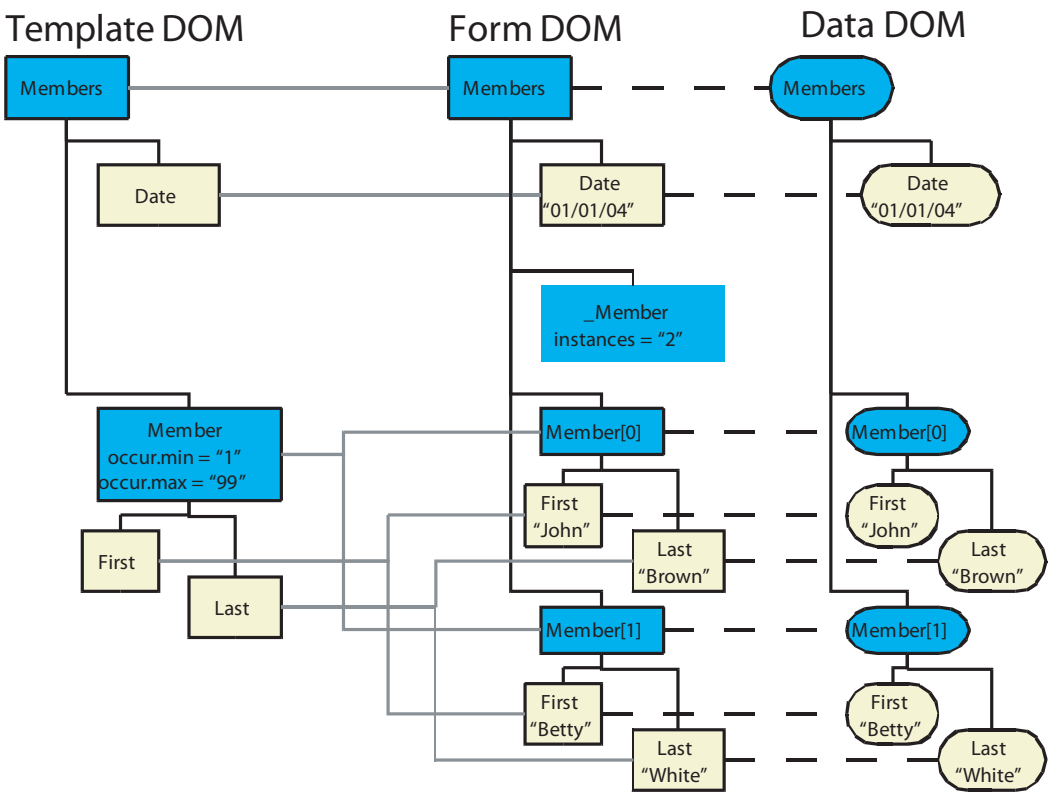
Instance Manager

An *instance manager* is an object placed into the Form DOM by the data binding process for the use of scripts. One instance manager is placed in the Form DOM for each dynamic subform in the Form DOM. Using the instance manager the script can find out how many instances of the subform have been copied into the Form DOM and it can delete instances or insert more instances. When an instance is deleted, if the instance was bound to data, the binding is automatically broken. When a new instance is inserted the instance manager may invoke the data binding process to attempt to bind the new instance.

Each instance manager is the peer of the subforms it manages. It is distinguished from them in two ways. First, it is not a subform object but an instance manager object. Second, its name is different. The name of the instance manager is an underscore (`"_"`) character followed by the name of the subforms it manages. For example, if a subform is called `Member`, the subform manager for that subform is called `_Member`.

Caution: It is legal for XFA node names to start with underscore. This can lead to a name conflict if two sibling subforms have names that differ only by the presence of a leading underscore. It is the responsibility of the form creator to ensure that this does not happen.

Instance managers have been omitted from drawings of the Form DOM elsewhere in this specification in order to reduce clutter. The following figure is another look at the result of merging data with the membership list template, as shown before in the figure "DOMs resulting from dynamic membership list example" on page 290, but this time showing the instance manager.



DOMs after merge showing instance manager

For more information about using instance managers, see "Relative References" on page 94, which is a section in "Scripting Object Model".

Using Fixed Multiple Occurrences for Pseudo-Static Forms

It is permissible for the maximum and minimum occurrences to be set to the same value. If the value is anything other than 1 the result is to invoke the full dynamic form mechanism but constrain it to operate in a pseudo-static manner.

For example, consider the membership list template given as Example 6.14 and reproduced below.

Example 9.14 Membership roster template using repeated subform declarations

```
<template ...>
  <subform name="Members">
    <field name="Date" ...>...</field>
    <subform name="Member">
      <field name="First" ...>...</field>
      <field name="Last" ...>...</field>
    </subform>
    <subform name="Member">
      <field name="First" ...>...</field>
```

```
        <field name="Last" ...>...</field>
      </subform>
      <subform name="Member">
        <field name="First" ...>...</field>
        <field name="Last" ...>...</field>
      </subform>
      </subform>
</template>
```

This can be expressed more concisely using occurrence numbers as follows.

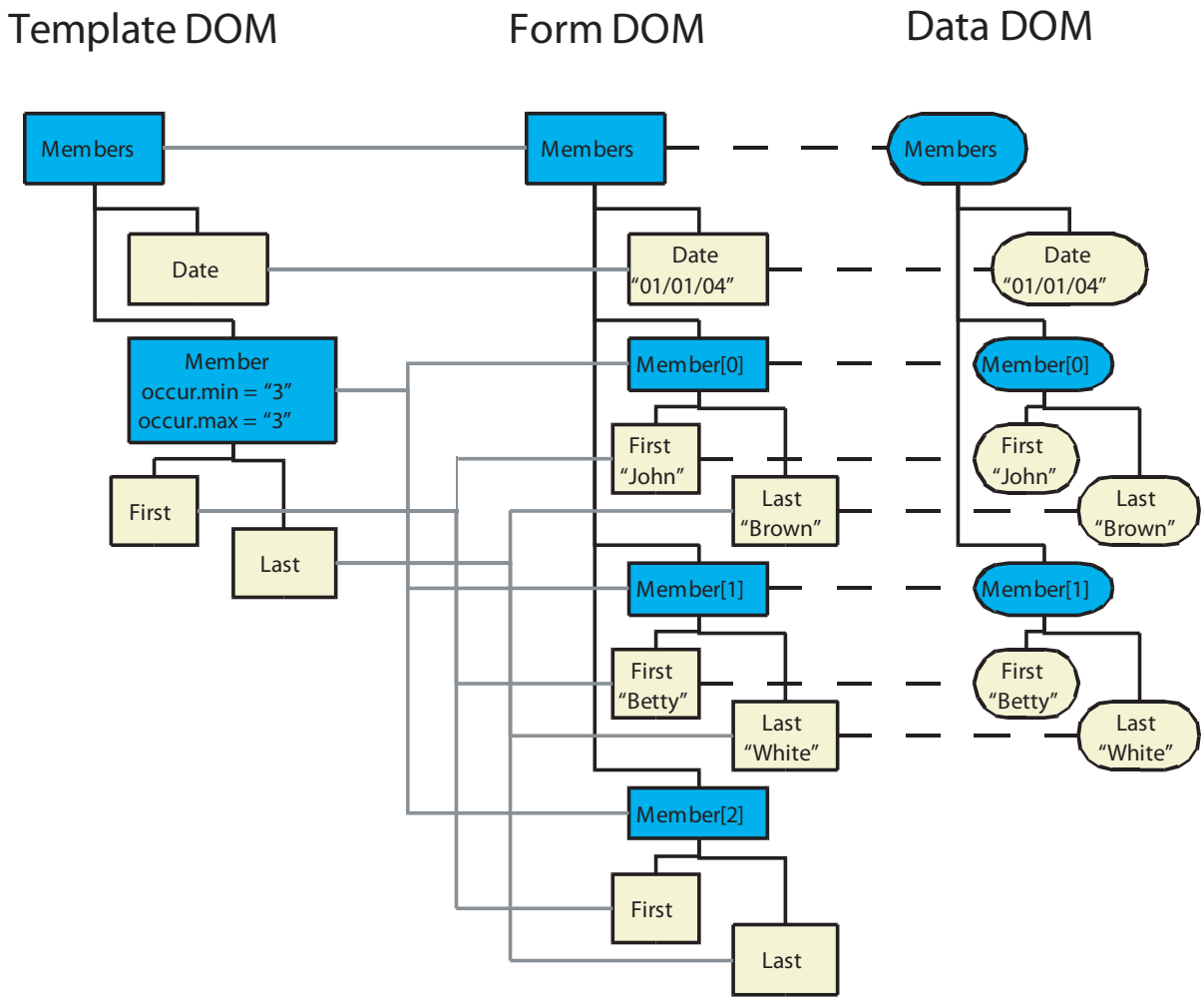
Example 9.15 Using fixed occurrence numbers

```
<template ...>
   <subform name="Members">
      <field name="Date" ...>...</field>
      <subform name="Member">
         <occur min="3" max="3" initial="3"/>
         <field name="First" ...>...</field>
         <field name="Last" ...>...</field>
      </subform>
   </subform>
</template>
```

This is fully equivalent to the earlier representation using three repetitions of the `Member` subform declaration. The Form DOM that results from the data binding operation has the exact same structure except that:

1. The Form DOM contains an instance manager named "_Member".

2. Multiple subforms in the Form DOM share the same prototype in the Template DOM, as shown in the following figure.



Result of binding repeated data groups to a multiply-occurring subform

As it happens, if the `max` attribute is not supplied then the `max` property defaults to the value of `min`. Therefore the above template can be expressed still more compactly as follows.

Example 9.16 Using a default max attribute with fixed occurrence numbers

```
<template ...>
   <subform name="Members">
      <field name="Date" ...>...</field>
      <subform name="Member">
         <occur min="3" initial="3"/>
         <field name="First" ...>...</field>
         <field name="Last" ...>...</field>
      </subform>
   </subform>
</template>
```

Nested subforms and subform sets can have multiple occurrences at each level of nesting. The result is to compound the occurrences. For example, suppose a template has a subform `Member` which is set to occur

three times, and `Member` contains a subform `Name` which is set to occur twice. This is equivalent to a template containing three subforms called `Member`, each of which contains two subforms called `Name`.

Layout for Dynamic Forms

A previous chapter ("Layout for Growable Objects" on page 237) has described the layout algorithm and how it accomplishes flowing layout. Dynamic forms use the same layout algorithm, except that adhesion is modifed slightly. However there are additional layout capabilities which are only useful for dynamic forms. Those additional capabilities are described in this section.

Flowing layout is generally used by dynamic forms so that different instances of the same subform, represented only once in the template, can be placed in different positions upon the page. Furthermore, flowing layout in XFA includes a simple and convenient way to lay out two-dimensional tables.

Adhesion in Dynamic Forms

The layout process treats subformSet objects as transparent. Hence a subform can adhere to another subform even though it is not really a sibling, as long as the two subforms are separated only by subformSet objects. For example, in the following template subform D does adhere to subform C even though they are not true siblings. Subform E also adheres to D.

Example 9.17 Dynamic form using adhesion

```
<template …>
   <subform name="root" layout="tb" …>
      <pageSet …>
         …
      </pageSet>
      <subform name="C" … />
      <subformSet name="X" …>
         <subform name="D" …>
            <keep prev="contentArea" />
            …
         </subform>
      </subformSet>
      <subform name="E" …>
         <keep prev="contentArea" />
         …
      </subform>
   </subform>
</template>
```

Adhesion can and does apply to multiple instances of a dynamic subform (that is, a subform with a variable number of occurrences dictated by the data). It is very common to combine adhesion with dynamic subforms. For example, the following template uses nested dynamic subforms to populate a form with repeated groups of five subforms.

Example 9.18 Template using nested dynamic subforms and adhesion

```
<template …>
   <subform name="root" layout="tb" …>
      <pageSet …>
         <pageArea …>
            <contentArea name="A" … />
            <contentArea name="B" … />
         </pageArea>
      </pageSet>
      <subform name="outer" … />
         <occur max="-1"/>
         <subform name="inner" …>
            <occur max="5"/>
            <keep next="contentArea" intact="contentArea" />
            …
         </subform>
      </subform>
   </subform>
</template>
```

Assume the data contains twelve data items. After data binding the resulting Form DOM (not Layout DOM!) looks like this:

```
[subform (root)]
   [subform (outer[0])]
      [subform (inner[0]) keep.next="contentArea"]
      [subform (inner[1]) keep.next="contentArea"]
      [subform (inner[2]) keep.next="contentArea"]
      [subform (inner[3]) keep.next="contentArea"]
      [subform (inner[4]) keep.next="contentArea"]
   [subform (outer[1])]
      [subform (inner[0]) keep.next="contentArea"]
      [subform (inner[1]) keep.next="contentArea"]
      [subform (inner[2]) keep.next="contentArea"]
      [subform (inner[3]) keep.next="contentArea"]
      [subform (inner[4]) keep.next="contentArea"]
   [subform (outer[2])]
      [subform (inner[0]) keep.next="contentArea"]
      [subform (inner[1]) keep.next="contentArea"]
```

When these objects are inserted into the Layout DOM each group of `inner` subforms adheres together, but there is no adherence between groups of `inner` subforms or between `inner` and `outer` subforms.

Break Conditions for Dynamic Forms

As explained in "Flowing Between ContentArea Objects" on page 254, the layout processor by default moves to a new `contentArea` if and when the current `contentArea` overflows (which can only happen with flowing layout). However the `overflow` property of the object can force the layout processor to move to a different `pageArea` and/or `contentArea` when the overflow occurs, as discussed in "Break on Entry" on page 312. In addition the object's `breakBefore` property can force the layout processor to move to a different `pageArea` and/or `contentArea` just before the layout processor places the object, as discussed in "Break on Exit" on page 316. Finally, the object's `breakAfter` property can force it to move to a different `pageArea` and/or `contentArea` after it has placed the object, as discussed in "Break on Overflow" on page 317.

Note: The preferred syntax for expressing break conditions changed in XFA 2.4. The old syntax using the `break` element is still permitted but is deprecated and will be removed from some future version of this specification.

The `overflow`, `breakBefore`, and `breakAfter` elements can force the layout processor to go to a particular `contentArea` or a particular `pageArea`. In either case, an attribute of the element provides either an XML ID or a SOM expression identifying the target. If there is an object of the correct type matching the XML ID or SOM expression the layout processor traverses the subtree below the `pageArea` in the Template DOM, taking the shortest route to get from the node corresponding to the current `contentArea` to the target node. (This traversal may have side-effects, which are discussed in "Leaders and Trailers" on page 267.) On the descending part of the traversal it adds new instances of `pageSet` and/or `pageArea` objects as appropriate to the Layout DOM. When the destination of the break is a `pageArea`, the layout processor then descends into a `contentArea`, adding new instances of `contentArea` objects to the Layout DOM if necessary.

The break target can be specified via a SOM expression. The expression is evaluated in the context of the object that is currently being laid down. This is consistent with the rules for evaluating prototype references using SOM expressions; the *using* object supplies the context.

A before or after break condition (but not an overflow break condition) can be controlled by a script at layout time. If a non-empty script is supplied within a `breakBefore` or `breakAfter` element, the layout processor executes the script at the appropriate time (just before or just after laying down the object). The script must return a Boolean value. If the script returns True the break is executed. If the the script returns False the break is inhibited.

The three types of layout breaks are processed at a different times and under different circumstances. Consequently a single object may freely assert any two or all three. If the object being placed asserts `breakAfter` and the next object to be placed asserts `breakBefore` the two breaks are processed sequentially.

Break conditions are also used with positioned layout, as discussed in "Break Conditions" on page 232.

Break on Entry

A subform may specify default behavior explicitly via a `breakBefore` element with a `targetType` attribute having a value of `auto`. Alternatively, it may specify that it must be placed inside an instance of a particular `contentArea` or `pageArea`. If the breakBefore element has a `targetType` property with a value of `pageArea` or `contentArea`, the layout processor gets a target specification from the value of the element's `target` property. If the target specification starts with the character '#' it is evaluated as a reference to an XML ID; otherwise it is evaluated as a SOM expression. If there is an object of the correct type matching the target specification the layout processor first checks whether the current container is within a `contentArea` or `pageArea` that is an instance of the one specified. If it is not, the layout processor breaks to the specified target. If the target is a `pageArea` then the layout processor traverses to the first unused (empty) child `contentArea`, as it would when flowing from one `contentArea` to the next. For example, a template contains the following declarations.

Example 9.19 Template employing break on entry into the root subform

```
<template …>
  <subform name="X">
    <breakBefore targetType="pageArea" target="#E_ID"/>
    <pageSet name="A">
      <pageArea name="B">
        <contentArea name="C" … />
```

```
          <contentArea name="D" … />
        </pageArea>
        <pageArea name="E" ID="E_ID">
          <contentArea name="F" … />
        </pageArea>
      </pageSet>
      <field name="Y">
    </subform>
</template>
```

The Form DOM contains the following content:

```
[subform (X)]
   [field (Y) = "some user-supplied data"]
```

At startup the layout processor would by default descend into the first `contentArea` (C) of the first `pageArea` (B) of the first `pageSet` (A) of the root subform (X). Another way of looking at this is that by default there is an implied break to the first `pageArea` of the root subform. However, subform X asserts an explicit break to the `pageArea` with ID `E_ID`. This happens to be `pageArea` E. The layout processor traverses the tree of `pageArea` and `contentArea` nodes until it reaches the specified `pageArea`. Then it descends into the first `contentArea` there to place the layout content. The resulting Layout DOM is:

```
[root]
   [pageArea (E)]
      [contentArea (F)]
         [subform (X)]
            [field (Y) = "some user-supplied data"]
```

➤ **Error Condition: Invalid break target**

A conforming template must not supply a target specification for the `breakBefore` element that does not resolve to exactly one `pageArea` or `contentArea`. However it is anticipated that layout processors will encounter some templates that are not conforming in this way. It is recommended that in such a case the layout processor emit a warning and go to the next available `pageArea` or `contentArea`.

Break to Empty pageArea or contentArea

A layout object may specify that it must start a new `pageArea` or `contentArea` without regard to the current `pageArea`. This is done by specifying a `break`Before element having an attribute of `startNew` with a value of 1 and an attribute of `targetType` with a value of `pageArea` or `contentArea`. When the layout processor encounters such an object it traverses to a new instance of the current `pageArea` or `contentArea`. For example, a template contains the following declarations.

Example 9.20 Template using break before with startNew

```
<template>
   <subform name="W">
      <pageSet name="A">
         <pageArea name="B">
            <contentArea name="C" … />
         </pageArea>
      </pageSet>
      <subform name="X">
         <breakBefore targetType="pageArea" startNew="1"/>
         <field name="Y"/>
      </subform>
   </subform>
</template>
```

The Form DOM contains the following content:

```
[subform (X[0])]
   [field (Y) = "data from first record"]
[subform (X[1])]
   [field (Y) = "data from second record"]
```

At startup the layout processor descends into the first `contentArea` (C) of the first `pageArea` (B) of the first `pageSet` (A). The first content it finds in the Form DOM is subform X[0], which asserts that it must be placed into a new `pageArea`. This forces the layout processor to leave the current `pageArea` (even though it is empty) and create a new one. Then the layout processor places the field and its text "`data from first record`" into the instance of `contentArea` C. This small amount of text does not fill `contentArea` C. Now it comes to the second instance of subform X (X[1]). Again the startNew condition forces it to start a new `pageArea`, the third instance of `pageArea` B. After this it adds a new instance of `contentArea` C and places subform X[1] and its field into the new instance of `contentArea` C. The resulting Layout DOM is:

```
[root]
   [pageset (A)]
      [pageArea (B[0])]
         [contentArea (C)]
            [subform (W)]
      [pageArea (B[1])]
         [contentArea (C)]
            [subform (W)]
               [subform (X)]
                  [field Y = "data from the first record"]
      [pageArea (B[2])]
         [contentArea (C)]
            [subform (W)]
               [subform (X)]
                  [field Y = "data from the second record"]
```

If the above example (which has no boilerplate) is rendered and printed, the first page is blank.

The root subform (subform W in the above example) may assert `startNew`, but it has no practical effect because the root subform always starts a new `pageArea` and `contentArea`.

Combining startNew with beforeTarget

The same breakBefore element may have both startNew and target attributes. This combination requires the layout processor to fulfill both the startNew and target conditions, that is, the subform must be placed in a pageArea or contentArea that is empty and also corresponds to a template object which matches the target specification.

Conditional Break on Entry

The breakBefore element may contain a script element. When the script is non-empty it is executed by the layout processor just before attempting to place the object. The script must return a Boolean value. If the value is False the before break is inhibited. However if the return value is True the before break is executed. In the absence of a script the break is always executed.

Scripts can be in any supported language. When the scripting language does not have a formal return value mechanism the last right-hand value computed by the script is used. This also allows the use of simple expressions that do not fulfill the syntax requirements for procedures. For example, in the following template fragment the expression is used to force the report to start a new page whenever the list of detail records starts a new customer ID.

Example 9.21 Subform breaking conditionally upon a script

```
<subform name="detailLine">
   <occur min="0" max="-1"/>
   <breakBefore target="#pageX" targetType="pageArea">
      <script>
         if (exists(Entry[-1]))
            then Entry[-1].CustID.value ne Entry.CustID.value
         endif
      </script>
   </breakBefore>
</subform>
```

Note that the script is, as usual, evaluated in the context of the enclosing subform. The expression used tests for the existence of the previous record in order to avoid a run-time error on the first record. As a side effect on the first record the before break is inhibited because the script returns the value False which was returned by the exists() function. For all other records the customer ID for the current and previous records are compared and the break is executed only when they differ.

Inserting a Trailer

A trailer is an object which is laid down before any other action is carried out. In particular it is laid down before any movement to another pageArea or contentArea as mandated by the before break. A trailer on a before break has the same effect as a leader on an after break asserted by the previous layout object.

In the following template fragment a before break trailer is used to lay down a message indicating that the text is continued overleaf.

Example 9.22 Subform using a trailer on a before break

```
<subform name="bigSubform">
   <breakBefore targetType="pageArea" trailer="#continued"/>
</subform>
```

If the breakBefore element supplies a script the trailer is only laid down if the script returns True..

Inserting a Leader

A leader is an object which is laid down just before the before the object asserting breakBefore but after the break mandate has been carried out, that is, after moving (if necessary) to the break target.

In the following template fragment a before break leader is used to lay down a heading. This is a common use of before break headers.

Example 9.23 Subform using a leader on a before break

```
<subform name="TermsAndConditions">
   <breakBefore targetType="pageArea" leader="#TAndCTitle"/>
</subform>
```

If the `breakBefore` element supplies a script the trailer is only laid down if the script returns True.

A `breakBefore` leader may be combined with a bookend leader. The bookend leader is treated like part of the object that is being laid down, hence after the break mandate has been carried out the `breakBefore` leader is laid down first, then the bookend leader, then the content of the object itself.

Break on Exit

A layout object may use its `breakAfter` property to force the layout processor to traverse to a different `pageArea` and/or `contentArea` after laying down the object. The semantics and use of this property exactly mirror those of the `breakBefore` property which is discussed in "Break on Entry" on page 312. Despite this symmetry (or perhaps because of it) the handling of leaders and trailers may be confusing. When `breakAfter` specifies a trailer the trailer is placed after the current layout object but before moving to the break target. Conversely when `breakAfter` specifies a leader the leader is placed after moving to the break target. The same effect is produced when an object is used as a leader for an after break or as a trailer for the next object's before break. For example, assume that the Form DOM contains:

```
[subform (root)]
   [subform (TransactionDetails)]
   [subform (TermsAndConditions)]
```

Suppose the template contains the following fragment.

Example 9.24 Fragment using a leader on an after break

```
<subform name="TransactionDetails">
   <breakAfter targetType="pageArea" leader="#continued"/>
   ...
</subform>
<subform name="TermsAndConditions">
   ...
</subform>
```

This produces the same result as the following.

Example 9.25 Equivalent fragment using a trailer on a before break

```
<subform name="TransactionDetails">
   ...
</subform>
<subform name="TermsAndConditions">
   <breakBefore targetType="pageArea" trailer="#continued"/>
```

```
   ...
</subform>
```

A `breakAfter` trailer may be combined with a bookend trailer. The bookend trailer is treated like part of the object that is being laid down, hence after the content of the object itself is laid down then the bookend trailer is laid down, followed by the `breakAfter` trailer, and then the break mandate is carried out.

Break on Overflow

A layout object may specify that when it does not fit in the current `pageArea` or `contentArea`, the object (or remaining fragment of the object) must be placed in a `pageArea` or `contentArea` matching a particular XML ID or SOM expression. This is done by specifying an `overflow` element. The XML ID or SOM expression is supplied as the value of the `overflow` element's target attribute. If overflow occurs and there is an object of the correct type matching the target specification, the layout processor breaks to the specified target.

Note that there is no `targetType` attribute for `overflow`. It is not necessary because the `target` attribute uniquely specifies an object. If the value of `target` is the empty string (the default) then the `overflow` property has no effect.

For example, a template contains the following declarations.

Example 9.26 Template using break on overflow

```
<subform name="X">
   <overflow target="#F_ID"/>
   <pageSet name="A">
      <pageArea name="B">
         <contentArea name="C" h="0.2in" w="1in" … />
         <contentArea name="D" … />
      </pageArea>
      <pageArea name="E">
         <contentArea name="F" ID="F_ID" h="0.2in" w="6in" … />
      </pageArea>
   </pageSet>
   <field name="Y"> … </field>
</subform>
```

The Form DOM contains the following content:

```
[subform (X)]
   [field (Y) = "lots and lots of text that overflows the contentArea"]
```

At startup the layout processor descends into the first `contentArea` (C) of the first `pageArea` (B) of the first `pageSet` (A). The first content it encounters in the Form DOM is subform X. It tries to place subform X into `contentArea` C but finds that it doesn't fit. So, it splits the subform and places the top fragment of it into `contentArea` C. At this point the overflow break comes into play. Instead of traversing to `contentArea` D as it would normally do, the layout processor traverses to the overflow target, which is `contentArea` F. There it puts the remainder of subform X (or at least as much of it as fits). Assuming the typeface is Courier and the typesize is 10 points, the result is:

```
[root]
   [pageSet (A)]
      [pageArea (B)]
         [contentArea (C)]
            [subform (X)]
               [field (Y) = "lots and "]
      [pageArea (E)]
      [contentArea (F)]
         [subform (X)]
            [field (Y) = "lots of text that overflows the contentArea"]
```

In this example, the overflow break of subform X affects every new `pageArea` or `contentArea` (unless overridden by a lower-level subform) because the root subform in effect flows through the entire document.

➤ **Error Condition: Invalid break target**

A conforming template must not supply a target specification for the `overflow` element that does not resolve to exactly one `pageArea` or `contentArea`. However it is anticipated that layout processors will encounter some templates that are not conforming in this way. It is recommended that in such a case the layout processor emit a warning and go to the next available `pageArea` or `contentArea`.

Combining Breaks and Occurrence Limits

A template may combine a subform asserting break conditions with `contentArea` and/or `pageArea` objects asserting occurrence limits. The layout processor simultaneously satisfies both the break condition(s) and the occurrence limit(s).

Combining Break and Maximum Occurrence

A maximum occurrence limit may force the layout processor to add nodes to the Layout DOM at a higher level than it would have otherwise done, in order to satisfy a break condition.

For example, a template contains the following declarations.

Example 9.27 Template combining break with maximum occurrence of a page area

```
<template …>
   <subform name="O">
      <pageSet name="A">
         <occur max="-1"/>
         <pageArea name="B" ID="B_ID">
            <occur max="1"/>
            <contentArea name="C" … />
         </pageArea>
      </pageSet>
      <subform name="P">
         <field name="Q"> … </field>
      </subform>
      <subform name="R">
         <breakBefore targetType="pageArea" target="#B_ID">
         <field name="S"> … </field>
      </subform>
   </subform>
```

```
</template>
```

The Form DOM contains the following content:

```
[subform (O)]
   [subform (P)]
      [field (Q) = "text in field Q"]
   [subform (R)]
      [field (S) = "text in field S"]
```

The layout processor lays out subform P first. This does not assert a break condition, so it is processed with default processing rules. After laying out subform P the Layout DOM contains:

```
[root]
   [pageSet (A)]
      [pageArea (B)]
         [contentArea (C)]
            [subform (O)]
               [subform (P)]
                  [field (Q) = "text in field Q"]
```

Subform P does not fill `contentArea` C. However, the next subform to be laid out is R. This subform asserts a `breakBefore` break condition. The break condition could be satisfied by adding another instance of B to the Layout DOM as a sibling of the current `pageArea`. However `pageArea` B has an occurrence limit of 1. In order to respect both this occurrence limit and the break condition, the layout processor ascends to the `pageSet` and adds another sibling in the Layout DOM at that level. Then it descends to the `contentArea` level, adding new nodes to the Layout DOM of as it goes. The result is:

```
[root]
   [pageSet (A[0])]
      [pageArea (B)]
         [contentArea (C)]
            [subform (O)]
               [subform (P)]
                  [field (Q) = "text in field Q"]
   [pageSet (A[1])]
      [pageArea (B)]
         [contentArea (C)]
            [subform (O)]
               [subform (R)]
                  [field (S) = "text in field S"]
```

Combined Break and Minimum Occurrence

A minimum occurrence limit may force the layout processor to add sibling nodes to the Layout DOM that it would otherwise not have added, in order to satisfy a break condition.

For example, a template contains the following declarations.

Example 9.28 Template combining break with minimum occurrence on a page area

```
<template …>
   <subform name="O">
      <pageSet name="A">
         <occur min="1"/>
         <pageArea name="B">
            <contentArea name="C" … />
         </pageArea>
         <pageArea name="D" ID="D_ID">
            <occur min="2"/>
            <contentArea name="E" … />
         </pageArea>
      </pageSet>
      <subform name="P">
         <field name="Q"> … </field>
      </subform>
      <subform name="R">
         <breakBefore targetType="pageArea" target="#D_ID">
         <field name="S"> … </field>
      </subform>
   </subform>
</template>
```

The Form DOM contains the following content:

```
[subform (O)]
   [subform (P)]
      [field (Q) = "text in field Q"]
   [subform (R)]
      [field (S) = "text in field R"]
```

The layout processor starts by descending to the first `contentArea` (C) of the first `pageArea` (B) of the first `pageSet` (A). It puts the subform P into `contentArea` C. At this point the Layout DOM contains:

```
[root]
   [pageSet (A)]
      [pageArea (B)]
         [contentArea (C)]
            [subform (O)]
               [subform (P)]
                  [field (Q) = "text in field Q"]
```

Subform P does not fill `contentArea` C. However, the next subform to be laid out is R, which asserts a `breakBefore` condition at the `pageArea` level. The layout processor satisfies this condition by traversing to `pageArea` D and adding an instance of it to the Layout DOM. However, `pageArea` D asserts a minimum occurrence limit which forces the layout processor to incorporate another instance of it into the Layout DOM. After subform R has been processed the result is:

```
[root]
   [pageSet (A)]
      [pageArea (B)]
         [contentArea (C)]
            [subform (O)]
               [subform (P)]
                  [field (Q) = "text in field Q"]
      [pageArea (D[1])]
```

```
      [contentArea (D)]
         [subform (O)]
            [subform (R)]
               [field (S) = "text in field S"]
   [pageArea (D[2])]
```

Hence when the form is rendered and printed single-sided it will have an extra page at the end. Since the `pageArea` for the extra page includes neither boilerplate nor variable data, the extra page will be blank.

The rules for `pageArea` traversal can be summarized as follows:

1. Both `pageArea` and `pageSet` may assert occurrence limits. For both `pageArea` and `pageSet` the occurrence minimum defaults to zero and the maximum defaults to -1 (no limit).

2. `pageSet` is always considered ordered.

3. Breaking from one `pageSet` or `pageArea` to another `pageSet` or `pageArea` forces the incorporation of instances of any intermediate `pageSet` and `pageArea` objects up to their individual minimum occurrence limits.

4. Unless specified otherwise, the root subform has an implied break to the first `contentArea` in the first `pageArea`.

5. Only the first `pageSet` under the root subform is ever used.

10 Automation Objects

This chapter describes the automation objects: calculate, validate, and event. It describes how automation objects are typical used, how they are activated, and how they interact with other automation objects.

The processing application invokes automation objects in response to a trigger particular to the type of object. Examples of such triggers include a form loading or a user clicking a field.

When an automation object is invoked, it performs some task particular to the type of object. Examples of such tasks include executing a script or executing a web services file. The tasks that can be performed are particular to the type of automation object.

How Script Elements Are Used Within Automation Objects

This section describes how procedural extensions such as calculations, validations, and event handling are specified in a template. The procedural descriptions of how values within a form are validated and calculated are among the central concepts that define what a form is. This is true of both electronic forms and traditional paper-based forms.

Electronic forms may be processed by a wide variety of processing applications. The obvious example is a visual presentation of a form that allows the user to enter data. In such a context, the form can be associated with a set of behaviors that can be described procedurally. This kind of scripting of user-initiated events is common to many applications. This specification recognizes that a form may be part of a much larger process. At each stage in that process, the form may be processed by very different kinds of applications. This specification allows a single form template to describe behaviors appropriate to very different processing applications within that process.

The XFA family of specifications includes a scripting language called FormCalc ["FormCalc Specification" on page 891], a simple expression language that implements a common set of functions useful for calculation. While FormCalc has special status due to the need for interoperable form templates, this specification allows processing applications to support alternative scripting languages such as ECMAScript[ECMAScript].

This specification takes the position that the abstraction of the form object model that is presented to any particular scripting language is not an inherent property of either the form object model or the scripting engine, but is a distinct script binding (not to be confused with data binding). The XFA Scripting Object Model specification ["Scripting Object Model" on page 73] describes a script binding between the form object model and the scripting languages (in particular, FormCalc) that can be used for interoperable form templates.

The related set of values associated with form elements is an essential aspect of what a form represents. This specification defines the following methods for defining the value of a field:

- Set an initial value, by providing non-empty content in a `value` element. A static value would be defined by the template designer.
- Bind data to a value. Such data is provided by an external source, such as a person filling in the form or by a database server.
- Derive a value dynamically, likely from other form values or ambient properties such as date and time. Such a value is defined using the `calculate` element.

- Derive a Boolean value that indicates whether the current value of a form object is valid — validate element

The field object supports both calculations and validations. Additionally, the subform and exclusion group elements also support validation to allow aggregate-level validations.

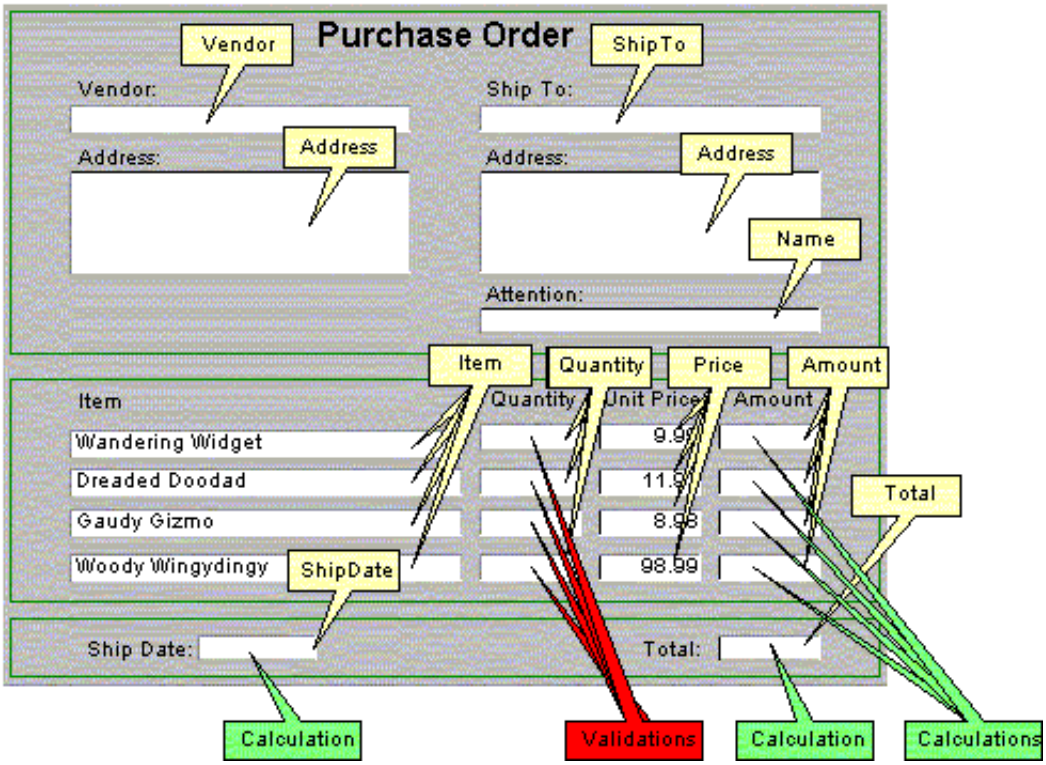
The `calculate` and `validate` elements enclose scripting to derive a value and return it to the processing application. Any scripting that is invoked by these elements must not attempt to alter the state of the form object model in any way. Not all scripting language implementations or processing applications may be able to enforce this restriction, so form templates should adhere to this restriction if they are designed to be interoperable.

The `calculate` and `value` elements are related in that each of them can be used to set an initial value.

Scripts in `calculate` and `validate` elements are interpreted as expressions. The value of such an expression is returned to the processing application. For scripting languages that cannot be interpreted as an expression, the binding of the scripting language to the XFA object model may include some facility for explicitly returning a value.

Calculate and validate scripts are not passed any parameters from the processing application.

The following form shows a simple purchase order application, and illustrates how calculations and validations might be used on such a form.



FORM-2 — A form with calculations

Down-pointing call-outs indicate all the field names on this form. In the tabular area of the form are four fields called `Item`, four fields called `Quantity`, four fields called `UnitPrice`, and four fields called `Amount`. Green up-pointing call-outs indicate fields with embedded calculations, and the red up-pointing call-outs indicate fields with embedded validations.

A subset of the XML used to defined this purchase order form might be as follows.

Example 10.1 Purchase order form with calculations and validations

```
<template …>
   <pageSet …>…</pageSet>
   <subform name="FORM-1" layout="tb">
      <subform name="Detail">
         <occur min="3" max="3"/>
         <field name="Item" …>…</field>
         <field name="Quantity" w="1" h="12pt">
            <validate>
               <script>within($, 0, 19)</script>
            </validate>
         </field>
         <field name="UnitPrice" …>
            <validate>
               <script>$ >= 0</script>
            </validate>
         </field>
         <field name="Amount" …>
            <calculate>
               <script>Quantity * UnitPrice</script>
            </calculate>
         </field>
      </subform>
      <subform name="Summary">
         <field name="ShipDate" …>
            <calculate>
               <script>Num2Date(date() + 2, DateFmt())</script>
            </calculate>
         </field>
         <field name="Total" …>
            <calculate>
               <script>Str(Sum(Amount[*]), 10, 2)</script>
            </calculate>
         </field>
      </subform>
   </subform>
</template>
```

An explanation of the FormCalc expressions used in this sample is contained in the "FormCalc Specification" on page 891. The `Quantity` field is validated to ensure that it is no less than zero and no more than 19. The `UnitPrice` field is validated to ensure it is not negative. The `Amount` field is calculated from the other fields in the same `Detail` subform. The `ShipDate` field is calculated as the current date plus two days. The `Total` field is calculated as the sum of the `Amount` fields from all three instances of the `Detail` subform, formatted as ten digits with two after the radix point.

The scripts in calculations, validations and events may specify whether the script is to be executed at the client, the server, or both. See "Specifying Where to Execute a Script" on page 354.

Document Variables

Document variables may be used to hold boilerplate or image references that can be inserted conditionally under control of a script or they may be used to define script object references.

Variables Used to Hold Boilerplate or Image References

For example, the terms and conditions of a purchase agreement can vary depending on some piece of information entered in the form. Placing the boilerplate or image references into a `variables` element makes it accessible to scripts via the usual mechanism of SOM expressions.

The variables element can hold any number of separate data items. The data items can be any kind of data. Each data item bears its own name attribute so they are individually addressable by scripts. In SOM expressions, data items are directly under the subform. The `variables` object is un-named, as described in "Variables Objects Are Always Transparent" on page 82. As a result, `variables` objects are treated as a transparent element in resolving SOM references.

Document variables may be read from or written to. That is, a document variable may be changed by the form, perhaps as the result of an event script.

Example 10.2 results in an interface such as the one shown at right. If the person filling in the form selects the button titled "Cat Lover," a picture of a cat appears in the field above the selection, or if he/she selects the button titled "Dog Lover," a picture of a dog appears. The value of the picture field is a calculation that resolves to one of the variable children, depending on the values of the exclusion group that contains the buttons. One of the variables is a URL that references a cat image, and the other is a URL that references a dog image.

Example 10.2 Button selection equates field with image variable

```
subform name="x">
   <variables>
      <image name="DocPicture">…</image>
      <image name="CatPicture">…</image>
   </variables>
   <field name="PersonalizedPicture">
      <imageEdit/>
      <calculate>
         <script>if (CatOrDogLover==1) then CatPicture
            else DocPicture endif;</script>
      </calculate>
   </field>
   <exclGroup name="CatOrDogLover" … >
      <field … >
         <ui> <checkButton shape="round">… </checkButton> </ui>
         <caption … >
            <value>
               <text>Cat Lover</text>
            </value>
         </caption>
         <items>
            <integer>1</integer>
         </items>
```

```
      </field>
      <field y="9.525mm" w="60.325mm" h="9.525mm">
        <ui> <checkButton shape="round">… </checkButton> </ui>
        <caption … >
          <value>
            <text>Dog Lover</text>
          </value>
        </caption>
        <items>
          <integer>2</integer>
        </items>
      </field>
    </exclGroup>
</subform>
```

It is conventional to place a single `variables` element in the root subform to hold all document variables, but this is only a convention. Any subform can hold a variable element.

Variables Used to Define Named Script Objects

XFA allows variables (`variables`) to define named script objects. The properties and methods of such objects may be referenced using the familiar class expression, `object.property` or `object.method()`.

Instantiation of Named Script Objects

Subforms are instantiated during data binding. At the same time, the variables contained in each subform are instantiated. If multiple occurrences of the same subform are instantiated, then each instance has its individual set of variables.

When a script document variable is instantiated, the contents of the element are compiled into a script object and that object is then registered with the subform. From that point, when the document variable is referenced the compiled script object is returned.

Declaring and Referencing Named Script Objects

Note: Scripts can only reference script document variables if they have the same contentType.

Example 10.3 Declaring and using a named script object

```
<template>
  <subform name="form1">
    <variables>
      <script name="foo" contentType="application/x-javascript">
        var x = 0;
        var y = 0;
        var factor = 1;
        function sum(val1,val2)
        {
          var sum = (val1 + val2) * this.factor;
          return (sum);
        }
      </script>
    </variables>
```

```
      <field name="f1">
      <!-- this example assumes naked references are available -->
        <calculate>
          <script contentType="application/x-javascript">
            foo.x = 2;
            foo.y = 2;
            foo.factor = 2;
          </script>
        </calculate>
      </field>
      <field name="f2">
      <!-- this example assumes naked references are available -->
        <calculate>
          <script contentType="application/x-javascript">
            foo.sum(foo.x, foo.y);
          </script>
        </calculate>
      </field>
      <field name="f3"> <!-- this example doesn't use naked references -->
        <calculate>
          <script contentType="application/x-javascript">
            xfa.form.form1.foo.x = 4;
            xfa.form.form1.foo.y = 4;
            xfa.form.form1.foo.factor = 4;
          </script>
        </calculate>
      </field>
      <field name="f4"> <!-- this example doesn't use naked references -->
        <calculate>
          <script contentType="application/x-javascript">
            xfa.form.form1.foo.sum(xfa.form.form1.foo.x,
              xfa.form.form1.foo.y);
          </script>
        </calculate>
      </field>
    </subform>
</template>
```

After the XFA processing application completes data binding the above template, it executes the resulting form's calculations. If those calculations are executed in the order f1 to f4, the following results occur:

1. The data binding algorithm instantiates the subform named `form1`, which interns creates an instance of the script object `foo`, by compiling the contents of script element.

2. The calculation on the field named `f1` fires, setting variables on `foo` to `x = 2`, `y = 2` and `factor = 2`.

3. The calculation on the field named `f2` fires, which calls the `sum` function on `foo` ((2+2)* 2), resulting in the field value being set to 8.

4. The calculation on `f3` fires, setting variables on `foo` to `x = 4`, `y = 4` and `factor = 4`.

5. The calculation on `f4` fires, which calls the `sum` function on `foo` ((4+4)* 4), resulting in the field value being set to 32.

Note: The above result sequence assumes the post-binding execution of calculations proceeds in order from `f1` to `f4`. There is no guarantee that this will be the execution order.

Calculations

This section explains how the processing application supports the `calculate` element represented in the Form DOM. It describes how the `calculate` element relates to other automation elements, when the processing application activates calculation scripts, where it stores the result of the calculation, and how it observes precedence in interconnected calculations.

The "Template Specification" describes the syntax of the `calculate` element.

About

The `calculate` element is one of the family of automation elements. The other automation elements are `event` and `validate`. Automation elements are procedural extensions to the XFA architecture.

The `calculate` element provides a means of calculating the value of a container element, with the calculation being represented as a script. The parameters in such a script may include the values of other container objects. The XFA processing application is responsible for updating the value of the container element with the value returned from the calculate script, although this obligation does not apply to subform `calculate` elements.

The `calculate` element can be a child of the container elements: `exclGroup`, `field`, and `subform`. It specifies a script to use for calculating the value of its parent container, and it specifies override conditions. Such conditions specify whether a processing application can allow a user to override a calculated value, and if so, what types of warnings should be issued.

Activation

This section describes the stimuli that cause the processing application to activate `calculate` elements. Many of those stimuli also trigger the processing application to activate the other automation elements, `event` and `validate`. In cases where a single stimuli triggers multiple automation elements, the order of activation is as follows: (1) `event` elements, (2) `calculate` elements, and (3) `validate` elements. (See "Order of Precedence for Automation Objects Activated by the Same Trigger" on page 348.)

The processing application activates `calculate` elements when the value of the field, subform, or exclusion group changes. Those values can change as a result of any of the following actions:

- *Data-binding*. As a final phase of data-binding, the processing application activates all `calculate` elements. It also re-activates `calculate` elements, as described in *Cascading value changes*.

  During the initial data-binding (data merge), the only data present in the Form DOM are default values supplied by the Template DOM. During subsequent data-bindings (data re-merges), the values from calculation scripts reflect the current data in the Data DOM and, where needed, default field values from the Template DOM.

- *Interactive data entry*. A processing application allows users to enter data, without repeating the data-binding process. Such entries simultaneously change values in the form and Data DOM. When a user enters data, the processing application activates `calculate` elements that are dependent on that container's value. It may also re-activate other `calculate` elements, as described in *Cascading value changes*.

- *Cascading value changes*. In some cases, multiple `calculate` elements may depend on one another, in a *cascading* relationship. In other words, a change to the value of one `field` can influence the calculated values of many others. In such cascading calculations, the processing application re-activates `calculate` elements, as the values on which they depend change.

  References to named script objects are not considered in determining dependencies. That is, if a calculation includes a reference to a property or method of a named script object and the properties of that object are changed by another calculation, the calculation under consideration is not re-activated ("Variables Used to Define Named Script Objects" on page 326). In other words dependencies due to self-modifying or mutually-modifying scripts are not detected.

  If the calculation of an element references its own value, either directly or indirectly, a circular reference is said to exist. The following points address responsibilities related to circular references:

  - *XFA form creators*. It is the responsibility of XFA form creators to prevent circular references from being specified in calculate scripts. Such checks should be done concurrently with form creation, rather than through the addition of validation scripts.
  - *Processing application*. It is recommended that the processing application provide some means of identifying and terminating the execution of seemingly infinite loops.

  Note: Scripts do not manage calculation dependencies; rather, the processing application is responsible for managing calculation dependency on behalf of the form. (See "Scripting" on page 353.)

Result

The processing application uses the result of executing the calculation script as described below:

Parent element	Result destination in the parent element
`field`	Replaces the value of the `field` container element.
`exclGroup`	Replaces the value of the `exclGroup` container element. This action has the side effect of changing the state and value of the fields contained in the exclusion group.
`subform`	`subform` elements do not have explicit values; however the result of a subform `calculate` script can be used to initiate some other function unrelated to setting a value.

Validations

This section describes the nature of validation, what types of tests are included in validation, when validation is done, and how an interactive XFA processing application interacts with a user when validation fails.

Validation allows a template designer to specify a set of tests to be performed on a field, subform, or exclusion group. As with calculations, validation tests are triggered by changes to the field, subform, or exclusion group.

About

As compared to UI validation

In an interactive context, the UI may perform some validation. For example, a numeric edit widget will not accept letters of the alphabet as input. However this type of validation does not apply to non-interactive applications, because they have no UI. Furthermore, this type of validation is quite limited. It cannot, for example, compare the numeric content of two fields to validate that one is larger than the other. Validation scripts provide a mechanism to perform validations that are more intelligent and that, optionally, apply in non-interactive as well as interactive contexts.

As compared to XML validation

XFA validation differs from XML validation, in the following ways:

- *Type of testing*. While some XFA validation tests have counterparts in XML validation, XFA validation also supports scripted tests. Such tests support highly specific validation for containers that can consider the values of other fields.
- *Activation of tests*. Activation of XFA validation for specific container may be independent of activation for validation in other containers and may be triggered at various stages in the life of the a form. For example, XFA validation may be done at all of the following stages: when the focus leaves a field (after data is entered), when a button is clicked, and when the form is committed.

  In contrast, XML validation is an all-or-none endeavor and would be performed just before committing the form. Unfortunately, such errors occur too late in the form's life for a user to respond. Consider a user's response to being pelted with numerous validation error messages, when attempting to commit (submit) a form with numerous inter-related fields.

Types of Validation Tests

Validation provides up to four types of tests. The following table describes those test types, the order in which they are executed, and their relevance to the container element. All but the data-type test are specified in the container element's `validate` element.

Exe-cution order	Test type	Container element		
		Field	Subform	Exclusion group
1	Null-content test (`nullTest`). Null content is not allowed. Typically, this is a mechanism for ensuring the user enters a value in a particular container object. This test is not applicable if the template provides a default value.	✔	Not applicable	✔
2	Datatype test. The datatype of a field must be consistent with the type of data entered in the field. Unlike the other validation tests, this test is not specified in the `validate` element; rather, the datatype test is implied by the existence of a datatype element (i.e. `integer` and `float`) within the field's `value` element, and the error messages it generates are application-specific.	✔	Not applicable	Not applicable

Execution order	Test type	Container element		
		Field	Subform	Exclusion group
3	Format test (`formatTest`). The format of the value must match the picture clause specified in the `validate` element.	✔	Not applicable	Not applicable
4	Script test (`scriptTest`). The script supplied in the `validate` element must return a true value for a script test to succeed.	✔	✔	✔

For any field, subform, or exclusion group. All validation tests specified for a field, subform, or exclusion group must succeed for the form object's value to be considered valid. If any test fails, the XFA processing application responds, as described in "Responding to Test Failures" on page 333.

As described in the "Localization and Canonicalization" on page 138, the result of any presentation formatting defined for the form object does not alter the value of the form object — it remains unformatted. Therefore, validation tests (`nullTest`, `formatTest`, and `scriptTest`) are performed against the unformatted value.

Activation

Validate elements can be activated multiple times during the life of a form.

Initialization

When an XFA-processing application initializes a form, it executes all validation tests specified for the form.

Interactive

An interactive XFA application performs the tests in a validation element upon exit from the field or subform, provided the user has entered a value of the field or subform. The application is not required to perform the validation tests if the value of the container object is unchanged.

An interactive XFA application also performs the tests in all validation elements when trying to commit the form. A form is said to be committed when it is in a final state, such as when it is submitted to a server.

Non-Interactive

An XFA application may perform the tests in validation elements after the data binding (merge) operation completes. This is optional because there is no point to a validation complaining about a field being empty when the partly-blank form is only going to be printed on paper so that blank fields can be filled in with pen and ink. The same situation occurs when a partly-blank form is going to be rendered into an interchange format (such as HTML) to send to a non-XFA client for filling in.

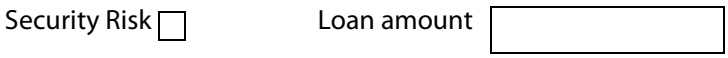
See "Order of Precedence for Automation Objects Activated by the Same Trigger" on page 348.

User Interactions With Validation Tests

Error and Warning Messages

The form designer uses the child message element to provide an optional warning message for each type of validation (null test, format test and script test). The processing application presents the appropriate message to the user if the value fails any of the validations tests. If no such message is configured for a particular validation test, the application provides its own. Note that there are attribute values that suppress the presentation of the warning message.

The name of each `text` element within a `message` element specifies the validation test to which it applies, as shown in the bold-faced lines in the following example. The example contains a format test that verifies the amount entered for "Loan amount" is a number between 0 and 999999 and a script test that limits the amount entered to 1000 if the "Security Risk" box is checked.



Example 10.4

```
<field name="LoanAmount" …>
   …
   <value>
      <integer/>
   </value>
   …
   <bind>
      <picture>zzzzz9</picture>
   </bind>
   <validate formatTest="error" scriptTest="error">
      <message>
         <text name="formatTest">
            You must enter a number between 0 and 999999.
         </text>
         <text name="scriptTest">
            You are a security risk, so we cannot lend you more than $1000.
         </text>
      </message>
      <picture>zzzzz9</picture>
      <script>
         (not SecurityRisk) | (SecurityRisk &amp; (LoanAmount &lt; 1000))
      </script>
   </validate>
</field>
<field name="SecurityRisk" y="30.00mm" x="13.49mm" w="12.44mm" h="9.26mm">
   <value>
      <integer>1</integer>
   </value>
   …
</field>
```

Interacting with the User to Obtain a Valid Value

Though a value just entered may be invalid, interactive processing applications are recommended *not* to force the user to remain in the current form object until the validation constraints are satisfied. Complex forms often contain validations that compare values across a number of form objects. Disallowing the user from navigating out of the currently active form object may make it impossible for the user to satisfy the validation of the current form object by altering one or more other values on the form.

The processing application may choose to prevent the form from being committed if any part of the form is invalid. For example, a processing application may choose to prevent the submission or saving of a form until it is considered valid.

Responding to Test Failures

This section explains how the processing application responds to errors in the validation tests applied to a field, subform, or exclusion group. For most tests the validate element's attributes specify how the XFA processing application should respond in the event of an error. For example, the following fragment specifies that the data will not be accepted if it fails a formatting validation (that is, a validation against a picture clause).

Example 10.5 Fragment specifying a formatting test that data must pass

```
<validate formatTest="error">
   <picture>…</picture>
</validate>
```

In addition there is a datatype test for which the error response is implied, as described in "The datatype test" on page 335.

The following sections describe the attributes of the validate element that control error response levels.

The nullTest attribute

The `nullTest` attribute on the `validate` element has three potential values that determine how this validation test is applied to the form:

`disabled`

The form object is permitted to have a value of null; that is, the field can be left without a value, and it will not negatively impact the validity of the form. This attribute value disables this validation test.

`warning`

The form object is recommended to have a non-null value. If the user does not supply any value for the form object or explicitly sets the value to null, the processing application will present the warning message. The message must inform the user that the form object is recommended to have a value, and provide two choices:

dismiss — The user understands the form's recommendation and wishes to return to the form, so that s/he may satisfy this constraint.

override — The user understands the form's recommendation, but has chosen to contravene this constraint.

error

The form object is required to have a non-null value. Failure to provide a non-null value shall constitute an error. The processing will present an error message, and the form object considered invalid. XFA application may skip the remaining validations for the field or exclusion group.

The formatTest attribute

The `formatTest` attribute on the `validate` element has three potential values that determine how this validation test is applied to the form:

disabled

the form object is permitted to have a value that does not conform to the input mask; that is, the field can be left with a non-conformant value, and it will not negatively impact the validity of the form. This attribute value disables this validation test.

warning

The form object is recommended to have a value that conforms to the input mask. If the user does not supply such a value, the processing application will present the warning message. The message must inform the user that the form object is recommended to have a value that conforms to the input mask, and provide two choices:

dismiss — The user understands the form's recommendation and wishes to return to the form, so that s/he may satisfy this constraint.

override — The user understands the form's recommendation, but has chosen to contravene this constraint.

error

The form object is required to have a value that conforms to an input mask. Failure to provide such a value shall constitute an error. The processing will present an error message, and the form object is considered invalid. XFA application may skip the remaining validations for the field.

The scriptTest attribute

Scripts specified as part of a validation should make no assumptions as to how the processing application might use the validation results, or when the `validate` element is invoked. In particular, the script should not attempt to provide feedback to a user or alter the state of the form in any way.

The scriptTest attribute on the `validate` element has three potential values that determine how this validation test is applied to the form:

disabled

The form object is permitted to have a value that does not conform to the script; that is, the field can be left with a non-conformant value, and it will not negatively impact the validity of the form. This attribute value disables this validation test.

warn

The form object is recommended to have a value that conforms to the script. If the user does not supply such a value, the processing application will present the warning message. The message must inform the user that the form object is recommended to have a value that conforms to the script's constraints, and provide two choices:

dismiss — The user understands the form's recommendation and wishes to return to the form, so that s/he may satisfy this constraint.

override — The user understands the form's recommendation, but has chosen to contravene this constraint.

error

The form object is required to have a value that conforms to the script. Failure to provide such a value shall constitute an error. The processing will present an error message, and the form object is considered invalid.

The datatype test

This test is implied by the existence of a restrictive datatype element (for example `integer` or `float`) within the field's `value` element. The error messages generated when the datatype test fails are application-specific, and the error handling is equivalent to a level of `error`. That is, if the datatype test fails, the form object is considered invalid.

Events

In XFA templates, scripts may be associated with particular *events*. An event is a particular change of state in the form. When the particular change of state happens, the actions associated with the event are automatically invoked. Those actions may be any of the following:

- Script. A script property specifies a set of scripted instructions that can perform a variety of actions, such as transforming the data, changing the presentation of the data, or triggering other events. See "Scripting" on page 353.
- Execute. An execute property invokes a WSDL web service. Such a service can be used to initate a complex interaction with a server. See "Using Web Services" on page 382.
- Submit. A submit property invokes an HTTP protocol to send all or part of the form to a server, and in some cases to accept new data provided by that server. See "Submitting Data and Other Form Content to a Server" on page 375.
- Sign. A signData property causes a signature handler to create an XML digital signature, as specified in the signData property. See "XML Digital Signatures" on page 471.

The object whose change of state triggers the event is called the *target*. There are six general classes of events, distinguished by the type of target. Some events in different classes share the same name because they are similar in function, however they are distinct events because an event is distinguished by both name and target. In addition calculations and validations are very much like events and can be treated as special types of events.

This section describes the types of events assigned to each class of event.

Application Events

Application events are triggered by actions of the XFA application. Because application events are not directly linked to user actions, they are triggered in both interactive and non-interactive contexts. The script in an application event can reference the event object using the SOM expression `xfa.host`, or the alias `$host`, as described in "Internal Properties and Methods" on page 84

The application events are as follows:

docClose

This event fires at the very end of processing if and only if all validations succeeded. Success in this case is defined as generating nothing worse than a warning (no errors). Note that this event comes too late to modify the saved document; it is intended to be used for generating an exit status or completion message.

docReady

This event fires before the document is rendered but after data binding. It comes after the `ready` event associated with the Form DOM.

postPrint

This event fires just after the rendered form has been sent to the printer, spooler, or output destination.

prePrint

This event fires just before rendering for print begins.

postSave

This event fires just after the form has been written out in PDF or XDP format. It does not occur when the Data DOM or some other subset of the form is exported to XDP.

preSave

This event fires just before the form data is written out in PDF or XDP format. It does not occur when the Data DOM or some other subset of the form is exported to XDP. XSLT postprocessing, if enabled, takes place after this event.

DOM Events

DOM events trigger when a DOM changes state. Because they are not directly linked to user actions, they are triggered in both interactive and non-interactive contexts.

A script binds to a DOM event by expressing a `ref` property whose value is a SOM expression pointing to the DOM. For example, the value `xfa.form` (or its alias `$form`) binds to the Form DOM.

The following DOM events are defined:

ready

The ready event fires after an XFA DOM has finished loading. This event applies to the Form DOM and the Layout DOM. It does not apply to the Template DOM or Data DOM primarily because it would be difficult for an XFA application to ensure that the scripts were loaded and bound to the events before the events fired.

In the case of the Form DOM (`$form`) it fires after the Template and Data DOMs have been merged to create the Form DOM, and the calculations and validations have fired. In addition, the `ready` event fires when the current data record advances. See "Exchanging Data Between an External Application and a Basic XFA Form" on page 108 for more information about processing data as records.

In the case of the Layout DOM (`$layout`), the `ready` event fires when the layout is complete but rendering has not yet begun. Thus a script can modify the layout before it is rendered.

preSubmit

This event triggers whenever form data is submitted to the host via the HTTP protocol, just after the data has been marshalled in the Connection Data DOM but before the data is submitted to the host. A script triggered by this event has the chance to examine and alter the data before it is submitted. If the script is marked to be run only at the server, the data is sent to the server with an indication that it should run the associated script before performing the rest of the processing.

The `preSubmit` event applies only to the Form DOM (`$form`). Note that `preSubmit` does not distinguish between submissions initiated by different button pushes or to different URLs. Any script that needs to make these distinctions must include code to find out what button was pushed. In general `preSubmit` is analogous to `preSave` and serves a similar purpose.

For example, consider the following template fragment.

Example 10.6 Fragment using preSubmit event

```
<subform name="root">
   <subform name="sub1">
      <field name="field1" … >
         <event ref="$" activity="click">
            <submit url="http://www.example.com/t1/?abcd" … />
         </event>
      </field>
      <field name="field2" … >
         <event ref="$" activity="click">
            <submit url="http://www.example.com/y78/" … />
         </event>
      </field>
   </subform>
   <event ref="$form" activity="preSubmit">
      <script>
         if ($event.target.name == "field1") then …
      </script>
   </event>
</subform>
```

In this example the `click` events from either of two fields initiate the submission of form data to a host. There is a script associated with the Form DOM's `preSubmit` event, so when either field is clicked, the outgoing data is marshalled, the `preSubmit` script runs, then the submit transaction takes place. The preSubmit script uses the `$event` object to find out which `click` event triggered it. The `$event` object is described below in the section "Properties" on page 343.

Subform Events

Subform events trigger in response to changes of state which affect subforms. Some are generated in interactive contexts and some in both interactive and non-interactive contexts.

A script binds to a subform event by expressing a `ref` property whose value is a SOM expression pointing to the subform.

The subform events are as follows:

enter

This event triggers when some field directly or indirectly within the subform gains keyboard focus, whether caused by a user action (tabbing into the field or clicking on it with the mouse) or by a

script programmatically setting the focus. It is not triggered by keyboard focus moving to another field within the same subform – focus must come in from outside the subform.

exit

This event triggers when keyboard focus is yielded from a field directly or indirectly within the subform to a field or other object outside the subform. It is not triggered by keyboard focus moving to another field within the same subform – focus must go out from inside the subform.

initialize

This event triggers after data binding is complete. A separate event is generated for each instance of the subform in the Form DOM.

Exclusion Group Events

Exclusion Group events trigger in response to user actions which affect exclusion groups. Some are generated in interactive contexts and some in both interactive and non-interactive contexts.

A script binds to an exclusion group event by expressing a `ref` property whose value is a SOM expression pointing to the exclusion group.

The exclusion group events are as follows:

enter

This event triggers when some field within the exclusion group gains keyboard focus, whether caused by a user action (tabbing into the field or clicking on it with the mouse) or by a script programmatically setting the focus. It is not triggered by keyboard focus moving to another field within the same exclusion group – focus must come in from outside the exclusion group.

exit

This event triggers when keyboard focus is yielded from a field within the exclusion group to a field or other object outside the exclusion group. It is not triggered by keyboard focus moving to another field within the same exclusion group – focus must go out from inside the exclusion group.

initialize

This event triggers after data binding is complete. A separate event is generated for each instance of the exclusion group in the Form DOM.

Field Events

Field events trigger in response to user actions which affect a field. Some are generated in interactive contexts and some in both interactive and non-interactive contexts.

A script binds to a field event by expressing a `ref` property whose value is a SOM expression pointing to the field.

The field events are as follows:

change

This event triggers when the content of the field is changed by the user. This event triggers on every keystroke as long as the field has keyboard focus. It also triggers when the user pastes into the field, makes a selection from a choice list or drop-down menu, checks or unchecks a checkbox,

or changes the setting of a set of radio buttons. It is *not* triggered by content changes that are made by the XFA application, for example calculations, nor is it triggered by a merge operation.

click

This event triggers when a mouse click occurs within the region.

enter

This event triggers when the field gains keyboard focus, whether caused by a user action (tabbing into the field or clicking on it with the mouse) or by a script programmatically setting the focus. It also triggers when a new selection is made in a choice list, but this behavior can be inhibited. See below.

exit

This event triggers when the field loses keyboard focus. It also triggers whenever a new selection is made in a choice list (followed by an enter event), but this behaviour can be inhitibited. See below.

full

This event triggers when the user has entered the maximum allowed amount of content into the field and tries to enter more content.

initialize

This event triggers after data binding is complete. A separate event is generated for each instance of the field in the Form DOM.

mouseDown

This event triggers when the mouse button is depressed at a moment when the mouse pointer is within the region.

mouseEnter

This event triggers when the user moves the mouse pointer into the region of the field, without necessarily pressing the mouse button. It is not triggered when the mouse pointer moves into the field for some other reason, for example because an overlying window closes.

mouseExit

This event triggers when user moves the mouse pointer out of the field, whether the mouse button is depressed or not. It is not triggered when the mouse pointer moves out of the field for some other reason, for example because an overlying window opens.

mouseUp

This event triggers when the mouse button is released at a moment when the mouse pointer is within the region.

preOpen

This event applies only to drop-down choice lists, or more specifically choice lists for which `open="userControl"` or `open="onEntry"`. This event is intended to house scripts that add choices to the choice list. It is especially useful when the choice list is infrequently used and its choices take a while to load.

The form object model method `addItem()` is particularly useful in scripts triggered by `preOpen` events. `addItem()` is described in the *Adobe XML Scripting Object Reference* [FOM].

It is triggered under the following circumstances:

- User clicks on the symbol that causes the choice list to drop down. This symbol is usually a down arrow.
- While the choice list is in focus, the user presses the keyboard sequence that causes the choice list to drop-down. In this situation, the choice list gains focus through some mechanism other than clicking, such as tab-order traversal or clicking the associated text box.
- Any script calls the `$host.openList()` method with a parameter pointing to the field which contains the choice list object. This causes the `preOpen` event to be triggered and also causes the choice list to gain focus. `openList()` is described in the *Adobe XML Scripting Object Reference* [FOM].

This event does not trigger in response to the user pressing the Enter key while the combo box has the focus.

It is recommended that users be provided with some sort of feedback mechanism before the script is executed. The script might block interaction for a time, which would be an unexpected characteristic of choice lists.

Starting with XFA 2.4 the issuing of events in a choice list field was modified. In versions of XFA prior to 2.4 an exit event was generated whenever a selection was clicked in the choice list, but an enter event was only generated on the first click, as long as the choice list held focus between clicks. In consequence when the user changed the selection without first transferring focus outside the choice list, consecutive exit events were generated without an enter in between. In XFA 2.4 this behavior was corrected. When the user commits a selection by clicking on it or pressing Enter an exit event is generated, even though focus remains on the choice list. If the user subsequently clicks on a selection or presses enter another enter event is generated, followed by an exit event. However each enter event that would not have been generated prior to XFA 2.4 is identified as such by its `reenter` property as described in "reenter" on page 347.

Normally the XFA processor is required to display the new choice list behavior whenever the XFA schema specified by the template is 2.4 or above. However to facilitate editing and updating older templates, without necessarily having to rewrite the template's scripts, a processing instruction is provided which forces the XFA processor to follow the older behavior even though the template references a newer schema. An example of this processing instruction follows.

Example 10.7 Template with originalXFAVersion processing instruction

```
<template xmlns="http://www.xfa.org/schema/xfa-template/2.5/">
   <? originalXFAVersion http://www.xfa.org/schema/xfa-template/2.1/
      LegacyEventModel:1 ?>
   …
</template>
```

The first parameter in the `originalXFAVersion` processing instruction is an XFA namespace URI. When this indicates version 2.4 or later, meaning the template started life in XFA version 2.4 or greater, the processing instruction is ignored and the XFA 2.4 behavior is always followed. Otherwise any remaining parameters are inspected. If there is an additional parameter with the value `LegacyEventModel:1` the pre-2.4 behavior is followed. Otherwise the XFA 2.4 behavior is followed.

In the example the XFA schema in use by the template is 2.5 but there is an `originalXFAVersion` processing instruction, the original XFA version is 2.1, and there is a `LegacyEventModel:1` parameter so the pre-2.4 behavior is followed.

Connection Events

Connection events trigger in response to activity in a link between the XFA processor, acting as a client, and some external processor providing a web service. Because connection events are not directly linked to user actions, they are triggered in both interactive and non-interactive contexts.

An script binds to a connection event by expressing a `ref` property whose value is a SOM expression identifying the connection.

The connection events are as follows:

`postExecute`

This event triggers when data is sent to a web service via WSDL, just after the reply to the request has been received and the received data is marshalled in the Connection Data DOM. A script triggered by this event has the chance to examine and process the received data. After execution of this event the received data is deleted.

`preExecute`

This event triggers when a request is sent to a web service via WSDL, just after the data has been marshalled in the Connection Data DOM but before the request has been sent. A script triggered by this event has the chance to examine and alter the data before the request is sent. If the script is marked to be run only at the server, the data is sent to the server with an indication that it should run the associated script before performing the rest of the processing.

For example, consider the following template fragment and accompanying connection set.

Example 10.8 Template fragment declaring a postExecute script

```
<subform name="root">
   <subform name="sub1">
      <field name="field1" … >
         <event ref="$" activity="click">
            <execute connection="service1" … >
         </event>
      </field>
      <field name="field2" … >
         <event ref="$" activity="click">
            <execute connection="service1" … >
         </event>
      </field>
   </subform>
   <event ref="$connectionSet.service1"
      activity="postExecute">
      <script>… </script>
   </event>
</subform>
```

Example 10.9 Connection set accompanying Example 10.8

```
<connectionSet … >
  <wsdlConnection name="service1" … >
    …
  </wsdlConnection>
</connectionSet>
```

In this example the `click` events from either of two fields initiate a web service transaction. There is a script associated with the connection's `postExecute` event, so when either field is clicked, the outgoing data is marshalled, the web server transaction takes place, the resulting incoming data is marshalled, then the `postExecute` script runs.

Instance Manager Events

An instance manager is an object that manages a dynamic array of same-named sibling objects. There is one instance manager for each such array. The instance manager fires an event to inform an object in the array that some change has happened to its position in the array.

`indexChange`

This event fires to tell an object that it has just been added to an array or that its position in the array (its subscript) has changed. The object's position can change because another object was added in front of it or because an object was removed in front of it.

When field objects are created and added to an array by data binding, the `initialize` event fires before the `indexChange` events.

All of the objects managed by a particular instance manager share the same declaration in the template and therefore the same script. However there is a property `instanceIndex` which holds the zero-based index of the current object. This can be used by the script to take different actions in different instances. For example, in the following fragment the color of the cell is dependent on the position of the cell and is recalculated whenever the position of the cell changes.

Example 10.10 Fragment using the instance manager to obtain its own index

```
<field name="itemValue"/>
   <event activity="indexChange">
      <script>
         // Whenever a cell is added or moved re-color the cell.
         var index = this.instanceIndex;
         if (index % 5)
            this.color.value = "225,0,0";
         else
            this.color.value = "0,255,0";
      </script>
   </event>
</field>
```

As explained in "Activation" on page 328, the XFA processor performs automatic dependency tracking to ensure that calculations are recomputed for all affected objects whenever necessary. Recalculation is performed whenever an instance is added, deleted, or moved for all calculations that use objects in the affected array. For example, the `Total` field in the following fragment has a dependency upon the `Detail` subform. Hence the XFA processor recalculates the value of the `Total` field whenever an instance of the `Detail` subform is added, deleted, or moved.

Example 10.11 Fragment that recalculates when the instance manager adds an instance

```
<subform name="Detail">
   <occur min="0" max="10"/>
   <field name="ItemValue" …>…</field>
   …
</subform>
<field name="Total" …>
```

```
   <calculate>
      Sum(Detail[*].ItemValue);
   </calculate>
   …
</field>
<field name="AddDetail" …>
   <event activity="click">
      <ui><button/></ui>
      <caption><value><text>Add a detail record</text></value></caption>
      <script>
         _Detail.addInstance()
      </script>
   <event>
</field>
```

Properties

There are two general categories of events, *primary events* which correspond to a user action and *secondary events* which are triggered by an internal change of state. Primary events update some or all of the *primary properties* of an object called `xfa.event` (more commonly known by the alias `$event`). By contrast, secondary events do not update the primary properties of `$event`. If a secondary event results from a primary event then the primary properties of `$event` during the secondary event have the values set by the primary event. However, secondary events may set *secondary properties* of `$event`.

It is an error for a script to try to use a property that is not set in the current context. The following table shows which properties are set by each primary event:

Target Type	Event Name	Sets Primary Properties
subform	enter exit	name target
field	click enter mouseEnter mouseExit mouseDown mouseUp	name modifier reenter shift target
field	exit	commitKey name modifier shift target

field	change	change keyDown fullText modifier name newContentType newT prevContentType prevText selEnd selStart shift start target
field	full	change fullText newContentType newText prevContentType prevText target

In addition, the following secondary events set secondary properties but don't change any primary properties:

Target Type	Event Name	Sets Secondary Properties
connection	preExecute postExecute	soapFaultCode soapFaultString

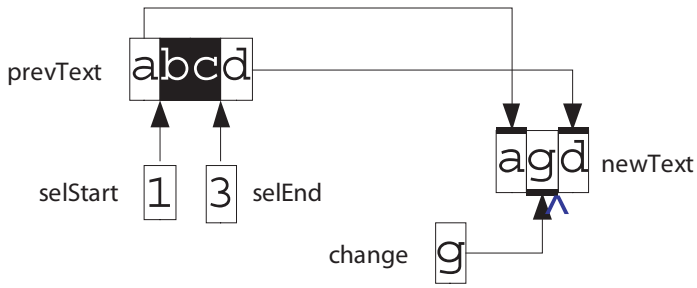
In addition, a script may conjure up an event from thin air using the $event object. Only the name and target properties are initialized when an event is created this way. It is up to the script that creates the event to assign values to the remaining relevant properties of $event before triggering the event.

Most properties of $event are read-only. The exceptions are the selStart, selEnd and change properties which are writable as well as readable.

selStart, selEnd, prevText, newText, and change are used together. The change property holds the text which is inserted into the field. Assigning to the change property replaces the typed or pasted characters with the characters from the assigned string. prevText holds the contents of the field as it was prior to the event. newText is read-only and yields the text that is placed into the field after the script has terminated. The value of newText changes in response to changes in the values of change, selStart, and selEnd. selStart and selEnd control which characters in the prevText is replaced by the characters in change. Like a UI text edit cursor, selStart and selEnd do not point to characters but to the boundaries between characters. A value of zero points in front of the first character in the field, one points in between the first and second characters, two in between the second and third characters, and so

on. When a change or full event occurs, if characters were selected in the field, selStart and selEnd are set to bracket the selected characters. By contrast, if no characters were selected, selStart and selEnd are both set to the text entry cursor position. selStart is always smaller than or equal to selEnd. Changing selEnd also repositions the text entry cursor in the UI.

For example, suppose the original content of the field was "abcd" (right). The user selected "bc" and then typed "g". When the change event script is invoked, the value of prevText is "abcd", the value of change is "g", the value of selStart is "1", the value of selEnd is "3", and the value of newText is "agd". If the script does not modify any of the change, selStart, or selEnd properties the letter "g" replaces the selected characters "bc" as one would expect. The blue ^ shows the location of the text entry cursor after the operation.

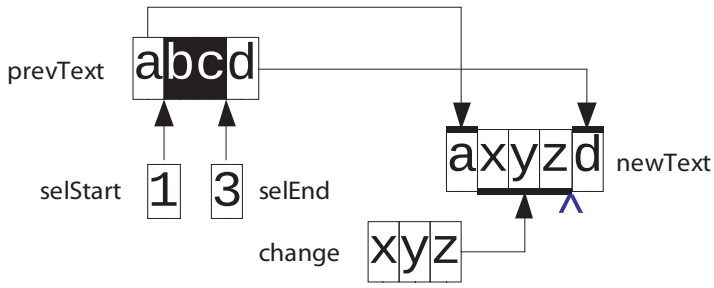


Two characters selected, one inserted

Now suppose that the script assigns the value "xyz" to the change property, as follows:

```
xfa.event.change = "xyz"
```

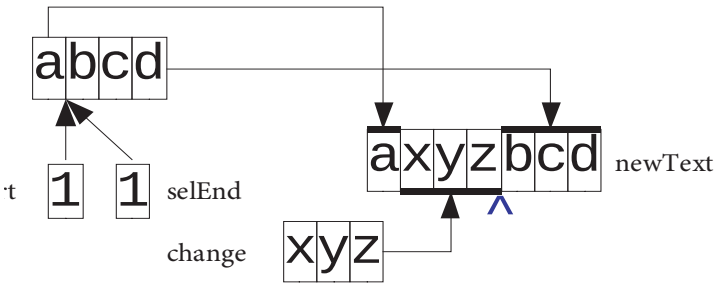
Since selStart and selEnd haven't been changed (yet) the value of newText becomes "axyzd". In other words the string "xyz" will replace the selected characters, as though the user had pasted "xyz" instead of typing "g". This is shown at right.



Two characters selected, three inserted

After this the script changes the value of selEnd from "3" to 1", which happens to equal the value of selStart:

```
xfa.event.selEnd = 1
```



No characters selected, three inserted

Now the value of newText becomes "axyzbcd" and the text cursor in the UI is repositioned between the "a" and the "x". It is as though the user had not selected any text, but positioned the text cursor between the "a" and the "b" and then pasted "xyz". This is shown at left.

$event properties

The individual properties of `$event` are described below:

change

When a `change` or `full` event occurs, this property holds the text that is to be inserted or updated. When referenced this property yields a string. Assigning to this property replaces the typed or pasted text with the value assigned. For example, the following fragment shows a script that converts the entered text to upper case.

Example 10.12 Script updating only changed text

```
<field … >
   …
   <event activity="change">
      <script contentType="application/x-javascript">
         xfa.event.change = xfa.event.change.toUpperCase();
      </script>
   </event>
</field>
```

The boundaries of the window are controlled by the `selStart` and/or `selEnd` properties.

commitKey

Describes what happened to the value in the field when the form field lost focus. The value of this property must be one of:

0

The value was not committed (for example, the escape key was pressed)

1

The value was committed by a click of the mouse outside the field

2

The value was committed by pressing the enter key

3

The value was committed by tabbing to a new field

fullText

If the user pastes into a field, the field may truncate the pasted text. The full (untruncated) value is stored in this property. Content type is indicated by `$event.newContentType`.

keyDown

A Boolean that is true if the arrow key was used to make a selection, otherwise false.

modifier

A boolean that is true if and only if the modifier key (the control or "Ctrl" key on Microsoft Windows) was held down during the event.

name

Name of the current event as a text string.

newContentType

Content type of the new data. The value of this property must be one of:

allowRichText

The field supports rich text. If the content is rich text, it is marked up as described in the "Basic Data Binding to Produce the XFA Form DOM" on page 155.

plainTextOnly

The field does not support rich text. Even if markup is present in the data it should be passed through rather than interpreted. However it is not guaranteed whether or not downstream processing will respond to the markup.

In this version of XFA, the values of the newContentType and prevContentType properties are always the same. It is anticipated that future versions will allow for them to differ.

newText

Content of the field after it changed.

prevContentType

Content type of the data before it changed. The value of this property must be one of:

allowRichText

The field supports rich text. If the content is rich text it is marked up as described in "Basic Data Binding to Produce the XFA Form DOM" on page 155.

plainTextOnly

The field does not support rich text. Even if markup is present in the data it should be passed through rather than interpreted. However it is not guaranteed whether or not downstream processing will respond to the markup.

In this version of XFA, the values of the newContentType and prevContentType properties are always the same. It is anticipated that future versions will allow for them to differ.

prevText

Content of the field before it changed.

reenter

Boolean which is false for a non-choice-list field or for the first enter event generated after a choice list gains focus. It is true for subsequent enter events generated while the same choice list still has uninterrupted focus.

selEnd

Ending position in prevText of the text to be replaced with the value of change. This is a 0-based index into the boundaries between characters. If no text was selected this is set to the position of the text entry cursor at the time the change was made. This property is read-write. Changing the value of this property changes which characters is replaced by the value of change and also repositions the text entry cursor.

selStart

Starting position in the prevText of the change window. This is a 0-based index into the boundaries between characters. If no text was selected this is set to position of the text entry cursor at the time the change was made. This property is read-write. Changing the value of this property changes which characters is replaced by the value of change.

shift

A boolean that is true if and only if the shift key was held down during the event.

soapFaultCode

The fault code returned by the SOAP operation within the `faultcode` child of the `Fault` element, as described in [SOAP1.1]. If no `Fault` element is returned this defaults to the empty string "". This is a secondary property.

soapFaultString

A human-readable string returned by the SOAP operation within the `faultstring` child of the `Fault` element, as described in [SOAP1.1]. If no `Fault` element is returned this defaults to the empty string "". This is a secondary property.

target

The object whose change of state triggered the event. This property is of type `XFANode`.

Order of Precedence When Multiple Actions Are Defined for an Event

A single event may contain any combination of the actions: script, execute, submit, and signData. When the event is triggered, the order in which these actions are satisfied are application-dependent.

Note: This information will be provided in a future release of this specification.

Order of Precedence for Automation Objects Activated by the Same Trigger

One might expect that the template schema would include an element for each event, with each event containing the script to which it is bound. In fact the schema does just the opposite. Scripts are usually located inside the element that declares the object which being modified by the script. Each script identifies the particular event to which it is bound by name and the object which gives rise to the event by SOM expression. This can be thought of as a "come from" notation, in contrast to the more conventional "go to" notation. The advantage of this inverted notation is that a complete object, including all the scripts it requires, can simply be dropped into a template intact. The included scripts plug themselves into the required events using the inverted notation.

One consequence of the above-described notation is that one event can be bound to any number of scripts. Note that when a single event is bound to multiple scripts, the order of execution of the scripts is not defined and the scripts may even run concurrently. For example, given the following template fragment, when the form `ready` event occurs the order of execution of the two scripts is not defined.

Example 10.13 Single event bound to multiple scripts

```
<subform name="outer">
   <subform name="sub1">
      <event activity="ready" ref="$form">
         <script>… </script>
      </event>
   </subform>
   <subform name="sub2">
      <event activity="ready" ref="$form">
         <script>… </script>
      </event>
   </subform>
</subform>
```

Multiple events may be triggered by a single change of state or user action. For example, tabbing from the current field to the next field triggers both the `exit` event for the current field and the `enter` event for the next field. If the current and next fields are in different subforms a total of four events are triggered, namely, `exit` events for the current field and subform and `enter` events for the next subform and field. It is necessary for script authors to know in what order their event scripts being executed. The order of event generation, including calculates and validates, is governed by the following rules.

➤ Rule 1: Enter/exit events, calculations and validations

This section describes the order in which an XFA processing application executes `enter` and `exit` events, calculations and validations that are triggered by the same change of state. This section describes a change in focus caused by the users selection and change in focus caused by the user tabbing from one field/subform to another.

When focus moves from one field, exclusion group, or subform to another, validations and `exit` events precede `enter` events

1. When focus leaves a field, exclusion group, or subform, calculation and validation precedes the `exit` event

2. Calculations, validations and `exit` events for nested elements occur in order from inner to outer element

3. Calculations, validations sand `enter` events for nested elements occur in order from outer to inner element

Note that although the order of validations is well-defined, this should not make any difference to the script writer, because validations can not legally make any changes to the DOMs. They are only allowed to inspect values and return true (valid) or false (invalid).

For example, consider the following template fragment.

Example 10.14 Scripts bound to nested enter and exit events

```
<subform name="outer">
   <subform name="X">
      <validate … />
      <event activity="exit" ref="$" … />
      <field name="A">
         <validate … />
         <event activity="exit" ref="$" … />
      </field>
   </subform>
   <subform name="Y">
      <validate … />
      <subform name="enter" ref="$" … />
      <field name="B">
         <validate … />
         <event activity="enter" ref="$" … />
      </field>
   </subform>
</subform>
```

When the user tabs from field A to field B the order of events is:

1. Validation for field A

2. Exit event for field A
3. Validation for subform X
4. Exit event for subform X
5. Enter event for subform Y
6. Enter event for field B

➤ **Rule 2: Full and change events**

For `full` and `change` events triggered by the same change of state the `change` event occurs before the `full` event. For example, consider the following template fragment.

Example 10.15 Field with both full and change events

```
<field name="A">
   <event activity="full" ref="$" … />
   <event activity="change" ref="$" … />
</field>
```

When the user types the last allowed character into field A, the order of events is:

1. `change` event for field A
2. `full` event for field A

➤ **Rule 3: Merge completion**

For calculations, validations, and initialize events triggered by the completion of a merge operation:

- All calculations are done, then all validations, and then all initialize events are triggered
- Calculations are repeated if the values on which they depend change
- The order of validations is not defined
- Initialize events occur in order of depth-first traversal of the Form DOM

It should not matter to the script writer that the order of validations is not defined, because validations can not legally make any changes to the DOMs. They are only allowed to inspect values and return true (valid) or false (invalid).

➤ **Rule 4: Scripts That Trigger (Invoke) Other Events**

A script may cause changes of state that in turn trigger or invoke other events. It may also directly declare an event. Thus, a script triggered by one event can indirectly invoke other events. In such cases the order of execution is implementation-defined and the scripts may even run concurrently.

For example, the following template fragment illustrates a script triggered by one event (form ready) that in turn explicitly triggers another event (enter to field A).

Example 10.16 A script in one field triggers an event in another field

```
<subform name="root">
   <event activity="ready" ref="$form">
      <script contentType="application/x-ecmascript">
         …
         $form.root.A.enter();
```

```
        …
      </script>
    </event>
    <field name="A">
      <event activity="enter" ref="$" … />
    </field>
</subform>
```

The line highlighted in bold causes an `enter` event to be triggered for field A, even though no change of focus occurs. The above expression `$form.root.A.enter()` has the following parts:

- `$form` references the Form DOM. `$form` is described in "Scripting Object Model" on page 73.
- `$form.root.A` references the specific field in the Form DOM.

`$form.root.A.enter()` triggers the enter events contained in the referenced field. For a field, the enter event triggers when the field gains keyboard focus.

In the above example, the order of execution of the field's enter event script and the portion of the form ready script after the highlighted line is undefined.

The following template fragment illustrates a script performing an action (changing keyboard focus) which indirectly triggers another event.

Example 10.17 A script that changes focus to another field, invoking that field's enter event

```
<subform name="root">
   <event activity="ready" ref="$form">
      <script contentType="application/x-ecmascript">
         …
         $host.setFocus("$form.root.A");
         …
      </script>
   </event>
   <field name="A">
      <event activity="enter" ref="$" … />
   </field>
</subform>
```

In an interactive context the line highlighted in bold causes keyboard focus to change, with the side effect of generating an `enter` event for field A (unless field A already had keyboard focus). The above expression `host.setFocus("$form.root.A");` has the following parts:

- `$host` pertains to anything not specifically associated with a DOM. `$host` is described in "Scripting Object Model" on page 73.
- `$host.setFocus("$form.root.A")` invokes a host method that sets the focus to a specific field or subform. The argument passed to this method is the SOM expression for the field in the Form DOM.

As for the previous example, the order of execution of the scripts after this line is undefined.

In a non-interactive context the line highlighted in bold has no effect because there is no keyboard focus to set.

XFA applications are not required to actually run concurrent events concurrently. The application may queue the events and run them sequentially. However, to ensure the same outcome across concurrent and sequential implementations, there is no way to dequeue an event. Once an event is queued it is committed to running to completion.

➤ **Rule 5: submit**

The order of processing submits relative to `click` events is not specified. Hence it is not safe to place a submit and a `click` event script on the same button. Instead, place the script on the `preSubmit` event.

11 Scripting

This chapter describes the role of scripting objects in templates. It describes how scripting langauges are selected and how their environments must be set up. It also describes exception handling

Purpose of Scripting

It is important to understand that scripting is optional. The template author can take advantage of scripting to provide a richer user experience, but all of the features described so far operate without the use of scripts. Script creation is part of the template authoring process.

XFA supports scripting in ECMAScript [ECMAScript], but it also defines its own script language, FormCalc, which is described in "FormCalc Specification" on page 891. Often, the scripts attached to a form are similar to those attached to a spread-sheet. FormCalc has been designed as an expression-oriented language, with simple syntax for dealing with groups of values in aggregate operations.

Both ECMAScript and FormCalc expose the same object model. Scripting almost always works with data values, so these are easily referenced (though you can script against any XFA DOM element present). Indeed, XFA defines a complete Scripting Object Model (XFA-SOM). A key feature of XFA-SOM is that it manages relative references. For example, when defining an invoice detail line the creator of a form sets up fields named unitPrice, quantity and amount. The calculation for amount is simply unitPrice*quantity. The form contains multiple detail records using the same field names, but XFA-SOM automatically manages the scope to select the unitPrice and quantity data that corresponds to the same detail record. It can do this in two ways: by selecting fields that are in the same or related subforms; or by selecting from among multiple fields with the same name within the same subform. For more information about XFA-SOM see "About SOM Expressions" on page 74.

Because of the declarative nature of XFA-Template, the largest use of scripting is for field calculations. A field with such a script typically is protected against data entry, and instead gets its value from an expression involving other fields. A field's calculation automatically fires whenever any field on which it depends changes (those fields may, in turn, also have calculated values dependent on other fields, and so on). See "Calculations" on page 328.

Similar to calculation, a field can have a validation script applied that validates the field's value, possibly against built-in rules, other field values or database look-ups. Validations typically fire before significant user-initiated events (e.g., saving the data). See "Validations" on page 329.

Finally, scripts can be assigned to events, for example, `onEnter`, `onExit`, `onClick`, and so on. See "Events" on page 335.

Specifying Where to Execute a Script

Scripts may include a property (`runAt`) that specifies where the script should be executed. The possible values for this property are as follows:

runAt value	Desciption
`client` (default)	Indicates scripts that may be executed only on an XFA processing application set up as a client.
`server`	Indicates scripts that may be executed only on an XFA processing application set up as a server.
`both`	Indicates scripts that may be executed on either an XFA processing application set up as either a client or a server.

Caution: For security reasons, the server should discard any template it receives in a submitted XDP package and obtain a fresh copy of the template from a trusted source.

The following template fragment includes a calculation that the server executes when the field value changes or when some other event triggers the calculate event on the server.

Example 11.1 Calculation that takes place only on server

```
<field …">
   …
   <calculate>
      <script runAt="server">Num2Date(Date())</script>
   </calculate>
</field>
```

Applications of the runAt="both" property

The script property `runAt="both"` is used primarily for calculations and validations. This script property supports the follow scenarios:

- Server re-calculates and re-validates submissions from the client XFA processing application.
- Client XFA processing application delegates to the server scripts that it (the client) cannot perform. That is, if the client determines that it cannot run the script, it can submit the form to the server with instructions to execute the event. For example, the XFA plug-in for Acrobat (a client XFA processing application) can change pages without going to the server, but the HTML client cannot. The following template segment supports both types of client applications: See "Submitting Data and Other Form Content to a Server" on page 375.

Selecting a Script Language

XFA processors are only required to support one scripting language, FormCalc. FormCalc was designed for XFA to be easy for novice programmers to pick up. At the same time it is fully capable. For more information about FormCalc see the "FormCalc Specification" on page 891.

The Adobe implementations of XFA also support ECMAScript. It is expected that many form creators will prefer ECMAScript because it is more familiar to them.

For each script in a template, the script language is encoded in the `script` element by the `contentType` attribute. The value `application/x-formcalc` signifies FormCalc. The value `application/x-javascript` signifies ECMAScript. If this attribute is not specified the language defaults to FormCalc.

It is entirely permissible to mix scripts of different languages within the same form.

Object References

In XFA scripts, whether in FormCalc or in ECMAScript, objects are referenced using XFA-SOM expressions. XFA-SOM has been defined in such a way that simple references can be used directly either in FormCalc or in ECMAScript as the name of the object. For example, the following object references are syntactically valid in both FormCalc and ECMAScript. Both references refer to the `Total` field within the `Receipt` subform of the form that is being processed. The `Receipt` subform is the root (outermost) subform.

Example 11.2 Simple object references valid in either FormCalc or ECMAScript

```
xfa.form.Receipt.Total
$form.Receipt.Total
```

The ECMAScript specification places certain restrictions upon the syntax of object names. These restrictions mean that more sophisticated XFA-SOM expressions may not be useable as object names. In these cases it is necessary to dereference an XFA-SOM expression by supplying it as a string to a method at run time. FormCalc can dereference mostXFA-SOM expressions without an explicit method call, however it too does have limitations. For more information about these limitations see "Using SOM Expressions in FormCalc" on page 92 and "Using SOM Expressions in ECMAScript" on page 93.

Naked References in ECMAScript

In ECMAScript it is normal to refer to a property of the current object using the `this` symbol. This works as one would expect in XFA. For example, a script associated with a field refers to the field's Y position property as `this.y`. However, in XFA any property of `this` may also be referred to by the property name alone. For example, `this.y` can be referred to simply as `y`. This type of reference is known as a *naked reference*.

Naked references are resolved using the normal XFA-SOM rules for resolving object references. They are mentioned separately here because they are commonly used and because a programmer familiar with other implementations of ECMAScript, but unfamiliar with XFA-SOM, is likely to be puzzled by them.

Passing a Pointer to an Object

As described in "Obtaining the value of an expression" on page 93, ECMAScript (unlike FormCalc) does not automatically resolve object references into references to the object's value. Instead an expression that resolves to an object represents a reference to the entire object (effectively a pointer to the object). This is in keeping with the object orientation of ECMAScript as contrasted to the value orientation of FormCalc.

There are times when scripts must pass references to entire objects. To make such a reference is easier in ECMAScript than in FormCalc. In ECMAScript it is enough to name the object whereas in FormCalc you have to use the built-in function `ref()` to make the reference. For example, the following script is simpler in ECMAScript than it is in FormCalc.

Example 11.3 Script using an object reference in ECMAScript

```
<script contentType="application/x-javascript">
   $dataWindow.isRecordGroup(xfa.datasets.data)
</script>
```

Example 11.4 Equivalent script in FormCalc

```
<script>
   $dataWindow.isRecordGroup(ref(xfa.datasets.data))
</script>
```

In both cases the script returns a Boolean which is True if the node `xfa.datasets.data` is a `dataGroup` and is the current record node but False otherwise. The `dataWindow.isRecordGroup()` method requires an object as its parameter. ECMAScript's behavior is a convenience, because simply naming the object is enough to pass it. By contrast coding the same thing in FormCalc requires the use of the `ref()` built-in-function to prevent the value from being passed, should `xfa.datasets.data` happen to be a `dataValue` node.

Setting Up a Scripting Environment

The XFA processor contains a scripting engine for each scripting language that it supports. The scripting engine creates the environment in which scripts of its language will run. For all scripting engines this environment includes a global object called `xfa`. The `xfa` object is the root of a tree under which are located a number of Document Object Models as subtrees. For example it includes the Template DOM under `xfa.template`, the Form DOM under `xfa.form`, the Data DOM under `xfa.data`, and so on. There are also some pseudomodels such as `xfa.host`. For an introduction to the various DOMs see "Document Object Models" on page 63. Most of the time scripts operate exclusively on the Form DOM.

The scripting environment also contains a number of pseudoobjects, each of which is a shortcut for a DOM or pseudomodel. For example `$form` is a shortcut for `xfa.form`. For the sake of symmetry `$xfa` can also be used as a synonym for `xfa`. For more information about the short forms see "Shortcuts" on page 78.

For a list of the objects below `xfa`, their methods, and their properties see the Adobe XML Form Object Model Reference [FOM].

In addition to the objects under `xfa`, global variables within each scripting engine persist across invocations of the same script or any other script using the same engine. However there is no sharing of global variables across engines, apart from `xfa` and the objects below it. Nor do global variables persist between sessions, except that the XFA processor may save the Data DOM as an XML document and subsequently reload it.

Because of peristence, a FormCalc script may declare functions that are available to subsequent FormCalc scripts. Similarly an ECMAScript script may declare a global object with methods available to subsequent ECMAScript scripts.

Script authors may place data in a subtree under `xfa.datasets` or in a hidden field within the form. Hidden fields are more powerful because they can generate events and otherwise take an active part in the form just as any field can. By contrast data under `xfa.datasets` is passive.

Scripts may be allowed to make changes in any DOM but the change may or may not have any effect depending upon details of the implementation. For example, changing the data loading options in the Configuration DOM generally will not affect data which has already been loaded. Scripts are themselves resident in the Template DOM and Form DOM and must not be altered during processing.

The Relationship Between Scripts and Form Objects

Scripts in XFA do not have an independent existence. Rather a script is always attached to an object, similar to the way a method is attached to an object in object-based programming languages. For a script in XFA this object is either a field, a subform, or a subform set. When the script is executing the symbol for the current object (`"$"` in FormCalc or `this` in ECMAScript) always resolves to the object to which the script is attached.

Note: A script may be physically placed under a `proto` element. However in that location it is just inert text, not executable. Such a script prototype can however supply content for scripts attached to objects. Potentially one prototype can supply content for many scripts.

A field may have a script under its `calculate` property. Such a script is executed whenever the value of the field is needed (for example to display it) and dependency analysis reveals that the previously established value is out of date. The XFA processor understands complicated and indirect dependencies. However when the field content is altered some other way, for example it is manually altered by the user, the resulting value is considered sacrosanct and calculations are inhibited for that field from then on.

A field may have a script under its `validate` property. Such a script is executed in interactive contexts when the focus is about to leave the subform containing the field and dependency analysis reveals that validity may have changed.

A field, subform, or exclusion group may have a script under one or more `event` children. In this position the script is invoked when the specified event occurs. The event may have been triggered by the associated field, subform, or exclusion group, but a different object may be specified as the source of the event. For example a script on one field may be triggered by the cursor exiting some other field.

Tip: Script authors may be tempted to put all the code that is invoked by a particular event into one script. This is not the best style to use with XFA. It is better to put all the code that operates on a particular object into `event` children of that object. In this way each small script can refer to its target as the current object (`"$"` or `this`). This yields better modularity because it means the content of the object is only modified by script provided by the object itself. Modularity in turn make it easier to reuse the script in another field with related functionality. In particular it allows the identical script to be used in each of a potentially large set of fields, where the whole set is represented in the template by a single field with multiple occurrences. To take advantage of this requires an understanding of relative references in XFA-SOM expressions. See “Relative References” on page 94.

For example, the order form below has two buttons each of which operates on multiple text fields. The actions for each text field are gathered together in scripts held by `event` objects under the text field itself.

Example 11.5 Recommended style for linking events on one object to script acting on another

```
…
<field name="SetDefaults" …>
   …
   <ui><button/></ui>
</field>
<field name="Clear" …>
   …
   <ui><button/></ui>
</field>
…
<field name="Item" …>
   …
   <event activity="click" ref="SetDefaults">
```

```
      <script>$ = "TodaysSpecial"</script>
   </event>
   <event activity="click" ref="Clear">
      <script>$ = "N/A"</script>
   </event>
</field>
…
<field name="Quantity" …>
   …
   <event activity="click" ref="SetDefaults">
      <script>$ = "1"</script>
   </event>
   <event activity="click" ref="Clear">
      <script>$ = ""</script>
   </event>
</field>
```

Exception Handling

Exceptions can be thrown during the execution of a script. In general, if the scripting environment doesn't support a feature and this feature is invoked via script, an exception is thrown automatically. Both FormCalc and ECMAScript also allow scripts to throw exceptions programmatically, although FormCalc has no method for scripts to catch exceptions.

FormCalc and ECMAScript respond to exceptions as follows:

- FormCalc. The script stops as soon as an exception is thrown. FormCalc exceptions are described in "FormCalc Specification" on page 891.
- ECMAScript. If an algorithm throws an exception, execution of the algorithm is terminated and no result is returned. The calling algorithms are also terminated, until an algorithm step is reached that explicitly deals with the exception. Once such an algorithm step has been encountered, the exception is no longer considered to have occurred. ECMAScript exceptions are described in the ECMAScript Language Specification [ECMAScript].

XFA processors should display or log a helpful message the first time any particular uncaught exception is thrown by a particular script. Subsequent repetitions of the same exception from the same script within the same session may be silent.

Tip: In Acrobat FormCalc error messages are displayed in the standard error pane but ECMAScript error messages are displayed in the JavaScript Console. Use Control/J to bring up the JavaScript Debugger, which includes the Console.

Exceptions do not affect other scripts. If there is a queue of scripts to be executed, processing continues with the next one in the queue.

Exceptions do not prevent subsequent re-execution of the same script. This is important because a script may refer to an object that does not yet exist, causing an exception. Yet the required object may exist later when the script is re-executed.

Picture Clauses and Localization

The FormCalc functions support localization in several ways, as listed in "Locales" on page 918.

Unicode Support

FormCalc supports Unicode 3.2 [Unicode-3.2], and ECMAScript supports some range of Unicode, depending on the implementation. A conforming implementation of ECMAScript interprets characters in conformance with the Unicode Standard, Version 2.1 or later, and ISO/IEC 10646-1 with either UCS-2 or UTF-16 as the adopted encoding form, implementation level 3. If the adopted ISO/IEC 10646-1 subset is not otherwise specified, it is presumed to be the BMP subset, collection 300 [Unicode-2.1].

Caution: An XFA processor and its scripting engine(s) may internally support bytecodes that are not allowed in XML and thus cannot be loaded from or saved to an XML data document.

XML 1.0 restricts characters to the following production:

```
Char ::= #x9 | #xA | #xD | [#x20-#xD7FF] | [#xE000-#xFFFD] | [#x10000-#x10FFFF]
```

This restriction applies even to characters expressed as numeric entities, according to "Well-formedness constraint: Legal Character" in [XML1.0].

XML 1.1 [XML1.1] removes this restriction, but XML 1.1 is not widely used and XFA processors are not required to support it. (Acrobat does not.) In addition, XFA expressly forbids the use of character code zero (`#x0`), as described on page 112.

12 Using Barcodes

Barcodes require specialized knowledge to use. This chapter does not substitute for a barcode textbook. However it does introduce and discuss issues that arise when using barcodes in XFA forms.

XFA supports barcodes as first-class features of a form. A barcode field is a normal field in every way except that its data is presented as an appropriately formed set of bars. Depending on the type of barcode the data may also be shown as characters in, over, or under the bars. In addition XFA supports compressing and encrypting the data in barcode fields. Compression and encryption are not appropriate for normal human-readable fields but are appropriate for some types of barcodes.

In a barcode the data is presented as a series of bars with varying widths. So-called *one-dimensional barcodes* have a single line of bars. So-called *two-dimensional barcodes* have multiple rows of bars.

The function of a barcode is to be read by a specialized piece of hardware called a barcode reader. Some barcodes also display the data in human-readable format as a string of glyphs, but this is a secondary function. Since the barcode is intended to be read by a machine its appearance is usually strictly constrained. For example, for a particular type of barcode the bars may have to have to particular height and distance apart. In addition it is common for a barcode to require a minimum amount of white space around it (the *quiet zone*) and a particular range of distances from a designated edge of the page.

There are many different types of barcodes, hundreds in wide use and thousands more in specialized applications. Some types of barcode are specified formally by standards organizations. Others, however, are nothing more than conventions. See the bibliographic appendix"Barcode References" on page 1093 for references for some barcode types. Note, however, that the barcode bibliography is not authoritative and includes only those barcode types mentioned in the specification and for which references could be identified.

Some printers (notably label printers) are capable of printing some barcodes all by themselves. The XFA application driving the printer only has to select the barcode type and supply any necessary parameters, including the data to be encoded in bars. This specification refers to such barcodes as *hardware barcodes*. In other cases the printer does not handle the barcode. In those cases the XFA application has to tell the printer to draw the barcode a line at a time. This specification refers to such barcodes as *software barcodes*. Because support for barcodes is a function of both hardware and software, XFA cannot specify a universal set of supported barcodes. Instead XFA supplies a generalized barcode grammar that includes a barcode type identifier and parameters controlling such things as the ratio of thick to thin bar widths. Later sections of this chapter describe those parameters. These parameters cover the needs of all of the common barcode types. However for any particular barcode type some settings are inapplicable. It is normal and expected for templates to contain inapplicable barcode settings; they are silently ignored.

Note: XFA does nothing to express or enforce positioning or quiet zone requirements. It is up to the form creator to ensure that these requirements are met.

XFA specifies barcode identifiers for some of the most commonly used types of barcodes. XFA applications (in combination with particular printers) that support any of these barcodes recognize the specified barcode type identifier. XFA applications may use custom identifiers to support other types of barcodes.

Note: When a specified barcode type is not supported either in hardware or software the result is application-defined.

Special Processing of Barcode Data

Some types of barcodes can hold arbitrary binary data. Others are limited to a particular set of characters or codes. It is the responsibility of the form creator to ensure that the data is appropriate for the barcode, for example by imposing a validation on the field.

Some types of barcodes are inherently fixed in data capacity. Others may optionally grow to fit the supplied data. However even growable one-dimensional barcodes hold a few tens of bytes at most. Two-dimensional barcodes can hold up to a kilobyte or so. When the data does not fit into the barcode the data is truncated.

In view of the limited capacity of even two-dimensional barcodes it is desireable to be able to easily specify a subset of the data to incorporate into the barcode. XFA includes a `manifest` element which represents a collection of nodes. The `manifest` element is a child of `variables` and methods have been provided so that a script can easily set the value of a barcode field to an XML representation of the values of the nodes in a `manifest`.

The data must be expressed as a stream of bytes before it can be placed into the barcode. By default it is encoded and serialized as UTF-8 (without a byte order mark). However other codings can be specified so that the encoding matches the expectations of existing applications. This is particularly important for QRCode barcodes which normally use Shift-JIS encoding.

There is an option to apply lossless compression to the data after it is encoded but before it is printed in the barcode. The default behavior is not to compress. However when the `dataPrep` attribute is set to `flateCompress` the data that is placed in the barcode is a byte with a numeric value of 257, followed by a byte with a numeric value of 1, followed by the data compressed using the method defined by the Internet Engineering Task Force (IETF) in [RFC1951]. No predictor is used. Compression must not be specified for barcode types that cannot hold arbitrary binary.

Finally, the data may be encrypted after it is encoded and optionally compressed but before it is printed in the barcode. The default behavior is not to encrypt. But when the `barcode` element has an `encrypt` child the certificate and other properties of the `encrypt` element are used to encrypt the data with a public key. This way the printed barcode does not reveal its content to anyone who does not know the private key. The encrypted data consists of:

- A byte containing the hexadecimal value 82 (decimal 130).
- A byte containing the value 1.
- A byte containing the second-least-significant byte of the certificate serial number.
- A byte containing the least-significant byte of the certificate serial number.
- A random 16-byte session key encrypted using the algorithm and the public key specified by the certificate.
- The serialized and optionally compressed data encrypted using the RC4 algorithm and the session key.

Note: RC4 is a copyrighted, proprietary algorithm of RSA Security, Inc. Adobe Systems has licensed this algorithm for use in its Acrobat products. Independent software vendors may be required to license RC4 to develop software that encrypts or decrypts RC4-encrypted barcodes. For further information, visit the RSA Web site at <http://www.rsasecurity.com> or send e-mail to <products@rsasecurity.com>.

The chosen encoding, compression, and encryption affect only the printed barcode, not the internal representation of the data or what the user sees when the field has focus in interactive contexts.

Barcode type

The barcode element supplies the information required to display a barcode. This includes the type of the barcode and a set of options which varies from one type of barcode to another. The most important property of the barcode is the `type`. This is a string controlling what sort of barcode to display. This property must be supplied (there is no approved default).

The set of supported values for this property is implementation-defined and may also be specific to the display device. However a set of values have been defined for this property as indicating particular barcode types. In addition individual XFA processors may implement other values for other barcode types, including custom barcodes. However for any of the defined types that the XFA processor implements it recognizes the defined name. The names are:

`codabar`

Codabar, as defined in ANSI/AIM BC3-1995, USS Codabar [Codabar].

`code2Of5Industrial`

Code 2 of 5 Industrial; no official standard.

`code2Of5Interleaved`

Code 2 of 5 Interleaved, as defined in ANSI/AIM BC2-1995, USS Interleaved 2-of-5 [Code2Of5Interleaved].

`code2Of5Matrix`

Code 2 of 5 Matrix; no official standard.

`code2Of5Standard`

Code 2 of 5 Standard; no official standard.

`code3Of9`

Code 39 (also known as code 3 of 9), as defined in ANSI/AIM BC1-1995, USS Code 39 [Code39].

`code3Of9extended`

Code 39 extended; no official standard.

`code11`

Code 11 (USD-8); no official standard.

`code49`

Code 49, as defined in ANSI/AIM BC6-1995, USS Code 49 [Code49].

`code93`

Code 93, as defined in ANSI/AIM BC5-1995, USS Code 93 [Code93].

`code128`

Code 128, as defined in ANSI/AIM BC4-1995, ISS Code 128 [Code128-1995].

`code128A`

Code 128 A, as defined in ANSI/AIM BC4-1995, ISS Code 128 [Code128-1995].

code128B

Code 128 B, as defined in ANSI/AIM BC4-1995, ISS Code 128 [Code128-1995].

code128C

Code 128 C, as defined in ANSI/AIM BC4-1995, ISS Code 128 [Code128-1995].

code128SSCC

Code 128 serial shipping container code, as defined in ANSI/AIM BC4-1995, ISS Code 128 [Code128-1995].

ean8

EAN-8, as defined in ISO/EEC 15420 [ISO-15420]

ean8add2

EAN-8 with 2-digit Addendum, as defined in ISO/EEC 15420 [ISO-15420]

ean8add5

EAN-8 with 5-digit Addendum, as defined in ISO/EEC 15420 [ISO-15420]

ean13

EAN-13, as defined in ISO/EEC 15420 [ISO-15420]

ean13pwcd

EAN-13 with Price/Weight customer data, as defined in ISO/EEC 15420 [ISO-15420]

ean13add2

EAN-13 with 2-digit Addendum, as defined in ISO/EEC 15420 [ISO-15420]

ean13add5

EAN-13 with 5-digit Addendum, as defined in ISO/EEC 15420 [ISO-15420]

fim

United States Postal Service FIM (Facing Identification Mark), as described in First-Class Mail [USPS-C100]. Note that the FIM doesn't carry arbitrary data, there are just 4 possible bar combinations. The data supplied for the barcode must be one of the strings "A", "B", "C" or "D" for FIM A, FIM B, FIM C, FIM D, respectively.

logmars

Logmars (Logistics Applications of Automated Marking and Reading Symbols) as defined by U.S. Military Standard MIL-STD-1189B [LOGMARS].

maxicode

UPS Maxicode, as defined in ANSI/AIM BC10-ISS Maxicode [Maxicode].

msi

MSI (modified Plessey); may have once had a formal specification but not any longer.

pdf417

PDF417, as defined in USS PDF417 [PDF417].

pdf417macro

PDF417, but allowing the data to span multiple PDF417 barcodebarcodes. The barcode(s) are marked so that the barcode reader knows when it still has additional barcodes to read, and can if necessary prompt the operator. This facility is defined in "USS PDF417" [PDF417].

plessey

Plessey; no official standard.

postAUSCust2

Australian Postal Customer 2, as defined in Customer Barcoding Technical Specifications [APO-Barcode].

postAUSCust3

Australian Postal Customer 3, as defined in Customer Barcoding Technical Specifications [APO-Barcode].

postAUSReplyPaid

Australian Postal Reply Paid, as defined in Customer Barcoding Technical Specifications [APO-Barcode].

postAUSStandard

Australian Postal Standard, as defined in Customer Barcoding Technical Specifications [APO-Barcode].

postUKRM4SCC

United Kingdom RM4SCC (Royal Mail 4-State Customer Code), as defined in the How to Use Mailsort Guide [RM4SCC].

postUSDPBC

United States Postal Service Delivery Point Bar Code, as defined in DMM C840 Barcoding Standards for Letters and Flats [USPS-C840].

postUSStandard

United States Postal Service POSTNET barcode (Zip+4), as defined in DMM C840 Barcoding Standards for Letters and Flats [USPS-C840].

postUSZip

United States Postal Service POSTNET barcode (5 digit Zip), as defined in DMM C840 Barcoding Standards for Letters and Flats [USPS-C840].

qr

QR Code, as defined in ISS - QR Code [QRCode].

telepen

Telepen, as defined in USS Telepen [Telepen].

ucc128

UCC/EAN 128, as defined in International Symbology Specification - Code 128 (1999) [Code128-1999].

ucc128random

UCC/EAN 128 Random Weight, as defined in International Symbology Specification - Code 128 (1999) [Code128-1999].

ucc128sscc

UCC/EAN 128 serial shipping container code (SSCC), as defined in International Symbology Specification - Code 128 (1999) [Code128-1999].

upcA

UPC-A, as defined in ISO/EEC 15420 [ISO-15420].

upcAadd2

UPC-A with 2-digit Addendum, as defined in ISO/EEC 15420 [ISO-15420].

upcAadd5

UPC-A with 5-digit Addendum, as defined in ISO/EEC 15420 [ISO-15420].

upcApwcd

UPC-A with Price/Weight customer data, as defined in ISO/EEC 15420 [ISO-15420].

upcE

UPC-E, as defined in ISO/EEC 15420 [ISO-15420].

upcEadd2

UPC-E with 2-digit Addendum, as defined in ISO/EEC 15420 [ISO-15420].

upcEadd5

UPC-E with 5-digit Addendum, as defined in ISO/EEC 15420 [ISO-15420].

upcean2

UPC/EAN with 2-digit Addendum, as defined in ISO/EEC 15420 [ISO-15420].

upcean5

UPC/EAN with 5-digit Addendum, as defined in ISO/EEC 15420 [ISO-15420].

Data formatting

There are a number of properties which govern how the data will be processed before it is written out to the barcode. This processing does not affect the content of the field in the DOM, only the data carried by the barcode. The individual properties are described below.

The charEncoding property

This property controls the serialization of data from the field into the barcode. This does *not* affect the interpretation of data loaded into the barcode field. It only affects the data on its way out to the barcode.

Note that the value of this property is case-insensitive. For that reason it is defined in the schema as cdata rather than as a list of XML keywords. However the value must match one of the following keywords in a case-insensitive manner.

`UTF-8`

The characters are encoded using Unicode code points as defined by [Unicode-3.2], and UTF-8 serialization as defined by *ISO/IEC 10646* [ISO-10646]. There is no byte order mark.

`none`

No special encoding is specified. The characters are encoded using the ambient encoding for the operating system.

`ISO-8859-1`

The characters are encoded using ISO-8859-1 [ISO-8859-1], also known as Latin-1.

`ISO-8859-2`

The characters are encoded using ISO-8859-2 [ISO-8859-2].

`ISO-8859-7`

The characters are encoded using ISO-8859-7 [ISO-8859-7].

`Shift-JIS`

The characters are encoded using JIS X 0208, more commonly known as Shift-JIS [Shift-JIS]. This is commonly used for QR Code barcodes.

`KSC-5601`

The characters are encoded using the *Code for Information Interchange (Hangul and Hanja)* [KSC5601].

`Big-Five`

The characters are encoded using Traditional Chinese (Big-Five). **Note:** there is no official standard for Big-Five and several variants are in use. XFA uses the variant implemented by Microsoft as code page 950 [Code-Page-950].

`GB-2312`

The characters are encoded using Simplified Chinese [GB2312].

`UTF-16`

The characters are encoded using Unicode code points as defined by [Unicode-3.2], and UTF-16 serialization as defined by *ISO/IEC 10646* [ISO-10646]. There is no byte order mark.

UCS-2

The characters are encoded using Unicode code points as defined by [Unicode 3.2], and UCS-2 serialization as defined by *ISO/IEC 10646* [ISO-10646]. There is no byte order mark.

fontSpecific

The characters are encoded in a font-specific way. Each character is represented by one 8-bit byte. The font referred to is font property of the enclosing field.

The dataPrep property

This property controls what preprocessing is applied to the data which is written out in the barcode. This does not affect the data in the DOMs, nor does it affect what the user sees when the field has focus in interactive contexts.

none

Use the data just as supplied. This is the default.

flateCompress

Write out a header consisting of a byte with decimal value 129 (0x81 hex) followed by another byte with decimal value 1. Then write the data compressed using the Flate algorithm, as defined by the Internet Engineering Task Force (IETF) in [RFC 1951]. No predictor algorithm is used. It is an error to specify this option with a `type` that cannot encode arbitrary binary data.

Note: The value `flateCompress` is recommended for use with 2-D barcodes only but this specification does not forbid its use with 1-D barcodes.

The encrypt property

This property controls encryption of the barcode data. If encryption is specified the data is first serialized, then optionally compressed, then encrypted. This does not affect the data in the DOMs, nor does it affect what the user sees when the field has focus in interactive contexts.

Encryption uses a public-key algorithm. This permits the use of paper forms as a secure data channel.

The encrypt property has two uses, for encrypting barcodes and for encrypting data submitted to a host. For more information see "The encrypt element" in the template syntax specification.

The startChar property

Optional starting control character to prepend to barcode data. This does not affect the content of the field, however it may be displayed in the human-readable legend. For example, it is displayed when the barcode type is `codabar`.

This property is ignored by the XFA processor if the barcode type does not allow it.

The endChar property

Optional ending control character to append to barcode data. This does not affect the content of the field, however it may be displayed in the human-readable legend. For example, it is displayed when the barcode type is `codabar`.

This property is ignored by the XFA processor if the barcode type does not allow it.

The checksum property

Algorithm for the checksum to insert into the barcode. The checksum is calculated based upon the supplied barcode data. The template schema allows any one of the choices listed below. Some barcode formats, however, either require a particular checksum or never allow a checksum; for such barcodes the XFA processor ignores this property. Some of the remaining barcode formats support only a limited subset of these choices; for such barcodes the template must not specify a non-supported choice.

Barcode types that use the `errorCorrectionLevel` property ignore this property and vice-versa.

`none`

Do not insert a checksum. This is the default.

`auto`

Insert the default checksum for the barcode format.

`1mod10`

Insert a "1 modulo 10" checksum.

`2mod10`

Insert a "2 modulo 10" checksum.

`1mod10_1mod11`

Insert a "1 modulo 10" checksum followed by a "1 modulo 11" checksum.

"1 modulo 10", "2 modulo 10", and "1 modulo 11" are defined in barcode standards documents for the barcodes to which they apply.

The errorCorrectionLevel property

1-D barcodes and some 2-D barcodes use simple checksums that can detect but not correct errors. By contrast certain 2-D barcodes have more sophisticated *forward error correction* mechanisms. These include redundant information which is calculated from the data in such a way that, if an error occurs and it does not affect too much of the data, the error can be mathematically corrected. For those barcodes the checksum property is ignored and this property instead controls the error correction level. The permitted range of values is specific to the barcode type. For example, for PDF417 the valid values are integers in the range 0 through 8, inclusive.

Barcode types that use this property ignore the `checksum` property and vice-versa.

The wideNarrowRatio property

Ratio of wide bar to narrow bar in supported barcodes.

The allowable range of ratios varies between barcode formats and also, for hardware barcodes, the output device. The template must not specify a value outside the allowable range. The XFA processor ignores this property for barcode formats which do not allow a variable ratio of wide to narrow bar widths. The default value for this property is 3:1.

The syntax for the value of this property is *wide*[:*narrow*] where:

wide is a positive number representing the numerator of the ratio, and

narrow is an optional positive number representing the denominator of the ratio. If *narrow* is not supplied it defaults to `1`.

The following values are equivalent: `2.5:1`, `2.5`, and `5:2`.

Legends

Some barcodes support the printing of the text in the barcode inside, above, or below the barcode itself. The following properties are defined to control the printing of legends. These properties are ignored when the barcode is of a type that does not allow a legend.

The region available for embedded text, if any, is determined by the barcode format. For most barcode formats it is a single contiguous region, but for EAN series barcodes it is split up into four separate regions. The typeface and size are inherited from the enclosing field. The template must specify a typeface and size for the field that will fit into the provided space without overlapping any bars. The typeface should be non-proportional. The XFA application centers the text in the provided space.

If the data in the barcode is encrypted or compressed the legend, if any, is the compressed and/or encrypted text.

Legends are controlled by the following properties, where applicable.

The textLocation property

Location, if any, of human-readable text. May be one of:

`below`

Text is placed below the barcode. This is the default.

`above`

Text is placed above the barcode.

`belowEmbedded`

Text is partially embedded in the bottom of the barcode. The baseline of the text is aligned with the bottom of the bars.

`aboveEmbedded`

Text is partially embedded at the top of the barcode. The top of the text is aligned with the top of the bars.

`none`

No text is displayed.

When the specification for the barcode type requires the legend to be in one particular place, or forbids the display of a legend, this property is ignored. If the template specifies `belowEmbedded` and there is no embedded text region at the bottom of the barcode, the XFA processor may interpret the property as `below`. Similarly if the template specifies `aboveEmbedded` and there is no embedded text region at the bottom of the barcode, the XFA processor may interpret the property as `above`. Otherwise it is an error for the template to specify a location that is not supported by the type of barcode.

The printCheckDigit property

Specifies whether the check digit(s) is/are printed in the human-readable text.

The XFA processor ignores this property if no checksum is generated, as determined by the other properties of the barcode.

0

Do not print the check digit in the human-readable text, only in the barcode itself. This is the default.

1

Append the check digit to the end of the human-readable text.

Properties of one-dimensional barcodes

The dataLength property

The expected maximum number of characters for this instance of the barcode.

For software barcodes, when `moduleWidth` is not specified, this property must be supplied by the template. The XFA processor uses this value and the field width, plus its knowledge of the barcode format, to compute the width of a narrow bar. The width of a wide bar is derived from the width of a narrow bar. When `moduleWidth` is specified this property, if present, is ignored by the XFA processor.

For hardware barcodes this parameter is ignored. Because the XFA processor does not know the details of the barcode format, it cannot use this information to determine the bar width.

The data being displayed is not validated against this parameter. For software barcodes the XFA processor allows the data to overflow the assigned region of the field. For hardware barcodes the result of an overflow depends upon the printer.

Note that there is no corresponding minimum length restriction. Some barcode formats have a fixed number of symbols and must be filled out with padding characters. Others allow a variable number of symbols and terminate after the last symbol.

The moduleHeight property

A module is a set of bars encoding one symbol. Usually a symbol corresponds to a character of supplied data. This property determines the height of the bars in the module. The allowable range of heights varies from one barcode type to another. The template must not specify a height outside the allowable range.

When this property is not supplied, 1-D barcodes grow to the height of the enclosing field, limited by the allowable height range.

This property is also used by 2-D barcodes but they have a different default behavior.

The moduleWidth property

For 1-D software barcodes the XFA processor sets the width of the narrow bars to the value of this property. The width of the wide bars is derived from that of the narrow bars. The allowable range of widths varies from one barcode format to another. The template must not specify a value outside the allowable range. If `moduleWidth` is supplied the XFA processor ignores the `dataLength` property. Conversely `moduleWidth` has no default, so when `dataLength` is not supplied then `moduleWidth` must be supplied.

For 1-D hardware barcodes `moduleWidth` either has no effect or has the same effect as for a software barcode, depending upon the printer and barcode. However for hardware barcodes the template may fall back upon the default value for this property. The default is 0.25mm. The allowable range for the value varies between printers and between barcodes.

This property is also used by 2-D barcodes but they interpret it differently.

Properties of two-dimensional barcodes

The dataColumnCount and dataRowCount properties

These properties are used to force a 2-D barcode to have a fixed number of rows and column. If either of these values is supplied by the template it must supply both values. If the supplied data does not fill the barcode it is padded out with padding symbols.

If these properties are not supplied the XFA processor uses `rowColumnRatio` plus the actual length of the data being inserted to determine the row and column count.

The moduleHeight property

A module is a set of bars encoding one symbol. Usually a symbol corresponds to a character of supplied data. This property determines the height of the bars in the module. The allowable range of heights varies from one barcode type to another. The template must not specify a height outside the allowable range.

When this property is not supplied, the default behavior depends on the type of barcode. 2-D barcodes default to a module height of 5mm.

1-D barcodes also use this property but they display a different default behavior.

The moduleWidth property

For 2-D barcodes the value of this property determines the module width. A module is a set of bars encoding one symbol. Usually a symbol corresponds to a character of supplied data. The allowable range of widths varies from one barcode format to another. The template must not specify a value outside the allowable range.

1-D barcodes also use this property but they have a different interpretation of the value. The default value for this property (0.25mm) is not useful for 2-D barcodes.

The rowColumnRatio property

Optional ratio of rows to columns for supported 2-D barcodes.

The XFA processor ignores this property if `dataRowCount` and `dataColumnCount` are specified.

When `rowColumnRatio` is supplied the XFA processor allows the barcode to grow to the number of rows required to hold the supplied data. If the last row is not filled by the supplied data it is padded out with padding symbols.

The truncate property

Truncates the right edge of the barcode for supported formats. Of the barcodes in the standard types list, this applies only to PDF417. The XFA processor ignores this property for barcode formats to which it does not apply.

0

The right-hand synchronization mark is included. This is the default.

1

The right-hand synchronization mark is omitted.

13 Forms That Initiate Interactions with Servers

This chapter describes the behavior of event objects that interact with servers. Such event objects allow you to implement forms with a range of Web behavior.

In a simple Web-interactive form, event objects can cause part or all of the XFA form to be submitted to a server, as though the user is committing the form. In some situations, data submission is followed by the the server processing the data as specified in the template and returning the result to the client. These simpler cases of submission are described in "Submitting Data and Other Form Content to a Server" on page 375.

In a more complex Web-interactive form, event objects can dynamically request a server to provide additional information and then populate the form with the delivered information. Further, the form may select from multiple servers depending on the data view. Such more complex interactions are described in "Structure of a Web Service Message" on page 383 and "Using Web Services" on page 382.

Forms can also interact with a external data, where the platform supports it, using ActiveX® Data Objects. This Microsoft®-specific interface is closely coupled to the ADO Application Program Interface, as described in [ADO]. This is a record-oriented interface which is most often used for accessing databases. It does not have Web-strength security features and hence is most suitable for use within a corporate LAN or WAN. The use of this interface is discussed in "Invoking ADO APIs Through the Source Set DOM" on page 397.

Forms can also interact with humans or servers via e-mail. In an e-mail based form, event objects cause part or all of the XFA form to be sent as a message to an e-mail address specified by the form. It is also possible for the form to send itself from one client to another until the workflow is complete. The use of e-mail messages is described in "Submitting Data and Other Form Content via E-mail" on page 403.

Submitting Data and Other Form Content to a Server

This section describes content submission performed by event objects having a submit action (henceforth called *submitting events*). More specifically, it describes the tasks that occur when such an event is activated, the types of content included in the submission, packaging of that content, and the encoding of those packages.

Submitting events may use the HTTP POST operation to submit content to a server. Such events request the origin server to accept the submitted content as a new subordinate of the resource identified by the Request-URI in the Request-Line. [HTTP] describes HTTP POST operations as follows:

> *POST is designed to allow a uniform method to cover the following functions:*
>
> - *Annotating an existing resource*
> - *Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles*
> - *Providing a block of data, such as the result of submitting a form, to a data-handling process*
> - *Extending a database through an append operation*

Alternatively submitting events may send e-mail.

About

Content submission is specified using an event object that specifies the `submit` property as the action to perform when the event is activated. (See also the syntax description for the `submit` element.)

The object that describes content submission may be a property of any type of event; however, it is typically associated with a click event that has a button appearance, as shown in the following example. Users would click such a button to indicate they have finished filling out the form and wish to submit it for processing. Before the content submission is allowed to progress, the form data must be successfully validated and other scripts must successfully execute. Typically, if the validation or scripts fail, users are asked to make corrections and resubmit the form. When the processing application successfully submits the form content, the form is said to be *committed*. That is, the content is in a final form.

Example 13.1 Field which is a submit button

```
<field name="Button1" y="223.31mm" x="134.41mm" w="35.98mm" h="18.52mm">
   <ui>
      <button/>
   </ui>
   …
   <event activity="click">
      <submit
           embedPDF="1"
           format="xdp"
           target="http://www.example.org"
           textEncoding="UTF-16"
           xdpContent="pdf datasets template"/>
   </event>
</field>
```

In the above example the domain part of the target URI is "`http:`", which indicates that the submission is to be performed via an HTTP POST operation. If the domain was "`mailto:`" then the submission would take place via e-mail.

Submitting a form and saving a form are similar in that they convert the Template/Form/Data DOM into an XML or PDF representation, but they differ in the level of checking done. Saving a form does not involve any validation or other checks because users may save forms at various stages of completion. Users do not expect to see error reports and warnings during such saves.

Content Interchange

The `submit` syntax is intended to support most XML-based content interchanges. It does so by specifying what types of content are submitted to the server, how content should be packaged (`format`, `embedPDF` and `xdpContent`) and how the content should be encoded (`textEncoding`). Starting with XFA 2.5 it also specifies what signatures should be applied (`signData`) and what encryption should be used (`encrypt`).

The behavior of a `submit` element differs depending on other properties in the submitting event. That is, submission may invoke any of the following behaviors:

- Client submits the specified content to the server. The server executes scripts flagged for execution on the server or on both the server and the client.
- Client submits the specified content with the expectation that the server will perform the script and return a result. The server performs the script and returns the result in its response to the HTTP POST.

The factors that influence submission behavior include the following:

Factor	Explanation
Event trigger	Submission may differ for an event triggered by user interaction, as opposed to an event activated by some other trigger. User-triggered events include: `enter`, `exit`, `mouseEnter`, `mouseExit`, `change`, `click`, `mouseUp`, and `mouseDown`.
Script	Submission may differ, depending on whether a script is included in the event and on the value of the script's `runAt` attribute. The `runAt` attribute specifies where the script should be executed.

The following table describes shows how the above factors affect submission behavior. This table applies *only* to events that include `submit` elements.

Factors That Influence Submission Behavior

Factors		Client tasks	Server tasks
event trigger	**script**		
User initiated	`runAt="client"` or `runAt` undefined	Execute script. Do *not* post data to server. See "Client Submit Event That Does Not Expect Returned Data" on page 378.	
	`runAt="server"`	Post data to server with an indication of the triggered event. When data returned from server, remerge it into form and display results. See "Client Submit Event That Processes Data at the Server" on page 380.	Binding data to form, activate indicate event, and return data to client. The data is returned in the HTTP POST response.
	`runAt="both"`	If the client successfully executes the script, the client does *not* post data to the server. This approach is taken to avoid a flood of unintentional interactions between the client and server. If the client cannot to process the script, the XFA processing application may chose to invoke the `submit` element to elicit the same behavior as described for `runAt="server"`.	Binding data to form, activate indicate event, and return data to client. The data is returned in the HTTP POST response. See "Client Programatically Submits a Request to the Server" on page 381.
	no script provided (but event contains a `submit` element)	Submit data to server, with an indication that a `preSubmit` event has occurred. The client has no expectation that data will be returned.	See "Client Submit Event That Does Not Expect Returned Data" on page 378.

Factors That Influence Submission Behavior

Factors		Client tasks	Server tasks
event trigger	**script**		
other initiated, such as form `ready`	`runAt="client"` or undefined	Execute script. Do *not* POST data to server.	
	`runAt="server"`	Ignore event. Such events are ignored to avoid a flood of unintentional interactions between the client and server.	
	`runAt="both"`	Execute script. Do *not* POST data to server.	
	no script provided (but event contains a `submit` element)	Submit data to server, with an indication that a `preSubmit` event has occurred. The client has no expectation that data will be returned.	See "Client Submit Event That Does Not Expect Returned Data" on page 378.

The following sections discuss how the `submit` syntax supports various form-oriented workflows.

Client Submit Event That Does Not Expect Returned Data

This section describes the processing steps that are initiated when a submitting event is activated. The steps depend on whether the XFA processing application declare itself as a client or as a server.

Client. The client XFA processing application performs the following steps when a `submit` event is activated:

1. Execute all validation scripts that contain a `runAt` attribute set to `client`, `both` or undefined. If this step raises exceptions, the requested content should not be submitted to the server.

2. Execute all `nullTest` checks. If this step raises exceptions, the requested content should not be submitted to the server.

3. Execute all `formatTest` checks. If this step raises exceptions, the requested content should not be submitted to the server.

   Note: Validation, `nullTest` and `formatTest` are executed for the entire form, regardless of the actual content submitted.

4. Collect the specified data into a `connectionData` element underneath `$datasets`.

   Note: When the above step completes, the Connection Data DOM changes state.

5. Examine the form for any `preSubmit` events that contain scripts with a `runAt` attribute set to `client`, `both` or undefined. If such scripts are found, execute them. Scripts executed at this point have the chance to examine and alter the data before it is submitted. If this step raises exceptions, the requested content may still be submitted to the server. ("DOM Events" on page 336)

6. Package the data and any other content, as specified in the `submit` properties: `format`, `embedPDF` and `xdpContent`. ("Content Interchange" on page 376)

7. Apply, clear, or validate signatures on the data and other content as specified by all `signData` children of the controlling `submit` element. This step is new in XFA 2.5.

8. Encrypt the signature, data and other content as specified by the `encrypt` child of the controlling `submit` element. This step is new in XFA 2.5.

9. Send the data to the server using the HTTP post operation. If the form contains any `preSubmit` events with scripts having a `runAt` attribute set to `server` or `both`, the client includes in the data special instructions. These special instructions tell the server which `preSubmit` events it (the server) should immediately process. The format of such special instructions is implementation-defined.

   The response to an HTTP POST failure is implementation-defined.

Server. The server XFA processing application typically performs the following steps after it receives the data:

Caution: For security reasons, the server should discard any template submitted in the XDP package and obtain a fresh copy of the template from a trusted source.

1. Decrypt the HTTP POST data if it is encrypted.

2. Validate the signature if one is required.

3. Create the Form DOM.

4. Examine the data for special instructions that indicate `preSubmit` events. If such instructions are found, examine the events they indicate for scripts that contain a `runAt` attribute set to `server` or `both`. If such scripts are found, execute them.

5. Execute all calculation scripts that contain a `runAt` attribute set to `server` or `both`.

6. Execute all validation scripts that contain a `runAt` attribute set to `server` or `both`.

7. Execute the scripts in any events activated as part of submit processing, provided those scripts contain a `runAt` attribute set to `server` or `both` and provided they do not contain `submit` elements.

Example 13.2 An event that does not expect data to be returned from the server

```
<field name="Button1" …>
   <event activity="click">
      <submit target="mailto:me@example.org"
         textEncoding="UTF-8" xdpContent="datasets template"/>
   </event>
</field>
<field name="NumericField1" …>
   <value>
      <float>100</float>
   </value>
   </caption>
   <event activity="preSubmit" ref="$form">
      <script runAt="server">valueA * valueB</script>
   </event>
</field>
<field name="valueA" …>
   <value>
      <float/>
```

```
   </value>
</field>
<field name="valueB" …>
   <value>
      <float/>
   </value>
</field>
```

Client Submit Event That Processes Data at the Server

This section describes the processing steps that are initiated when a submitting event is activated. The steps depend on whether the XFA processing application declares itself as a client or as a server.

Client-side actions

The client XFA processing application performs the following steps when a `submit` event is activated. The submitting event requests data be processed and returned. (See the table "Factors That Influence Submission Behavior" on page 377).

1. Package the data and any other content, as specified in the `submit` properties `format`, `embedPDF` and `xdpContent`. ("Content Interchange" on page 376)
2. Sign the data and other content as specified by the `signData` child of the controlling `submit` element. This step is new in XFA 2.5.
3. Encrypt the signature, data and other content as specified by the `encrypt` child of the controlling `submit` element. This step is new in XFA 2.5.
4. Include in the package an indication of the event initiating the submission. The format of this is implementation-defined.
5. Send the data to the server using the HTTP POST operation.
6. When the HTTP POST operation returns (Step 5. below), merge the returned data into the Data DOM. For more information on the HTTP POST interchange, see [HTTP].

   The response to an HTTP POST failure is implementation-defined.

Server-side actions initiated by above

The server XFA processing application typically performs the following steps after it receives the data:

Caution: For security reasons, the server should discard any template submitted in the XDP package and obtain a fresh copy of the template from a trusted source.

1. Decrypt the HTTP POST data if it is encrypted.
2. Validate the signature if one is required.
3. Create the Form DOM.
4. Execute the scripts in the indicated event.
5. Package the data and include it in the HTTP POST return.

Client Programatically Submits a Request to the Server

Regardless of the value of `runAt`, the submission defined by a `submit` element can be invoked programmatically by scripts. The action specified by the `submit` element is exposed to scripts as a `submit()` method on the parent `field` object.

Example 13.3 Invoking a submission defined in the template programmatically

```
<field name="x" …>
   …
   <submit …/>
</field>
…
<field name="submitButton" …>
   …
   <event activity="click">
      <script>x.submit()</script>
   </event>
</field>
```

This facility is especially useful when the client may not support a needed function, as shown in the following example.

Example 13.4 Script falls back to carrying out the pageDown() action on the host

```
<field name="pagedown" …>
   …
   <event activity="click">
      <script runAt="both" contentType="application/x-javascript">
         system == xfa.host.pageDown();
         onerror = xfa.form.root..pagedown.submit();
      </script>
      <submit
         format="xdp"
         target="http://www.example.org"
         textEncoding="UTF-16"
         xdpContent="datasets"/>
   </event>
</field>
```

Because the script has `runAt="both"` it would not normally be submitted to a server even though the event contains a `submit` element. However the script tries to process the `pageDown()` method on the current XFA processor and, if that fails, programmatically submits the request to the host. ECMAScript is used rather than FormCalc because ECMAScript provides convenient facilities for catching runtime errors.

A template that contained the above example would elicit different results, depending on the capabilities of the application processing it, as described below:

- Acrobat with XFA plug-in. This XFA processing application implements the `pageDown()` method. As a result, if the user clicks on the field named `pagedown`, the script command to page down is successful.
- Adobe XFA HTML client. This XFA processing application does not implement the `pageDown()` method. As a result, if the user clicks on the field named `pagedown`, ECMAScript detects an error event (`onerror`). The ECMAScript error event invokes the `submit` element, which causes the XFA HTML client to send the script to the server for execution and then to merge the returned result.

Using Web Services

Web services provide a flexible communication interface between many different systems, allowing clients to submit data to and request data from servers, databases, custom systems, vendors exposing Web Services, and others. XFA allows fields to connect to web services and thus access any of these facilities.

Data descriptions and connection sets are used heavily for web services based on [WSDL1.1] and [SOAP1.1]. Furthermore the XFA structures for the two are tightly integrated. For that reason, the reader should be familiar with "Data Description Specification" on page 834 and "Data Description Specification" on page 834 before proceeding with this section.

It is important to clarify the terms *client* and *server*. Normally in XFA the server is the computer that serves the template to the XFA client. However web services have their own client-server relationship. The server that provides a web service most likely has no knowledge of XFA or of the template. Instead, in web services parlance, the XFA processor that initiates the transaction with (makes the request of) the web service is the client and the provider of the web service is the server. In this chapter the words client and server, unless otherwise qualified, have the web service meanings.

Web Service Architecture

The web service architecture can be summarized as follows:

- Fields and exclusion groups can act as either senders of data to a particular web service, receivers of data from the web service, or both.
- A field or exclusion group is not limited to a single Web Service. Instead it has the ability to interact with different web services at different times.
- Data may be coerced into a web service's input message schema, as defined by a provided data description. The coercion is done in a separate DOM (the Connection Data DOM) so it does not affect the regular Data DOM. The root node for the Connection Data DOM is `xfa.datasets.connectionData` (also known as `!connectionData`).
- Scripts in the template may inspect and modify the data while it is in the Connection Data DOM.
- Data returned from the web service is retained in the Connection Data DOM in its own schema.
- Scripts in the template may inspect and modify data returned from a web service while it is retained in the connection DOM.
- Data from a web service may be imported into the existing Form DOM, or used to create a new Form DOM.
- In this version, XFA supports only web services that implement *doc-literal* SOAP operations over HTTP. This means that the Web Service's WSDL must define a SOAP binding, operations with "document" style, and messages with "literal" encoding. These terms are defined in Web Services Description Language (WSDL) 1.1 [WSDL1.1]. RPC (Remote Procedure Call)-style operations are *not* supported by this version of XFA. Also "encoded" messages are *not* supported by XFA, even though [WSDL1.1] permits their use with "document" style operations.

Cycle of Operation in a Web Service Transaction

The cycle of operation in a web service transaction goes through the following steps:

1. An event object is activated. The event object has an execute property. The execute property has a connection subproperty which names a particular connection within the `connectionSet`.

2. The XFA application marshals the complete SOAP message in the Connection Data DOM. The schema for the message comes from the data description named by the connection's `dataDescription` property. The message includes data from all subforms, fields, and exclusion groups that are linked to the connection by their connect children. Each connect child supplies a pointer mapping to a node in the message where its parent's data is copied.

3. The `preExecute` event is triggered. If there is a script associated with it, the script has a chance to examine and modify the message in the Connection Data DOM. For example, the script may add additional SOAP headers.

4. The XFA application serializes the message in the Connection Data DOM to XML and constructs the input message for the service.

5. The XFA application sends the input message to the server.

6. The server performs the operation.

7. The server sends a reply to the XFA application. The reply may include an output message.

8. If the operation has an output message, it contains data serialized as XML. The XFA application loads the data from the received message into the Connection Data DOM, replacing the input message that was there previously.

9. The `postExecute` event is triggered. If there is an associated script, it runs. While the script is running`$event.soapFaultCode` and`$event.soapFaultString` are set to the received fault code and fault string. These are contained in the `soap:faultcode` and `soap:faultstring` elements, respectively, inside the `soap:fault` element. If the operation succeeded, there is no `soap:fault` element and both event properties default to the empty string (“”). The script can detect success by checking for an empty string in `$event.soapFaultCode`. The script can also inspect and modify the received data in the Connection Data DOM before it is imported into the Form DOM. For example, it may check for headers.

10. The XFA application imports the received data into the Form DOM. There are two ways the XFA processor can carry out the importation. When the executeType property of the event object is set to import, it simply updates the data that is bound to the output of the connection. This is simple and efficient but it does not support dynamic subforms, which are inserted into the form where required and/or as often as required by the data. For dynamic subforms the XFA processor clears the Form DOM and rebuilds it using a merge (data binding) operation. This is done when the `executeType` property of the event object is set to import.

    Note: The merge operation is modified when there is a connection active. As the XFA processor builds the Form DOM, when it comes to a candidate field or exclusion group associated with the current connection, it reaches into the Connection Data DOM and plucks the associated data (if any) from there. If the field or exclusion group was already bound to data in the Data DOM, the new data propagates through to the Data DOM, updating the node that is already there; otherwise a new data node is created to hold the data.

11. The message in the Connection Data DOM is deleted.

Structure of a Web Service Message

A Web Service message uses the XFA data description grammar to describe its header and body (“Data Description Specification” on page 834) and references a Web connection defined in the XFA connection set (“Connection Set Specification” on page 820).

The data description for a web service is a special case. The data description must declare the schema for the input message (from client to server). In addition it must declare the name of the web service connection which will be referenced in connect elements in the template. This is done using the following structure:

```
<dd:dataDescription
   xmlns:dd="http://ns.adobe.com/data-description/"
   dd:name="dataDescriptionName">
   <connectionName>
      <soap:Header
         xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
         ... data description for header ...
      </soap:Header>
      <soap:Body
         xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
         ... data description for message ...
      </soap:Body>
   </connectionName>
</dd:dataDescription>
```

Note: There can only be one *connectionName* element per data description. Note also that the *connectionName* element's namespace is ignored.

For example, the following data description declares the message schema to use with a web service connection called `POConnection`.

Example 13.5 Data description for the POConnection web service

```
<dd:dataDescription
   xmlns:dd="http://ns.adobe.com/data-description/"
   dd:name="ExampleSoapInfo">
   <POConnection>
      <soap:Body
         xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
         <po1:orderItem
            xmlns:po1="http://www.example.com/po1">
            <po1:OrderId/>
            <po1:Description dd:minOccur="0"/>
            <po1:Quantity/>
         </po1:orderItem>
      </soap:Body>
   </POConnection>
</dd:dataDescription>
```

In the above example, the `soap:Body` element contains the schema for the message. The optional `soap:Header` element has been omitted.

Example That Illustrates the Web Services Architecture

The best way to understand this architecture is to go through an example. This example uses a simple stock-quote service. In order to use the service the client sends a message to the server containing a header and a ticker symbol. The server replies with a message containing status information and, if the query is successful, the current stock price.

This example is borrowed from the WSDL 1.1 specification [WSDL1.1].

Example 13.6 Stockquote web service description file

```
<?xml version="1.0"?>
<definitions name="StockQuote"
  targetNamespace="http://example.com/stockquote.wsdl"
  xmlns:tns="http://example.com/stockquote.wsdl"
  xmlns:xsd1="http://example.com/stockquote.xsd"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <schema targetNamespace="http://example.com/stockquote.xsd"
      xmlns="http://www.w3.org/2000/10/XMLSchema">
      <element name="TradePriceRequest">
        <complexType>
          <all>
            <element name="tickerSymbol" type="string"/>
          </all>
        </complexType>
      </element>
      <element name="TradePrice">
        <complexType>
          <all>
            <element name="price" type="float"/>
          </all>
        </complexType>
      </element>
    </schema>
  </types>

  <message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
  </message>

  <message name="GetLastTradePriceOutput">
    <part name="body" element="xsd1:TradePrice"/>
  </message>

  <portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
      <input message="tns:GetLastTradePriceInput"/>
      <output message="tns:GetLastTradePriceOutput"/>
    </operation>
  </portType>

  <binding name="StockQuoteSoapBinding"
    type="tns:StockQuotePortType">
    <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetLastTradePrice">
      <soap:operation
        soapAction="http://example.com/GetLastTradePrice"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
```

```
54          <soap:body use="literal"/>
55        </output>
56      </operation>
57    </binding>
58
59    <service name="StockQuoteService">
60      <documentation>My first service</documentation>
61      <port name="StockQuotePort"
62        binding="tns:StockQuoteBinding">
63        <soap:address
64          location="http://example.com/stockquote"/>
65      </port>
66    </service>
67
68 </definitions>
```

This definition file tells potential clients how to access the service. The following table describes the definition file's parts. For more information about the meaning of WSDL definition files see [WSDL1.1].

Line #	Element name	Defines
8	types	XML components used in the other sections
28	message	Input message
32	message	Output message
36	portType	Operations and how they use messages
43	binding	Binding between the messages and the SOAP protocol (GetLastTradePriceInput as the input message and GetLastTradePriceOutput as the output message)
59	service	URL of the server and the name of the service (port)

Input message queries server for the trading price of a corporation

The input message defined by the above WSDL definition carries a ticker symbol for a publicly-listed corporation. It has the following form:

```
<soap:Body
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <tns:TradePriceRequest>
      <tns:tickerSymbol>stockTickerSymbol</tns:tickerSymbol>
   </tns:TradePriceRequest>
</soap:Body>
```

Note: This is not an XFA-specified schema. It is specified by the web service.

Output message provides trading price (if successful) or a status indicator (if not successful)

If the query succeeds (that is, if a share quotation can be obtained for the given ticker symbol), the output message from the server carries the price per share for the requested corporation. It has the following form:

```
<soap:Body
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
```

```
   <tns:TradePrice>
      <tns:price>pricePerShare</tns:price>
   </tns:TradePrice>
</soap:Body>
```

Note: This is not an XFA-specified schema. It is specified by the web service.

If the query fails (for example because there is no known listing for the given ticker symbol), the output message carries a status indicator. It has the following general form (with whitespace added for clarity):

```
<soap:Body
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Fault>
      <faultcode>...</faultcode>
      <faultstring>...</faultstring>
   </soap:Fault>
</soap:Body>
```

The `soap:Fault` element can also contain a `faultactor` element but this is uncommon.

Note: This is not an XFA-specified schema. It is specified by the web service.

Definition file may define multiple individual operations, each using a different connection set

The service defined by this definition file can include many individual operations. However a `wsdlConnection` element describes just one operation. Hence it may require many `wsdlConnection` elements to fully describe a service.

The simple service in this example supports a single operation, so in this case only one `wsdlConnection` element is needed.

Example 13.7 Complete connection set packet

```
<connectionSet
   xmlns="http://www.xfa.org/schema/xfa-connection-set/2.4/">
   <wsdlConnection
      dataDescription="DataConnectionTradePriceRequestDD"
      name="TradePriceWS">
      <wsdlAddress>
         http://example.com/stockquote.wsdl
      </wsdlAddress>
      <soapAction>
         http://example.com/GetLastTradePrice
      </soapAction>
      <soapAddress>
         http://example.com/StockQuote
      </soapAddress>
      <operation input="GetLastTradePriceInput"
         output="GetLastTradePriceOutput">
         TradePriceRequest
      </operation>
   </wsdlConnection>
</connectionSet>
```

The `wsdlConnection` element has attributes (below) that link it to other parts of the XDP.

- `dataDescription` attribute points to a data description that must be created ("Data Description Specification").
- `name` attribute supplies a name for this connection that is used in `connect` elements to indicate that the content of a field, exclusion group, or subform takes part in the transaction ("Connection Set Specification").

The `wsdlAddress` child of the `wsdlConnection` element contains the URL of the WSDL service definition. This is optional information for the use of form creation tools. In this case, its location happens to be the same as its namespace, but it could be anywhere.

The `soapAction` child of `wsdlConnection` is copied from the `soapAction` attribute of the `soap:operation` element in the WSDL definition file. In this case `soapAction` is empty because in the WSDL definition file the value of the attribute is the null string.

Similarly, the `soapAddress` child of `wsdlConnection` is copied from the location attribute of the `soap:address` element in the WSDL definition file.

The `operation` child of `wsdlConnection` associates the names used within the XDP to the operation(s) and messages defined in the WSDL definition. Here, the service exposes one operation, identified by the string `"TradePriceRequest"`. This is the content of the operation element. The input and output attributes identify the element definitions for the input and output messages, respectively, in the WSDL definition.

Note that in a WSDL file several operations with the same name can exist. This is analogous to function overloading. In this case, the input and output attributes uniquely identify the selected operation. If the input or output element in the WSDL file does not have a name attribute, the attributes must be set to the default input or output name as specified in [WSDL1.1] Section 2.4.5.

The associated data description controls the format of the message sent by the XFA application to the web server. This message includes the input message defined by the WSDL description and a [SOAP 1.1] envelope around it.

Example 13.8 Data description packet for the example

```
<dd:dataDescription
   xmlns:dd="http://ns.adobe.com/data-description/"
   dd:name="DataConnectiongetTradePriceRequestDD">
   <TradePriceWS>
      <soap:Body
         xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
         <tns:TradePriceRequest>
            <tns:tickerSymbol/>
         </tns:TradePriceRequest>
      </soap:Body>
   </TradePriceWS>
</dd:dataDescription>
```

This data description matches the input message `GetLastTradePriceInput` from the WSDL service description. Note that the data description merely describes the message format. It does not define the binding of data in the Data DOM to the message. This binding is performed by the individual connect children of fields, exclusion groups, and/or subforms in the Form DOM.

This data description does not include a schema for the output message. This is not required because XFA does not need a schema to import data.

Messages Represented in the Connection Data DOM

The structure of data in the Data DOM is unlikely to match the required structure for input and output messages, more so because a single form may exchange data with any number of web services. In addition the messages are wrapped in SOAP envelopes which may also contain information of interest. Therefore the input message is sent from, and the output message received into, a separate DOM called the Connection Data DOM. The Connection Data DOM is located under the node `xfa.datasets.connectionData` (or equivalently `!connectionData`).

Associating Fields and Exclusion Groups with Nodes in the Connection Data DOM

Fields and exclusion groups can have connect children that associate them with particular nodes in the Connection Data DOM. The connect children control the movement of data between the Data DOM and the Connection Data DOM, in either or both directions. Each connect child controls only that particular data value which is bound to its parent.

In the example there are two wrapper subforms involved in data transfer. The subform input and its contents provide the input message, while output and its contents display the output message. The section of the template containing input is reproduced below.

Example 13.9 Portion of the template controlling the input message

```
<subform name="input" ...>
   <subform name="query">
      <field name="stockTickerSymbol" ...>
         ...
         <connect
            connection="TradePriceWS"
            ref="!connectionData.TickerPriceWS.Body.TradePriceRequest.tickerSymbol"
            usage="exportOnly"/>
      </field>
      ...
   </subform>
   ...
</subform>
```

For each `connect` element, the `connection` attribute identifies a `wsdlConnection` in the `connectionSet`. The connect element has no effect except when the XFA processor is exchanging data with the service represented by that particular connection.

The `ref` attribute is a modified SOM expression that identifies the location of the node in the Connection Data DOM corresponding to the subform, field, or exclusion group. Note that ref must point to a node inside the Connection Data DOM.

When the SOM expression is fully qualified, it is a standard SOM expression. Hence, in the example, the contents of the field are mapped to the node `xfa.datasets.connectionData.TickerPriceWS.Body.TradePriceRequest.tickerSymbol`, which is in the Connection Data DOM. The name `Body` in this SOM expression refers to a SOAP element that contains the message to be sent inside the SOAP envelope.

When the SOM expression is relative, the base location ("$") is inherited from the `connect` child of the enclosing container, instead of being the location in the Form DOM of the container that asserts the SOM expression. Consider the following modified template fragment for input.

Example 13.10 Previous example modified to take advantage of connect-relative SOM resolution

```
<subform name="input" ...>
   <connect
      connection="TradePriceWS"
      ref="!connectionData.TickerPriceWS.Body"
      usage="exportOnly"/>
   <subform name="query">
      <connect
         connection="TickerPriceWS"
         ref="TradePriceRequest"
         usage="exportOnly"/>
      <field name="stockTickerSymbol" ...>
         ...
         <connect
            connection="TickerPriceWS"
            ref="tickerSymbol"
            usage="exportOnly"/>
      </field>
      ...
   </subform>
   ...
</subform>
```

In this example, the input subform has a ref property asserting a fully-qualified SOM expression "`!connectionData.TickerPriceWS.Body`". Because this is fully-qualified it does not matter what value of "$", if any, it inherits from its parent. The expression resolves to `xfa.datasets.connectionData.TickerPriceWS.Body`. Its child subform `TradePriceRequest` inherits the node `xfa.datasets.connectionData.TickerPriceWS.Body` as its base location. The `TradePriceRequest` subform in turn has a `connect.ref` asserting a relative SOM expression, "`TradePriceRequest`". This combines with the inherited base, resolving to `xfa.datasets.connectionData.TickerPriceWS.Body.TradePriceRequest`. This resolved node in turn becomes the base location inherited by the field `stockTickerSymbol`, which is the child of input. The field has a `connect.ref` asserting the relative SOM expression "`tickerSymbol`". This combines with the inherited base to resolve into `xfa.datasets.connectionData.TickerPriceWS.Body.TradePriceRequest.tickerSymbol`. Hence, the effect is the same as the earlier syntax.

The inheritance mechanism has an important advantage when dealing with arrays of data (sets of sibling data nodes sharing the same name). The rules for resolving SOM expressions allow the expression to leave out some subscripts. When the subscript is omitted from an unqualified reference, the XFA processor uses the subscript of the container in the Form DOM that is asserting the SOM expression. (Or it may use the subscript of an ancestor. See "Inferred Index for Ancestors of the Container" on page 103. This makes it possible to use dynamic subforms. Such a subform is declared just once in the template but allowed to instantiate multiple times in the Form DOM. Using an unqualified SOM expression, each of its instantiations correctly references the data, even though each instantiation uses the same SOM expression. Index inferral does not apply to fully-qualified SOM expressions. The inheritance mechanism makes it possible to use unqualified SOM expressions and thus to take advantage of index inferral.

Data Conditionally Copied Between the Data DOM and the Connection Data DOM

The usage attribute controls whether data is copied from the Data DOM to the Connection Data DOM (`exportOnly`), from the Connection Data DOM to the Data DOM (`importOnly`), or both ways (`exportImport`). The effect of copying data both ways with `exportImport` is to update the data. This is

not required for the example application. Hence, in the example fragments, the field associated with the input message has a `connect.usage` of `exportOnly`.

After the message is marshalled in the Connection Data DOM, but before it is sent, the `preExecute` event triggers. A script activated by `preExecute` can modify the input message before it is sent. It is also acceptable for a `preExecute` script to programmatically copy data out of the Connection Data DOM. For example, the following fragment shows some debug code in a test template.

Example 13.11 Script executing on the preExecute event

```
<field name="PREEXECUTE" ...> ... </field>
   <event activity="preExecute"
      ref="$connectionSet.TickerPriceWS">
      <script>
         PREEXECUTE =
            $xfa.datasets.connectionData.TickerPriceWS.saveXML();
      </script>
   </event>
```

In the example, when the `preExecute` script is invoked, the Connection Data DOM contains:

```
[dataGroup (soap:Body)
   xmlns="http://schemas.xmlsoap.org/soap/envelope/"]
   [dataGroup (tns:TradePriceRequest)
      xmlns="http://example.com/stockquote.wsdl"]
      [dataValue (tns:tickerSymbol) = "stockTickerSymbol"
         xmlns="http://example.com/stockquote.wsdl"]
```

This is exactly equivalent to the input message shown above. The mapping between XML and objects in the Connection Data DOM is the same as the default mapping rules used in the regular Data DOM, as described in "Default Data Mapping Rules" on page 117.

Replying to the Web Server and Error Responses

After the `preExecute` event finishes, the input message in the Connection Data DOM is converted to XML and sent to the web server. The web server replies with the output message, also in XML.

In the example, if the query succeeds, the output message contains the share price for the requested stock. This data is wrapped inside a SOAP envelope. In addition, when the query fails, the element `soap:Fault` is returned. As described in [SOAP1.1], the `soap:Fault` element is a child of the `soap:Body` element. `soap:Fault` contains a fault code and a human-readable (but not localized) fault string. When the query succeeds, the message does not contain a `soap:Fault` element.

If there is a communication error or an error reported by the HTTP protocol, the XFA client is unable to receive the output message. In this case the client generates an error message, clears the Connection Data DOM, and terminates the transaction.

Upon receipt of the output message, the client XFA processor parses it and adds its content to the Connection Data DOM. Nodes that are already present (as parts of the input message) are retained. If a particular node corresponds to content in both the input and output messages, its value is updated in place.

After the output message is added to the Connection Data DOM, the client XFA processor triggers a `postExecute` event. In preparation for the `postExecute` event it copies the fault code and fault string into `$event.soapFaultCode` and `$event.soapFaultString`, respectively. If the query succeeds, these elements are not present in the output message, and the values of `$event.soapFaultCode` and

$event.soapFaultString are both empty strings (""). Note that the event properties $event.soapFaultCode and $event.soapFaultString are only available for the duration of the postExecute event, hence only to scripts activated by the postExecute event. The following fragment illustrates how a script can check for the failure of the query.

Example 13.12 Script executing on the postExecute event

```
<event activity="postExecute"
   ref="$connectionSet.TickerPriceWS">
   <script>
      if ($event.soapFaultCode == "")
      {
         // No fault code. Check for the header:
         // connectionData.TickerPriceWS.Header
         if (connectionData.nodes.namedItem("Header") != null)
            {
            // Header exists - do something with it...
            }
      }
      else
      {
         // display $event.soapFaultString...
      }
   </script>
</event>
```

When the service request fails, the return message may include a faultactor element. This element is not used by the stock quote service and in fact is rarely used. Even when faultactor is used, the XFA processor does not copy the content of faultactor into $event . However the postExecute script can get its contents directly from the Connection Data DOM.

It is also acceptable for a postExecute script to programmatically copy data out of the Connection Data DOM. For example, the following fragment shows some debug code in a test template.

Example 13.13 Script executing on postExecute copies data out of the Connection Data DOM

```
<field name="POSTEXECUTE" ...> ... </field>
<field name="FAULTSTRING" ...> ... </field>
<field name="FAULTCODE" ...> ... </field>
<event activity="postExecute"
   ref="$connectionSet.TickerPriceWS">
   <script>
      POSTEXECUTE = !connectionData.TickerPriceWS.saveXML();
      FAULTCODE = $event.soapFaultCode;
      FAULTSTRING = $event.soapFaultString;
   </script>
</event>
```

After the postExecute script finishes the XFA processor resets the $eventobject, so $event.soapFaultCode and $event.soapFaultString are no longer available.

At this point if a soap:Fault element was returned the XFA processor clears the Connection Data DOM and the transaction is finished. However if no soap:Fault element was received the XFA processor proceeds to import the received data from the Connection Data DOM into the main Data DOM. Note that this version of XFA does not define any way for the postExecute script to prevent the import from

happening. However the script can delete all nodes from the Connection Data DOM, which has much the same effect.

The usage attribute of each connect element controls whether the associated data is copied from the Data DOM to the Connection Data DOM (`exportOnly`), from the Connection Data DOM to the Data DOM (`importOnly`), or both ways (`exportImport`). Note that the same node in the Connection Data DOM can receive exported data from one node in the Data DOM while supplying imported data to another node in the Data DOM, using one `connect.usage` set to `exportOnly` and another set to `importOnly`. This is not necessary for the example template because the web service uses a separate element for the data returned by the query. The section of the template that imports the returned data is.

Example 13.14 Portion of the template handling the output message

```
<subform name="output" ... >
   <subform name="response">
      <field name="sharePrice" ...>
         ...
         <connect
            connection="TickerPriceWS"
            ref="!connectionData.TickerPriceWS.Body.TradePrice.price"
            usage="importOnly"/>
      </field>
      ...
   </subform>
   ...
</subform>
```

After the data has been copied from the Connection Data DOM to the Form DOM and Data DOM the transaction is complete. At this point the XFA processor clears the Connection Data DOM. This prevents any possible interference between consecutive web service transactions and simplifies security analysis.

One thing remains to be arranged. There must be some way to trigger the data exchange with the web service. This can be done using an `execute` child of `event`, as follows.

Example 13.15 Button field initiates web service request

```
<field name="getQuoteBtn" ...>
   ...
   <ui>
      <button ... />
   </ui>
   <event activity="click">
      <execute connection="TickerPriceWS"
         executeType="import"
          runAt="client"/>
   </event>
</field>
```

The field `getQuoteBtn` is a button. When the user clicks on the button, the XFA processor initiates the web service transaction.

The `execute` element has a `runAt` property which specifies whether the transaction is to be initiated on the XFA client, on the XFA server, or both. Note that the XFA server is in no way related to the web service server. The XFA server is the computer that served the template to the XFA client. The web service server may be located somewhere else and have no knowledge of XFA. Hence, `runAt` does not affect the web

service server. Rather it determines whether the XFA client, the XFA server, or both, may act as a client to the web service server.

The `execute` element also has an `executeType` property. This can take the values import and re-merge. When the value is imported, the XFA processor updates the existing nodes in the Form DOM and Data DOM with the values from the Connection Data DOM. However if the value is re-merged, the existing nodes are not updated in place. Instead the Form DOM is cleared and a fresh merge operation is performed between the Template DOM and both the Connection Data DOM and the Data DOM. In this merge operation, as the template is traversed, candidate data for binding is sought not only in the Data DOM but also in the Connection Data DOM. If suitable data is found in the Connection Data DOM, it is appended to the Data DOM. The result is that, if data from the Connection Data DOM can be appended the Data DOM and bound to the Form DOM, it is. But any data in the Connection Data DOM that does not match suitable template structure remains un-copied and is lost when the Connection Data DOM is cleared. The re-merge operation has the advantage that the output message can include dynamic structure (optional elements or variable numbers of occurrences of elements) and the form adapts just as it would to dynamic structure in an ordinary data document. However many web services produce output messages with static structures which are more efficiently processed using import.

Schema and WSDL

The architecture of WSDL transactions in XFA relies on XFA-style data descriptions to determine the structure of the messages exchanged with the server. XFA data descriptions are described in "Data Description Specification" on page 834. However application programmers are more likely to be familiar with the W3C XML Schema schema language [XMLSchema]. As a convenience for application programmers the connection set can hold additional information linking particular data descriptions either to sample data or to schemas expressed in XML Schema. This additional information is optional and not used by XFA processors.

Linking a Data Description to a W3C Schema

The `xsdConnection` element links a data description to a schema expressed in W3C XML Schema language [XMLSchema].

The data description is referred to by name and must be located in the Data Description DOM. If this attribute is not supplied the name of the root subform in the template is used.

The schema is referred to by URI. The URI is contained in an element called, appropriately, `uri`. This URI in turn can point to a user packet in the XDP or to an external schema accessible via, for example, HTTP. The `uri` element must be supplied and non-empty.

Caution: A URI followed to an external resource provides a potential point of attack. Applications should only follow the URI to a resource outside the package if the package as a whole is trusted.

The `xsdConnection` element also contains a `rootElement` element which specifies the starting point within the associated schema. This identifies the outermost element that was used to create the associated data description. This element must be supplied and non-empty.

The following example extends "Example That Illustrates the Web Services Architecture" on page 384 by adding schemas for the trade price request message in W3C Schema language. Two schemas are required because the message combines elements belonging to two different namespaces. The schemas are included as custom packets within the XDP. The associated data description is shown in "Data description packet for the example" on page 388.

Example 13.16 An xsdConnection linking to a schema in the package

```
<xdp …>
   <!-- A schema for the stock quote soap envelope -->
   <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:tns="http://example.com/stockquote.wsdl"
      ID="getTradePriceRequestEnvelopeID"
      targetNamespace="http://schemas.xmlsoap.org/soap/envelope/">
      <import namespace="http://example.com/stockquote.wsdl"
         schemaLocation="#getTradePriceRequestSchemaID"/>
      <xsd:element name="Body">
         <xsd:complexType>
            <xsd:sequence>
               <xsd:element ref="tns:TradePriceRequest"/>
            </xsd:sequence>
         </xsd:complexType>
      </xsd:element>
   </xsd:schema>
   <!-- A schema for the stock quote request -->
   <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:tns="http://example.com/stockquote.wsdl"
      xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
      targetNamespace="http://example.com/stockquote.wsdl"
      ID="getTradePriceRequestSchemaID">
      <import namespace="http://schemas.xmlsoap.org/soap/envelope/"
         schemaLocation="#getTradePriceRequestEnvelopeID"/>
      <xsd:element name="TradePriceWS"/>
         <xsd:complexType>
            <sequence>
               <xsd:element ref="soap:Body"/>
            </sequence>
         </xsd:complexType>
      </xsd:element>
      <xsd:element name="TradePriceRequest">
         <xsd:complexType>
            <xsd:sequence>
               <xsd:element name="tickerSymbol"
            </xsd:sequence>
         </xsd:complexType>
      </xsd:element>
   </xsd:schema>
…
   <connectionSet
      xmlns="http://www.xfa.org/schema/xfa-connection-set/2.4/">
      …
      <xsdConnection name="TradePriceWS"
         dataDescription="DataConnectionTradePriceRequestDD">
         <rootElement>TradePriceWS</rootElement>
         <uri>#getTradePriceRequestSchemaID</uri>
      </xsdConnection>
   </connectionSet>
</xdp>
```

Linking a Data Description to a Sample Document

The `xmlConnection` element links a data description to a sample XML document.

The data description is referred to by name and must be located in the Data Description DOM. If this attribute is not supplied the name of the root subform in the template is used.

The sample is referred to by URI. The URI is contained in an element called, appropriately, `uri`. This URI in turn can point to a user packet in the XDP or to an external schema accessible via, for example, HTTP. The `uri` element must be supplied and non-empty.

Caution: A URI followed to an external resource provides a potential point of attack. Applications should only follow the URI to a resource outside the package if the package as a whole is trusted.

The following example extends "Example That Illustrates the Web Services Architecture" on page 384 by adding a sample document for the trade price request message. The document is included as a custom packet within the XDP. The associated data description is shown in "Data description packet for the example" on page 388.

Example 13.17 An xmlConnection linking to a sample document in the package

```
<xdp …>
   <!-- A sample stock quote request -->
   <tns:TradePriceWS ID="getTradePriceRequestSampleID"
      xmlns:tns="http://example.com/stockquote.wsdl"
      xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
      <soap:Body>
         <tns:TradePriceRequest>
            <tns:tickerSymbol>XXXXXX</tns:tickerSymbol>
         </tns:TradePriceRequest>
      </soap:Body>
   </tns:TradePriceWS>
…
   <connectionSet
      xmlns="http://www.xfa.org/schema/xfa-connection-set/2.4/">
      …
      <xmlConnection name="TradePriceWS"
         dataDescription="DataConnectionTradePriceRequestDD">
         <uri>#getTradePriceRequestSampleID</uri>
      </xmlConnection>
   </connectionSet>
</xdp>
```

Invoking ADO APIs Through the Source Set DOM

The Source Set DOM is used to control interactions with a source of sink of information via the ActiveX® Data Object (ADO) interface. Usually ADO is used to access databases, however it is sometimes used for other purposes. XFA's implementation of the API is strongly oriented to data base access.

It is necessary to employ some scripting to use this facility. The names of objects in the source set DOM for the most part correspond closely to names of objects defined in the ADO API Reference [ADO]. The Adobe XML Form Object Model Reference [FOM] should be consulted for detailed information about the scripting interface. However the objects in the source set DOM are not hard to use and a simple example suffices to illustrate them.

In the example there is a customer database containing a table of contact information. Each row of the contacts table contains first and last names for one contact, and possibly other columns but we are not interested in them. The form displays one row from the table at a time, using one field for each of the first and last names. The fields are editable. There are buttons to open and close the database, to move forward or backward in the table, to update the current record after editing, and so on. Each button operates on the database via a very simple (one-line) script which invokes a method of an object in the source set DOM.

The Source Set Packet

The source set packet for the customer contact example described above follows.

Example 13.18 Source set packet for the customer contact example

```
<sourceSet xmlns="http://www.xfa.org/schema/xfa-source-set/1.0/">
   <source name="FFADOData1" db="customer">
      <connect>
         <connectString>DSN=NoPasswordTest</connectString>
         <user/>
         <password/>
      </connect>
      <command timeout="30">
         <query>
            <recordSet cursorType="static" cursorLocation="client"
                       lockType="optimistic" max="0" bofAction="moveFirst"
                       eofAction="moveLast"/>
            <select>Contacts</select>
            <map from="FirstName" bind="#bind0"/>
            <map from="LastName" bind="#bind1"/>
         </query>
      </command>
      <bind id="bind0" ref="$record.contact.FirstName"/>
      <bind id="bind1" ref="$record.contact.LastName"/>
   </source>
</sourceSet>
```

The elements that make up the source set packet are discussed individually below. They are discussed in the order in which they occur in this example. See "Source Set Specification" on page 843.

The sourceSet element

This is the outer element of the source set document, or source set packet inside an XDP. It must be in the namespace `http://www.xfa.org/schema/xfa-source-set/1.0/` as must the other elements discussed here.

The source element

The source set DOM contains any number of `source` objects, each of which represents an individual connection to a data base or other ADO object. These can be connections to different data bases, or different views of the same data base, or the same view of the same data base. The same piece of data in the data base may be involved in different connections at the same time. It is up to the form creator to understand the data base and ensure that such concurrent connections do not cause bad results.

Each `source` object has a `db` property which identifies the ADO object to which it connects. In the example the target object is the `customer` database, which contains the `contacts` table.

The connect element

Each `source` element contains a single `connect` element. The `connect` element controls the connection to the data base, but not interactions with the individual data records. The `connect` element holds at most one `connectString` element, at most one `user` element, and at most one `password` element.

The connectString element

The value of `connectString` is arbitrary as far as XFA is concerned. The string is simply passed to the data base server at connect time.

The user and password elements

The `user` and `password` elements are available so that the form can supply the required credentials for logon automatically. Clearly there is a security risk in displaying the password in plain text! The password should only be included if the form will have limited circulation. However, it is generally the case that ADO is used only with a corporate LAN or WAN, not on the public Internet. If the user name and/or password are omitted, and the data base requires one or both, the XFA application prompts the user to supply it or them.

The command element

Each `source` element in the source set packet contains one or more `command` elements. In the example there is just one command element, which has a `query` child.

The query element

A `query` element represents a method that maintains a cursor and a record set. The type and characteristics of the cursor are determined by its contained `recordSet` element. The `query` element also contains a `select` element and one or more `bind` elements.

The recordSet Element

The `recordSet` object in the source set DOM has many important properties. For convenience the `recordSet` element from the example is reproduced below.

Example 13.19 The recordSet element from the customer contact example

```
<recordSet cursorType="static" cursorLocation="client"
           lockType="optimistic" max="0"
           bofAction="moveFirst" eofAction="moveLast"/>
```

The `cursorType` property causes a a `static` cursor to be used, which grabs a copy of the record set and does not respond to subsequent changes by other users. Furthermore, updates to the record set are not committed to the data base automatically. They are committed only when the `update` method of the `recordSet` object is invoked.

The `cursorLocation` property here causes the cursor to be located on the client. Where the cursor is located affects the efficiency and responsiveness of the interaction. In this case, because the cursor type is `static`, it is desirable to maintain the cursor on the client.

The `lockType` property governs locking of the data in the database while the client has the record set open. The optimum value for this is determined by the data base design, the cursor type, and the nature of the application.

The `max` property sets a limit for the number of records to be returned in the record set. When the value is zero, as in the example, there is no limit.

The `bofAction` and `eofAction` properties control what the cursor does when a cursor positioning command places the cursor at the beginning or end, respectively, of the record set. Using `bofAction` and `eofAction` the form can auto-insert a new record at the end, pre-position for insertion at the beginning, and so on. With all these options available it is possible to make a form that accesses a database with a minimum of scripting. Indeed in the example all of the scripts associated with each button are single method invocations.

The select element

The `select` element contains a SQL select clause. This defines the set of records that belong to the record set. In this case the selection is all columns from a table called `Contacts`.

The map element

A `map` element connects a column in the data base to a `bind` element in the source set. Note that it can not connect directly to a `bind` element in the template.

In the example there are two columns of interest, `FirstName` and `LastName`.

The bind element

Columns in the record set must be mapped into the Data DOM so that the data can be processed there. In the example there are two columns of interest, `FirstName` and `LastName`, and each is mapped via a binding. Each of the bindings is identified by an XML ID so that a `map` element can make reference to it.

In the example the data in the current record is bound to nodes under `$record`. It is not automatic that query data is located there. Rather the source set dictates where the query data will be located. However it is convenient to use `$record` and this is frequently done. The relevant part of the source set is highlighted below.

Example 13.20 Two bind elements using $record

```
<source name="FFADOData1" db="customer">
   <connect ...>...</connect>
```

```
   <command ...>...</command>
   <bind id="bind0" ref="$record.contact.FirstName"/>
   <bind id="bind1" ref="$record.contact.LastName"/>
</source>
```

For each data item two bindings are necessary, one from the data base into the XFA Data DOM and another from a field into the same node of the Data DOM. The binding of fields to nodes in the Data DOM is controlled by the template, as shown in "Fields bound to columns in the data base" on page 400.

Template features used with the Source Set DOM

Fields bound to columns in the data base

In the example the table (or view) being queried has two columns, for a contact's first and last name. These columns are bound to fields in the form using the `bind` property of each field. A skeleton of the template for the two fields is as follows.

Example 13.21 Fields bound to database columns for the customer contact example

```
<field …>
   <value>
      <text/>
   </value>
   <ui>
      <textEdit/>
   </ui>
   <bind match="dataRef" ref="$record.contact.FirstName"/>
</field>
…
<field …>
   <value>
      <text/>
   </value>
   <ui>
      <textEdit/>
   </ui>
   <bind match="dataRef" ref="$record.contact.LastName"/>
</field>
```

The push buttons

In the example the form has a separate button for each database function. The skeleton for a typical button is shown below.

Example 13.22 Delete button for the customer contact example

```
<field name="FFButton7" …>
   <ui>
      <button/>
   </ui>
   <caption>
      <value>
         <text>Del</text>
      </value>
   </caption>
```

```
   <event activity="click">
      <script>$sourceSet.FFADOData1.delete()</script>
   </event>
</field>
```

When the button in the example is pressed it deletes the current record from the record set. `FFADOData1` is the name of the `source` object which controls the query.

The table below shows all of the buttons in the form. Each of them simply invokes a method of the controlling source object. None of these methods require parameters, because the parameters are supplied by the source set document.

Function Performed by Button	Script associated with button
Make a connection, select a set of records, and create a cursor for that record set.	`$sourceSet.FFADOData1.open()`
Rewind to the beginning of the record set.	`$sourceSet.FFADOData1.first()`
Move backward one record in the record set.	`$sourceSet.FFADOData1.previous()`
Move forward one record in the record set.	`$sourceSet.FFADOData1.next()`
Fast forward to the end of the record set.	`$sourceSet.FFADOData1.last()`
Delete the current record in the record set.	`$sourceSet.FFADOData1.delete()`
Update the current record from the data in the form (which presumably has been edited by the user).	`$sourceSet.FFADOData1.update()`
Reload the current record from the data base.	`$sourceSet.FFADOData1.resync()`
Close the connection to the data base and free the cursor. If updates have been made but not committed with `updateBatch()` they are thrown away.	`$sourceSet.FFADOData1.close()`
Post updated records from the record set to the data base. This commits the changes made via previous calls to the `update()` method.	`$sourceSet.FFADOData1.updateBatch()`
Roll back changes made via previous calls to the `update()` method.	`$sourceSet.FFADOData1.cancelBatch()`
Repeat the query, regenerating the record set.	`$sourceSet.FFADOData1.requery()`
Insert a new record into the record set at the current position, shifting the current and subsequent records down one.	`$sourceSet.FFADOData1.addNew()`
Cancel changes made to the data representing the current row, prior to an invocation of the `update()` method.	`$sourceSet.FFADOData1.cancel()`

Updates and rollbacks

Updates take place in three stages. The first two stages can be cancelled (rolled back) if the changes have not yet been committed to the next stage. The stages are:

- Updating the contents of the current row in the Data DOM by assigning to the data values or by editing associated fields via the UI. This can be cancelled by the `cancel()` method.
- Updating the contents of the current row in the data set using the `update()` method. Multiple rows can be updated this way, one at a time. This can be cancelled by the `cancelBatch()` method.
- Updating the contents of the data base using the `updateBatch()` method. This commits the changes to the data base.

Submitting Data and Other Form Content via E-mail

An XFA processor recognizes that it is required to submit via e-mail message when the URI supplied for a `submit` object begins with the `mailto:` domain.

E-mail submission is in some ways simpler and in some way more complicated than other types of submission. It is simpler because it is by nature a one-way process. No response is expected within a short enough time that a user could reasonably be expected to wait for it. It is more complicated because the e-mail medium is not secure. Even if delivery itself is secure (between mail transfer agents that encrypt their conversation), there is no guarantee that the recipient is who he says he is. Neither is there a guarantee that the sender is who he says he is. XFA supports the use of public-key encryption both to sign the submission and to ensure that the message can only be read by the intended recipient.

An XFA processor carries out the following steps to perform an e-mail submit:

1. Execute all validation scripts that contain a `runAt` attribute set to `client`, `both` or undefined. If this step raises exceptions, the requested content should not be submitted to the server.

2. Execute all `nullTest` checks. If this step raises exceptions, the requested content should not be submitted to the server.

3. Execute all `formatTest` checks. If this step raises exceptions, the requested content should not be submitted to the server.

   Note: Validation, `nullTest` and `formatTest` are executed for the entire form, regardless of the actual content submitted.

4. Collect the specified data into a `connectionData` element underneath `$datasets`.

   Note: When the above step completes, the Connection Data DOM changes state.

5. Examine the form for any `preSubmit` events that contain scripts with a `runAt` attribute set to `client`, `both` or undefined. If such scripts are found, execute them. Scripts executed at this point have the chance to examine and alter the data before it is submitted. If this step raises exceptions, the requested content may still be submitted to the server. ("DOM Events" on page 336)

6. Adds, verfies, and/or removes signatures under the control of the `submit` object's `signData` children. There can be multiple children so that multiple signatures can be manipulated at once.

7. Encrypts the data under the control of the `submit` object's `encrypt` child.

8. Generates and sends the e-mail message.

Signing and encryption operations operate only on the data under the `connectionData` object.

Each `signData` object causes one of the following operations:

`verify`

Causes the signature of the data to be verified. If the verification fails, the submission process is canceled and the XFA processor issues a message indicating why the submission failed. This operation is performed before any signature is added or cleared.

`clear`

Causes the signature, if it exists, to be removed from the data. This operation is performed before any signature is created.

sign

Adds a signature to the data.

The `encrypt` object has a `format` property that selects the type of envelope to be used, a PDF envelope or an XML envelope. If PDF is selected the data is inserted as an attachment into an encrypted PDF file. If XML is selected the data is encrypted using W3C XML encryption [XMLEncryption] and placed inside an XML wrapper.

Note: Acrobat 8.0 does not support the use of XML encryption or an XML wrapper. It is limited to inserting the data as an attachment inside an encrypted PDF file.

The wrapper with the encrypted, signed data inside is sent as an attachment to an e-mail message. The addressee of the e-mail (the `To:` field) is the `submit` URI with `mailto:` domain name stripped from the beginning. In an interactive environment the user may be given the option to edit the subject line, the body text, or other aspects of the message before sending it.

Null handling

XFA supports the use of the `xsi:nil` attribute to explicitly mark an element as containing or not containing null data. This is an extension of the notation defined in [XMLSchema]. The use of `xsi:nil` in XFA is described in "Data Values Representing Null Data" on page 125.

14 User Experience

This chapter describes the appearance and behavior or template User Interface objects. It also provides guidance on using such objects to provide accessibility.

Widgets

This section describes the general appearance and behavior of the widgets specified by forms that interface with a user.

As described in "Size, Margins and Borders of Widgets" on page 49, a widget refers to a simulated mechanism displayed by the user interface to enable the user to enter or alter data. For example, a check box displayed on a monitor, which checks or unchecks in response to a mouse click, is a widget.

If a field omits a widget, the XFA processing application selects a widget for the field, as follows:

- Field has data. If the field includes default data or if the field is bound to data, the XFA processing application assigns a widget that reflects the type of data.
- Field is empty. If the field contains neither a default value nor is bound to data, the XFA processing application assigns the text editing widget.

This default behavior is identical to that of the default UI widget, described "Default UI" on page 412.

Barcode Widget

Barcodes are normally used only in non-interactive contexts (printing to paper or facsimile). Usually the barcode field is filled with data by a calculation and is not editable by the user. However XFA does permit barcode fields to be editable. When an editable barcode field gains focus the application displays some sort of edit widget, probably the same edit widget used for ordinary text fields. When the field loses focus the application may display it as a realistically rendered barcode or it may just display a placeholder. It only has to be rendered accurately to paper (or facsimile), not to glass.

There are many different types of barcodes. Here is an example of a common type of barcode, code 3 of 9, displaying the text "1234567890". This barcode is life-sized but may not meet other requirements such as a requirement for whitespace surrounding the barcode.

Button

The button widget (`button`) provides a button interface, usually associated with a click event. When the user selects the button, the click event is activated.

Buttons usually have two appearances, an up appearance and a down appearance. The `highlight` property determines how the two appearances differ. However a button with its `highlight` property set

to `push` has three appearances: up, down, and rollover. Moreover such a button may have a different legend in each state. The down and rollover legends are contained in the field's `items` property and must be named `down` and `rollover`, respectively.

Example 14.1 A button with up, down, and rollover legends

```
<field ...>
   <ui>
      <button highlight="push"/>
   </ui>
   <caption>
      <value><text> Up Text </text></value>
   </caption>
   <items>
      <text name="down"> Down Text </text>
      <text name="rollover"> Rollover Text </text>
   </items>
</field>
```

Starting with XFA 2.5 a button widget in a dynamic form (but not in XFAF) may be an icon button. An icon button displays an image and may also display a text legend. The image is specified in the field default value and the legend in the field caption.

Example 14.2 An icon button

```
<field ...>
   <ui>
      <button highlight="none"/>
   </ui>
   <caption>
      <value><text> Up Text </text></value>
   </caption>
   <value>
      <image contentType="image/jpeg" href="Up_Image.jpg"/>
   </value>
</field>
```

If the `highlight` property of an icon button is set to `push` the button can have three different images and three different legends.

Example 14.3 An icon widget with up, down, and rollover images and legends

```
<field ...>
   <ui>
      <button highlight="push"/>
   </ui>
   <caption>
      <value><text> Up Text </text></value>
   </caption>
   <value>
      <image contentType="image/jpeg" href="Up_Image.jpg"/>
   </value>
   <items>
      <image name="down" contentType="image/jpeg" href="Down_Image.jpg"/>
      <text name="down"> Down Text </text>
      <image name="rollover" contentType="image/jpeg"
```

```
        href="Rollover_Image.jpg"/>
      <text name="rollover"> Rollover Text </text>
    </items>
</field>
```

The button widget is commonly used to trigger a click event that submits a completed form to a server.

Button widgets do not provide a mechanism for entering a value for the parent field.

Check Box and Check Button

The check box or check button widget (checkButton) allows the user to turn on or off the value of its enclosing field. The following are examples of the appearance of a check button widget. Other shapes are available.

When the user selects the check button target, the widget fills in the target to indicate selection, as shown below. The user may select the button by clicking on it or by pressing the enter key when the containing field is in focus.

Check Buttons Not Contained in an Exclusion Group

Check buttons not contained in an exclusion group have the value "on" or "off". They may also be specified as having a third value "neutral".

The check button widget cycles through the allowable values, as the user repeatedly selects the field.

Check Buttons Contained in an Exclusion Group

When multiple fields are contained within an exclusion group and each field contains a check-button widget, those widgets interact so that at most one of the check-buttons within the exclusion group is on.

The check button widget supports only an on or off value for check buttons within an exclusion group, regardless of whether the check button element specifies support for neutral (`allowNeutral="1"`).

Choice List

The choice list widget (choiceList) describes a user interface that provides a set of choices. Some types of choice lists are also called drop-down lists.

Example 14.4 Basic choice list that provides a default and that uses selections as values

```
<field >
  <ui>
    <choiceList open="userControl | onEntry | always | multiSelect"
      commitOn="select | exit"
      textEntry="0 | 1" />
```

```
    </ui>
    <items save="1">
      <text>apples</text>
      <text>bananas</text>
      <text>pears</text>
    </items>
    <value>
      <text>apples</text>
    </value>
</field>
```

The choice list widget has properties that affect how the widget behaves, whether it accepts user input, and when it propagates selected data values to the XFA Data DOM.

Choice List Before Interacting with User

The choice list widget specifies the appearance of the widget before the user interacts with it. The `open` property specifies whether the widget choice list should always be visible (`open="always"` or `open="multiSelect"`). Otherwise, the choice list is hidden until the user selects or otherwise enters the field.

The choice list element may provide a default value, as does Example 14.4. If the choice list provides a default selection, that selection is presented as though the user had selected it, as illustrated on page 410.

User Selects Field

If the choice list is not already visible, selecting the field (or using keyboard sequences to enter the field) causes the choice list to be come visible.

When the user selecting data items from the choice list, those items become highlighted. The highlighting remains until another selection is made, except for choice lists that allow selection of multiple data items.

Whether the User May Select Multiple Data Values

If the choice list allows the user to select multiple data values (`open="multiSelect"`), user may select multiple data, by holding down the shift key.

Whether the User May Provide Data Values

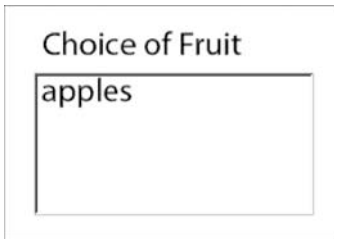
The choice list may allow the user to provide data values (`textEntry="1"`) property allows the widget to specify whether the user may provide data values in place of the values provided by the choice list. Allowing the user to supply data values changes the list of values into a list of hints or prototypes that the user may modify.

The user may not provide data values when the choice list `open` property is set to `"always"` or `"multiSelect"`, regardless of the value of `textEntry`.

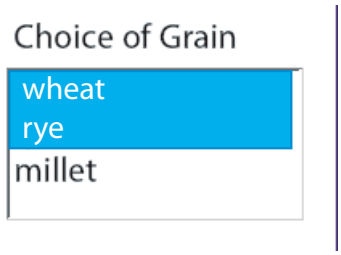
User Exits Field

When the user exits the field (by pressing the enter key, tabbing out, or selecting another field), the choice list reverts to its original presence (where choices are revealed or hidden); however, the selected data remains visible. In the case of choice lists having hidden choices, the chosen value appears in the field; and in the case of choice lists having revealed values, the chosen value or values remain highlighted.

Choice list appearance after selecting data and then exiting the field



A choice list whose list is hidden when not selected.



A multi-select choice list for which multiple items are selected. Such a list is always revealed.

Data Associated with Choice List Selections

A choice list element may use the data displayed in the choice list widget as the value associated with the selection, as does Example 14.4. Alternatively, a choice list element may define values associated with a selection, as shown in Example 14.5.

Example 14.5 Choice list that associates hidden data values with selections

```
<field >
   <ui>
      <choiceList … />
   </ui>
   <items >
      <text>apples</text>
      <text>bananas</text>
      <text>pears</text>
   </items>
   <items save="1" presence="hidden">
      <text>0</text>
      <text>1</text>
      <text>2</text>
   </items>
</field>
```

The non-hidden items ("apples", "bananas", and "pears") appear to the user as possible choices and the hidden items ("0", "1", and "2") are used for data values. The association between non-hidden items and hidden ones is positional. For example, if the user selects "apples", the data value for the field is "0".

When to Update the XFA Data DOM

The choice list element includes an attribute (`commitOn`) that allows the widget to specify whether the data selected in the choice list should be propagated to the XFA Data DOM as soon as the selection is made (`commitOn="select"`)or after focus shifts to another field (`commitOn="exit"`). The former is the default value.

Note: Having a choice list commit data as soon as selections are made may be important in forms that contain non-XFA interactive features, such as Acrobat annotations or hypertext links. People filling out such forms may erroneously believe that selecting an item from a choice list followed by

clicking a non-XFA interactive feature is the same as exiting the check list. In fact, the check list remains the field in focus.

Date/Time Editing Widget

The date-time editing widget (`dateTimeEdit`) helps the user supply a date, a time, or a date-time.

User Selects a Field That Has Date-Time Edit Widget

When the user selects a field or tabs into a field that has a date-time editing widget, the widget displays the field's data using the field's ui picture clause if present. If such a picture clause is not present, the widget displays the canonical format of the data. ("Canonical Format Reference" on page 887).

There is a property of the `dateTimeEdit` object which controls whether, upon selection, it displays a date picker. A date picker makes it easy to enter the current date or a date picked from a calendar-style display of dates. For backwards compatability the default behavior is to refrain from displaying the picker.

It is possible for the field content to exceed the physical size allocated for display. By default a date-time edit widget supports horizontal scrolling to handle this situation. However scrolling may be forbidden by a setting of the `hScrollPolicy` property. In that case when the field is not selected content which extends beyond the physical space is not visible; upon selection the extra content *may* be made available, but whether it is or not is implementation-defined.

User Supplies Data to the Field

After selecting a field, if a date picker is displayed the user may select a date by clicking on the date picker. Alternatively the user may enter date, time, or date-time data for the field as text. The data may be entered as text in any of the following formats:

- As specified by the ui picture clause
- Localized format, as specified in the localeSet picture clause for the locale
- Canonical format

The template may indicate that the field is a date or date-time type by including in the field element a value element which in turn contains a (possibly empty) dateTime element. If this is present and the entered data is incompatible with the field content type, the XFA processing application produces an error response. Alternatively the field may include a validation test or some other script that verifies the data.

The widget puts the data as entered into the `formattedValue` property of the node in the Form DOM that represents the field. It also converts the data into canonical format and puts that into the `rawValue` property of the same node.

Optionally a date-time edit widget may supply comb lines in between character positions to assist the user in entering text.

A date-time edit widget by default supports horizontal scrolling and allows the entry of content that is too long for the phsyical space available. Alternatively the `hScrollPolicy` property may forbid scrolling, in which case the user is prevented from typing or pasting content into the field beyond the physical space allocated.

User De-Selects the Field

After the user de-selects the field (by pressing the enter key, tabbing out, or selecting another field), the field's data is displayed as follows:

- As specified by the field's format picture clause, provided the data matches the picture clause.
- Localized format, using the picture clause specified for the locale in the localeSet, provided the data conforms to the canonical format for the content type. Unless otherwise specified, localization is always done for the current locale.
- Canonical format. If the data does not conform to the canonical format, it is displayed as is.

After the user de-selects the field, the entered data is propagated to the XFA Data DOM.

See "Localization and Canonicalization" on page 138.

Default UI

The default UI widget (`defaultUi`) allows a template to be defined without exact knowledge of the field content type. During form fill-in, the appearance and interaction of the default UI widget is determined by examining the content type of the field. For example, if the content is a number, a numeric editing widget is used. This element can also supply additional hints to a custom GUI via its extras child.

Form designers use the default UI widget when the type of data to be bound to the field is not known at the time the template is created.

External Object Widget

Note: Information on this widget will be provided in a later release of this document.

Image Edit Widget

The image edit widget (`imageEdit`) provides a UI that allows the user supply image data as URI references. When an image edit widget is selected, it presents a browser windows that allows the user to select the specific image desired for the field. when an image edit widget is not selected, it presents the image specified as the field's value.

Numeric Edit

The numeric editing widget (`numericEdit`) helps the user supply an integer, float or decimal number.

User Selects a Field That Has a Numeric Edit Widget

When the user selects or tabs into a field that has a date-time editing widget, the widget displays the field's data using the field's ui picture clause if present. If such a picture clause is not present, the widget displays the canonical format of the data. ("Canonical Format Reference" on page 887)

It is possible for the field content to exceed the physical size allocated for display. By default a numeric edit widget supports horizontal scrolling to handle this situation. However scrolling may be forbidden by a setting of the `hScrollPolicy` property. In that case when the field is not selected content which extends beyond the physical space is not visible; upon selection the extra content *may* be made available, but whether it is or not is implementation-defined.

User Supplies Data

After selecting the field, the user may enter date, time, or date-time data for the field. The data may be entered in any of the following formats:

- As specified by the ui picture clause
- Localized format, as specified in the localeSet
- Canonical format

If the data is incompatible with the field content type, the XFA processing application produces an error response, only if the field includes a validation test or some other script that verifies the data type.

The widget puts the data as entered into the `formattedValue` property of the node in the Form DOM that represents the field. It also converts the data into canonical format and puts that into the `rawValue` property of the same node.

If the user enters an empty string (an enter key without numeric data), the widget sets the value of the field to the empty string.

Optionally a numeric edit widget may supply comb lines in between character positions to assist the user in entering text.

A numeric edit widget by default supports horizontal scrolling and allows the entry of content that is too long for the phsyical space available. Alternatively the `hScrollPolicy` property may forbid scrolling, in which case the user is prevented from typing or pasting content into the field beyond the physical space allocated.

User De-Selects the Field

After the user de-selects the field (by pressing the enter key, tabbing out, or selecting another field), the field's data is displayed as follows:

- As specified by the field's format picture clause, provided the data matches the picture clause.
- Localized format, using the picture clause specified for the locale in the localeSet, provided the data conforms to the canonical format for the content type. Unless otherwise specified, localization is always done for the current locale.
- Canonical format. If the data does not conform to the canonical format, it is displayed as is.

After the user de-selects the field, the entered data is propagated to the XFA Data DOM.

See "Localization and Canonicalization" on page 138.

Password Edit Widget

The password edit widget (`password`) is similar to the text edit widget, except that it is restricted to single-line operation and it displays each character as an asterisk.

Signature Widget

The signature widget (`signature`) allows the user to sign the completed form. This is a signature covering the whole document or part of the document using the PDF signing facility. See "User Experience with Digital Signatures" on page 415.

User in an Interactive Context Clicks on a Signature Widget

A digital signature is generated and added to the document. The appearance of the signature widget changes to indicate that the document has been signed.

There may be multiple signature widgets. They operate independently of each other.

User in an Interactive Context Changes Signed Data

Signing does *not* affect the signing user's ability, or any other user's ability, to modify the data. However when any change is made to the content of any field all signatures covering that field become invalid. All invalid signatures are removed from the document and the appearance of the associated signature widgets changes to indicate that the signatures are missing.

User in a Non-Interactive Context Signs a Printed Form

When the form is printed signature widgets are *not* rendered. The form author may choose to include signature boxes for hand-written signatures, but that is not automatic. One reason for this is that hand-written signatures can not be validated automatically so they do not have an equivalent place in the work flow.

Text Edit Widget

The text editing widget (`textEdit`) allows the user to supply text.

User Selects a Field That Has a Text Editing Widget

Normally the appearance of a text field does not change when the user selects it. Unlike the date, time or date-time content types or the numeric content types, text is not localized.

It is possible for the field content to exceed the physical size allocated for display. By default a text edit widget handles this situation by supporting horizontal scrolling for a one-line field and vertical scrolling for a multi-line field. However horizontal scrolling may be forbidden by a setting of the `hScrollPolicy` property, and similarly vertical scrolling by a setting of the `vScrollPolicy` property. If the appropriate scrolling is forbidden then when the field is not selected content which extends beyond the physical space is not visible; upon selection the extra content *may* be made available, but whether it is or not is implementation-defined.

User Supplies Data

After selecting the field, the user may enter text data for the field. The widget is responsible for enforcing the following text properties:

Property name	Description
`maxChars`	Specifies the maximum number of characters that may be entered in the field. This is the string length, not necessarily the number of characters seen by the user. For example, the user may enter an accented letter which is expressed as two successive Unicode characters in the string.
`allowRichText`	Specifies whether the text may be rich text, which is text that includes HTML styling instructions.
`multiLine`	Specifies whether a line entered in the field may wrap.
`hScrollPolicy`	Specifies whether the text may scroll horizontally to accomodate text too wide for the physical width allocated. If not, and the field is single-line, the user is prevented from typing or pasting content that extends beyond the space allocated.
`vScrollPolicy`	Specifies whether the text may scroll vertically to accomodate a block of text too tall for the physical height available. If not, and the field is multi-line, the user is prevented from typing or pasting content that extends beyond the space allocated.

The widget puts the data into both the `formattedValue` property and the `rawValue` property of the node in the Form DOM that represents the field. In addition if rich text is allowed the corresponding data nodes in the Data DOM are structured as described in "Representing Rich Text in the XFA Data DOM" on page 190.

Optionally a text edit widget may supply comb lines in between character positions to assist the user in entering text. A comb can only be used when the widget is operating in a single-line non-scrollable mode.

User De-Selects the Field

After the user de-selects the field, the entered data is propagated to the XFA Data DOM.

If the content of the field exceeds the physical size allocated for display of the field and the field is not growable, the widget adds a visual clue to the field indicating that some text is hidden from view.

If the text entered in the field exceeds the size of the field and the field is growable ("Growable Containers" on page 238), the widget may interact with the layout processor to resize the field and in the case of flowable layout, to reposition the field. Alternatively the visual representation of the field may be clipped to the size available, provided that when the field regains focus some mechanism is provided to allow access to the full content of the field (for example a scroll bar). Growable fields ignore the values of `hScrollPolicy` and `vScrollPolicy`.

User Experience with Digital Signatures

XFA includes two mechanisms for digitally signing a form: XML digital signatures and PDF digital signatures. A single form may contain both types of signatures, although such use seems unlikely

considering the purpose and level of security provided by each type. These mechanisms and the levels of security they can achieve are described in "Signed Forms and Signed Submissions" on page 464.

Signature Event Produces an XML Digital Signature

A template designer can create a button widget that allows the person filling out the form to apply an XML digital signature to the form ("XML Digital Signatures" on page 471). The person selects the button widget after completing the form. The template designer creates such a button by associating a `signData` action to the button.

A single form may contain multiple signature events, each one signing different or overlapping parts of the form.

XML digital signatures are used to ensure data integrity, as described in "Using Digital Signatures to Achieve Different Levels of Security" on page 464.

Signature Widget Produces a PDF Signature

A template designer uses the signature widget to allow the person filling out the form to create a PDF digital signature. The person would select the signature widget after completing the form. PDF signatures can achieve various levels of security, the most important of which is "integrity". A document with integrity is called a document-of-record. See "PDF Signatures" on page 478.

The PDF digital signature includes the template as well as the data so that it attests to what the user saw and signed. The fact that the document has been signed is indicated in some implementation-defined way in the visual representation of the signature widget. The presence of a signature does not by itself prevent further changes to the data but if any such change is made the signature widget changes in appearance to indicate that the document is no longer signed.

It is possible to apply multiple document-of-record signatures to the same form. Once multiple signatures have been applied, a change to any data invalidates all of the signatures.

Starting with XFA 2.5 a PDF signature may optionally cover only parts of the form, not the entire document. This is the XFA implementation of the *signature digest* introduced into [PDF] in version 1.5.

Accessibility and Field Navigation

XFA templates can specify form characteristics that improve accessibility and guide the user through filling out a field.

- Traversal order. An XFA template may be defined with a traversal order, which allows the user to tab from one field to the next ("Traversal: Tabbing Order and Speech Order" on page 417).
- Accelerator keys. An XFA template may include accelerator keys that allow users to move from field to field, by typing in a control sequence in combination with a field-specific character ("Accelerator Keys: Using Keyboard Sequences to Navigate" on page 421).
- Speech. An XFA template supports speech enunciation, by allowing a form to specify the order in which text descriptions associated with a field should be spoken ("Speech of Text Associated with a Container" on page 421).
- Visual aids. XFA template may specify text displayed when the tooltip hovers over a field or a subform ("Other Accessibility-Related Features" on page 421).

Traversal: Tabbing Order and Speech Order

The traversal (sequencing) capability of XFA forms simplifies navigation between objects on a form and supports speech programs.

The traversal capability simplifies navigation by allowing a person filling out a form to more easily move from one field to the next. The user initiates such movement by typing special characters such as tab, up-arrow, or down-arrow, or by supplying the maximum number of characters allowed in a field.

The traverse capability of XFA forms allows speech programs to advance from one object to the next, after they have completed reading the text associated with the container.

Traversal specifications are provided in `traversal` elements, which may contain a set of `traverse` elements. Traverse specifications are properties of the following types of containers: subforms, exclusion groups, fields, and draws.

Each `traverse` element contains an `operation` attribute that specifies a keystroke or event that triggers traversal, and a traverse destination. The traverse destination is specified as a SOM expression or as a script. In either case, the destination must resolve to a SOM expression for a valid layout object. Default transition directions are defined for any omitted traverse operations.

If traversal order is not explicitly defined, it defaults to a left-to-right and top-to-bottom order.

Explicitly Defined Keystroke-Initiated Traversals

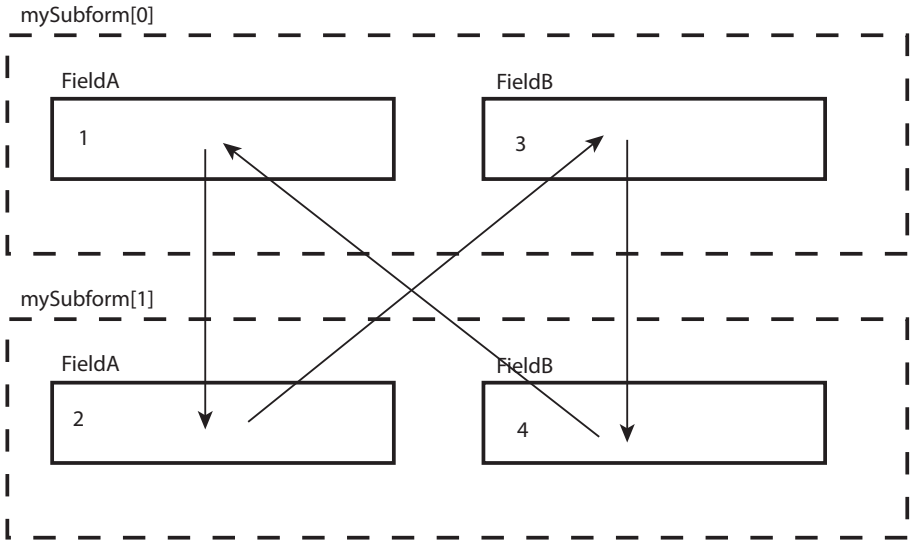
The following example is illustrated on page 418.

Example 14.6 Explicitly-defined traversals

```
<subform name="mySubform" … >
   <field name="A" x="5mm" y="20mm" …>
      <traversal>
         <traverse operation="next" ref="mySubform[+1].A"/>
         <traverse operation="back" ref="mySubform[+1].B"/>
      </traversal>
      <ui … />
      <value … />
   </field>
   <field name="B" x="45mm" y="20mm" …>
      <traversal>
         <traverse operation="next" ref="mySubform[+1].B"/>
         <traverse operation="back" ref="mySubform[+1].A"/>
      </traversal>
      <ui … />
      <value … />
   </field>
</subform>
<subform name="mySubform" … >
   <field name="A" x="5mm" y="120mm"…>
      <traversal>
         <traverse operation="next" ref="mySubform[-1].A"/>
         <traverse operation="back" ref="mySubform[-1].B"/>
      </traversal>
      <ui … />
      <value … />
```

```
    </field>
    <field name="B" x="45mm" y="120mm" …>
      <traversal>
        <traverse operation="next" ref="mySubform[-1].B"/>
        <traverse operation="back" ref="mySubform[-1].A"/>
      </traversal>
      <ui … />
      <value … />
    </field>
</subform>
```

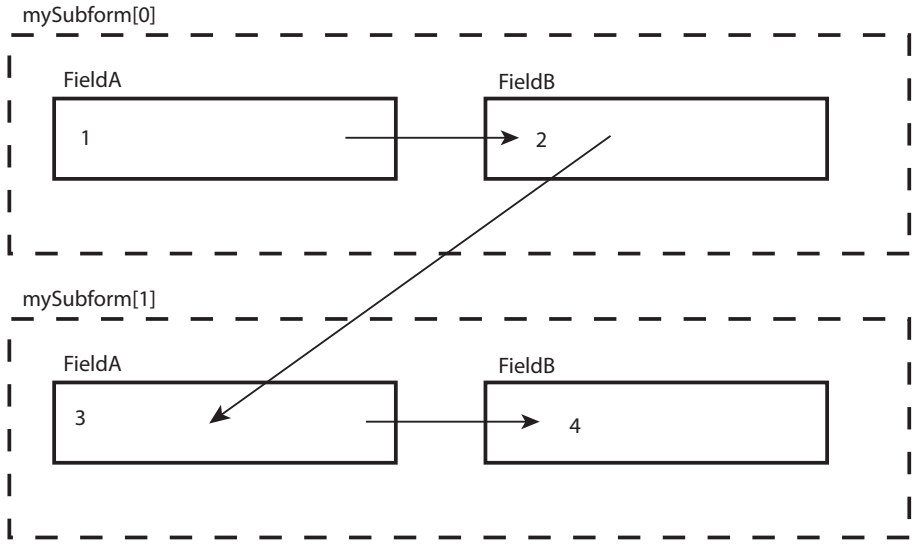
Explicitly-defined traversal



Default Keystroke-Initiated Traversal

Each `traverse` operation contains a default traversal destination. For example, if a container omits a specification for "`next`" (`operation="next"`), the container is assigned a default "`next`" destination relative to the container's position, going left-to-right and top-to-bottom of the current container. For example, if all traversal specifications are omitted from Example 14.6, the traversal direction changes to that shown in the following illustration.

Default traversal



Non-Keystroke Traversals

Non-keystroke traverses are associated only with `traverse` elements that have `operation` values of `"next"` and "`first`". The effect of the latter value is described in "Delegating Focus to Another Container" on page 420.

Traversal Initiated When Maximum Number of Characters Supplied

In the following example, focus changes between fields when the maximum number of characters is entered. This example yields the same behavior shown on page 419.

Example 14.7 Default traversal for fields having a maximum number of characters

```
<subform name="mySubform" … >
   <field name="A" x="5mm" y="20mm" …>
      <ui … />
      <value … >
         <text maxChars=3 … />
      </value>
   </field>
   <field name="B" x="45mm" y="20mm" …>
      <ui … />
      <value … >
         <text maxChars=3 … />
      </value>
   </field>
</subform>
<subform name="mySubform" … >
   <field name="A" x="5mm" y="120mm"…>
      <ui … />
      <value … >
         <text maxChars=3 … />
      </value>
   </field>
   <field name="B" x="45mm" y="120mm" …>
```

```
      <ui … />
      <value … >
         <text maxChars=3 … />
      </value>
   </field>
</subform>
```

Traversal When Speech Application Completes the Current Container

If a speech-capable XFA processing application is presenting an XFA form, the text associated with each container is spoken. The `speak` element describes the order in which a container's text should be spoken. See "Speech of Text Associated with a Container" on page 421.

Traversal Sequences That Support Speech

In order to serve the speech tool, the chain of "next" links may include draw objects. Such objects cannot accept input focus. Therefore, when advancing focus to the next input widget the XFA application continues traversing the chain until it reaches an object that does accept input focus. It is up to the template creator to ensure that the template does not present the XFA application with a non-terminating loop. See also "Speech of Text Associated with a Container" on page 421.

Delegating Focus to Another Container

The traverse element allows a container to delegate focus to another container. When an XFA form is first displayed, focus is assigned to the first subform, per document order. Such focus assignment may be delegated to another container with `operation="first"`, as shown in the following example.

Example 14.8 Delegated focus

```
<subform … >
   <traversal>
      <traverse ref="mySubform.A" operation="first"/>
   </traversal>
   <subform name="mySubform" … >
      <field name="A" … />
   </subform>
</subform>
```

Script Override of Traversal Order

Scripts can set the keyboard focus explicitly via a call to `$host.setFocus()`. This call takes precedence over any traversal that is pending due to the user's actions. For example, consider the following fragment.

Example 14.9 setFocus() overrides traversal

```
<field name="fieldA" …>
   <traversal>
      <traverse ref="fieldB" operation="next"/>
   </traversal>
   <event action="onExit">
      <script>$host.setFocus("fieldC")</script>
   </event>
</field>
<field name="fieldB" …/>
```

```
<field name="fieldC" …/>
```

When the user tabs out of `fieldA`, a traversal to `fieldB` is queued. Next the event script runs with the focus (and current container) still set to `fieldA`. However the event script sets the focus explicitly to `fieldC`. As a side-effect of setting the focus explicitly, the traversal to `fieldB` is discarded without being executed.

The same behavior applies if the user attempts to leave `fieldA` by clicking on another field, or via any other means of interactive navigation.

Accelerator Keys: Using Keyboard Sequences to Navigate

An XFA template may include accelerator keys that allow users to move from field to field, by typing in a control sequence in combination with a field-specific character. Accelerator keys may also be specified for exclusion groups.

Accelerator keys support the following:

- Accessibility for sight-impaired users
- Speed navigation for expert users

The following XFA template segment shows fields for which accelerator keys have been defined.

Example 14.10 Accelerator keys

```
<field name="fieldA" accessKey="A" … />
<field name="fieldB" accessKey="B" … />
<exclGroup name="exclusionGroupC" accessKey="C" … />
```

If a form having the fields and exclusion groups summarized above is opened on a Windows system, typing the keys **"Alt**" and "**A**" takes the user to the field named `fieldA` and typing the the keys **"Alt**" and "**C**" takes the user to the exclusion group named `exclusionGroupC`.

Speech of Text Associated with a Container

XFA template supports speech enunciation, by allowing a form to specify whether the text associated with a container should be spoken and if so, the order in which it should be spoken. This element is a property of the following container elements: subform, field, exclusion group, and draw.

The `speak` element specifies speech order of the following text sources:

- Text defined by the `speak` element itself
- Text defined by the `toolTip` element
- Container caption, although this does not apply to `speak` properties within subform containers
- Container `name`

Other Accessibility-Related Features

Each XFA container may include a role property, which specifies the role played by the container. XFA processing applications can use this property to identify the role of subforms, exclusion groups, fields, and draws. One possible use of this new attribute is to assign it values from the HTML conventions. Such values would declare the role of the parent container, such as `role="TH"` (table headings) and `role="TR"`

(table rows). Such role declarations may be used by speech-enabled XFA processing applications to provide information about a particular container.

In the following example, the `TableHeading` subform sets the `role` property to `TH` to indicate it is a table heading and the `TableRow` subforms sets the role property to `TR` to indicate it is a row. XFA allows the `role` attribute to be any pcdata; however, this example adopts naming conventions described in the *HTML Techniques for Web Content Accessibility Guidelines 1.0*.

Example 14.11 Fragment using assist to declare row and column roles

```
<subform layout="table" columnWidths="1in -1" … >
   <subform layout="row" name="TableHeading" … >
      <assist role="TH"/>
      <draw name="ColumnHeadA" … />
      <draw name="ColumnHeadB" … />
      <draw name="ColumnHeadC" … />
   </subform>
   <subform name="TableRow" … >
      <assist role="TR"/>
      <field name="CellA" … />
      <field name="CellB" … />
      <field name="CellC" … />
   </subform>
</subform>
```

A speech-enabled XFA processing application might include role info in the spoken or tool-top information about a particular container. For example, if the user floats the tool tip over CellA, a speech program might announce information about the heading under which CellA appears.

15 Dealing with Data in Different XML Formats

Although XFA's default data loading mechanism is quite generalized, there are times when it is not appropriate for a particular XML document. XFA provides alternate data loading behaviors, described in "Extended Mapping Rules" on page 423. For most (but not all) of these extended mapping rules the behavior on input and output is symmetrical, so that round-tripping is automatically accommodated. In addition XFA also supports the use of XSLT to transform the data document before it is loaded and/or after it is unloaded, as described in "XSLT Transformations" on page 458.

Extended Mapping Rules

This specification provides a number of rules that are not in effect by default, but are made available by the implementation as overrides or extensions to the default mapping rules. These extended mapping rules may be invoked via configuration settings in the XFA Configuration DOM or via namespaced attributes in the XML data document. The extended mapping rules are described in detail in the following sections.

The following table summarizes the relationship between the extended mapping rules. *Sequence* indicates the order in which the rules are applied to the data as it is loaded from an XML data document into the XFA Data DOM, processed, and subsequently unloaded from the XFA Data DOM into a new XML data document. *Category* is a rough grouping of rules that are similar in character. *Description* indicates the specific rule. *Governed By* indicates what part of the XFA Configuration DOM controls the invocation of the rule.

The phrases in the *Category* column have the following meanings:

Data Binding. Modifies data in the XFA Data DOM but not until data binding is performed

Document Range. Alters what portion of the original XML data document is mapped into the XFA Data DOM during load and back out again during unload

Transform. Modifies data in the XFA Data DOM and possibly the XML data DOM

Sequence	Category	Description	Governed By
1	Document Range	Nominate start element, which is described on page 452	`startNode` element in configuration
2	Document Range	Exclude namespaces	`excludeNS` element in configuration
3	Document Range	Map attributes to `dataValue` nodes	`attributes` element in configuration
4	Transform	Omit content based on element name	`presence` element in configuration with value `ignore` or `dissolve`
5	Transform	Force mapping to a `dataValue` node or `dataGroup` node, as described on page 427	`xfa:dataNode` attribute in XML data document

Sequence	Category	Description	Governed By
6	Transform	Rename nodes based on element or attribute name	`rename` element in configuration
7	Transform	Replace element name with attribute value	`nameAttr` element in configuration
8	Transform	Flatten substructure based on original element name	`presence` element in configuration with value `dissolveStructure`
9	Transform	Trim white space	`whitespace` element in configuration
10	Transform	Modify structure based on empty content	`ifEmpty` element in configuration
11	Transform	Group adjacent data into records	`groupParent` element in configuration
12	Document Range	Nominate record elements	`record` element in configuration
13	Document Range	Exclude records by position	`range` element in configuration
14	Data Binding	Supply default bind picture clause	`picture` element in configuration

For simplicity, the descriptions of some of the extended mapping rules below assume that the XFA Data DOM is first loaded using the default behavior and then the extended mapping rules are applied in subsequent passes by modifying the XFA Data DOM. Conforming implementations may reduce the number of physical passes to optimize performance provided the same end result is obtained.

The extended mapping rules are invoked by supplying the appropriate content to elements in the XFA Configuration DOM. For many of those elements the content may be one of the following standardized keywords:

`preserve`

`preserve` in every case causes the data loader to copy the content as-is from the XML data DOM into the XFA Data DOM. (Note that this description assumes that XML escape sequences such as `"&quot;"` have already been converted to literal Unicode characters in the XML data DOM. The details of an implementation may differ provided the effect is the same.) For most mapping rules `preserve` mimics the default behavior, but there are exceptions, which are described along with the individual mapping rules below.

`ignore`

`ignore` in every case causes the data loader to omit content from the XFA Data DOM but do not make any changes to it in the XML data DOM. If a subsequent unload operation is done, the data unloader inserts the ignored content into the output document at the appropriate place, copying it from the XML data DOM.

remove

`remove` in every case causes the data loader to omit content from the XFA Data DOM and also remove it from the XML data DOM. If a subsequent unload operation is done, the data unloader produces a document without the removed content.

Other keywords and their meanings are defined under the extended mapping rule in which they are used.

Document Range

The term *document* range refers to the extent of the XML data document that is processed by the data loader, such as the whole XML data document or a fragment, as previously described in the section "Default Data Mapping Rules" on page 117.

Some extended mapping rules affect the position and extent of the document range. The set of elements associated with these rules consists of `startNode`, `record`, `incrementalLoad`, and `window`.

Transforms

Some of the extended mapping rules are known collectively as *transforms*. The elements that invoke these rules are only valid inside a `transform` element, as specified by the XFA configuration schema "Config Specification" on page 761. The set of elements associated with transforms consists of `presence`, `whitespace`, `ifEmpty`, `nameAttr`, `rename` and `groupParent`.

The XFA Configuration DOM may include one or more `transform` elements. Each `transform` element has a `ref` attribute. The value of the `ref` attribute determines where the mapping is applied. The value is a list of one or more names, where each name is either an element tag or `"*"`. The `transform` element applies to all data elements with tags that match any of the names in its list. `"*"` is a special pattern that matches all tags. The transform also matches all tags if the value of its ref attribute is an empty string (`""`). The set of data element tags that match a particular `transform` element is referred to here as the *transform set* for that element.

The data loader applies the mapping to elements whose names match (case-sensitive) the value of the `ref` attribute or, if the value of the `ref` attribute is the empty string (`""`), to the entire document. Some transforms apply to all content of the matched element, including sub-elements, whereas other transforms apply only to the element itself and its character data. The scoping in each case is natural to the type of transform. For example, white space trimming transforms, when applied to data values, affect all of the descendants of the `dataValue` node containing data (as opposed to metadata), whereas renaming affects only the node corresponding to the nominated element or attribute itself. However, an element mapped by one `transform` may include an element mapped by the same or a different `transform`. The data loader responds to such nesting by applying the specified transforms sequentially in a depth-first manner. For example, if the transform for an outer element says it is to be discarded but the transform for an inner element says it is to be preserved, the inner element is at first preserved (by its own transform) but then discarded along with the other descendents of the outer element.

Some transforms change the name of the node in the XFA Data DOM that corresponds to the element. These do not affect which later transforms will be applied to the node. Processing is controlled by the original element tag, not the name of the node in the XFA Data DOM.

Note: The value of `ref` must be a simple node name. Readers who are familiar with XFA SOM expressions might assume that SOM expression syntax is valid here, but it is not. As a consequence of this restriction, the transforms within the `transform` element apply to every node in the document range with a matching name and appropriate type, regardless of the node's position in the tree.

Each `transform` element must include at least one operation (child element). There must not be more than one child element of the same name within a single `transform` element.

The data loader does not apply these transformations to any part of the XML data document above the start element, as described in "The startNode Element" on page 452. Likewise it does not apply these transformations to any part of the XFA Data DOM representing parts of the XML data document above the start element. Also, it does not apply these transformations to any part of the XML data document excluded from loading by the namespace rules *except* when carrying out a `remove` directive. When carrying out a `remove`, all descendents of the node to be removed from the XML DOM are also removed, regardless of namespace.

The data loader applies these transformations in the order shown in the table in Extended Mapping Rules. The `ignore` keyword has an effect both when data is loaded into the XFA Data DOM and when it is unloaded into an XML data document. The other transforms only have effects when loading.

It is not recommended for a single XFA Configuration DOM to contain multiple `transform` elements with the same value for `ref`. However if this does happen the data loader selects the highest-precedence transformation of the type currently being processed for a particular data element as shown by the following table. In the table higher precedence is indicated by a higher number.

Transform	Precedence	Option
`presence`	3	`dissolveStructure`
	2	`dissolve`
	1	`ignore`
	0	`preserve`
`whitespace`	3	`normalize`
	2	`trim`
	1	`ltrim, rtrim`
	0	`preserve`
`ifEmpty`	2	`remove`
	1	`ignore`
	0	`dataGroup, dataValue`

The table has no entries for the other transforms because they do not prioritize. Instead when there are conflicting transforms, the data loader uses the one that comes last in the configuration document. Similarly when different transforms have the same type and precedence (for example `ltrim` and `rtrim`) the data loader uses whichever comes last in the configuration document.

For example, suppose the XFA configuration document contains the following.

Example 15.1 Fragment showing transforms of different priorities

```
<transform ref="book">
  <rename>pamphlet</rename>
  <presence>dissolve</presence>
```

```
  <whitespace>trim<whitespace>
</transform>
<transform ref="book">
  <rename>publication</rename>
  <ifEmpty>remove</ifEmpty>
  <whitespace>normalize<whitespace>
  <presence>preserve</presence>
</transform>
```

The order of processing for a `book` element in the XML data document is:

1. the last-occurring `rename` transformation `publication`

1. the higher-precedence `presence` transformation `dissolve`

2. the higher-precedence `whitespace` transformation `normalize`

3. the sole `ifEmpty` transformation `remove`

The XFA Configuration DOM may contain a `transform` element with a `ref` value of `""` (the empty string), which supplies a default transformation for all elements. The data loader applies this transformation to all elements that do not match a `transform` element with a non-empty value for `ref`. For example, consider the following fragment of an XFA configuration document.

Example 15.2 Fragment showing use of ref=""

```
<transform ref="">
   <whitespace>trim</whitespace>
</transform>
<transform ref="address">
   <whitespace>preserve</whitespace>
</transform>
```

causes leading and trailing white space to be trimmed from all `dataValue` node `value` properties except for those originating from `address` elements, which keep their leading and trailing white space.

There can be multiple `transform` elements with a `ref` value of `""`. Conflicts between these are resolved the same way as described above for `transform` elements having identical values for `ref`.

The attributes Element

This section defines an extended mapping rule that can be used to prevent the mapping of XML attributes to `dataValue` nodes.

The XFA Configuration DOM may include an `attributes` element that overrides the default behavior. See "Config Specification" on page 761 for the full schema. The content of the `attributes` element must be one of the following keywords:

attribute keyword	Meaning
`delegate`	Allows an implementation-defined behavior, which may have the effect of the `ignore` or `preserve` keywords.

attribute keyword	Meaning
ignore	The data loader does not map attributes to dataValue nodes.
preserve	The data loader maps attributes to dataValue nodes, except those attributes excluded by some other rule.

Invoking the extended mapping rule with preserve keyword produces the same results as the default mapping behavior for attributes. This default behavior excludes certain attributes that have special meanings. For more information see "Attributes" on page 128.

Attributes may also be excluded from loading based on namespace (as described in "The excludeNS Element" on page 428). The data loader does not load attributes with excluded namespaces regardless of the value of the attributes element in the XFA Configuration DOM.

Invoking the extended mapping rule with the ignore keyword causes the loader to exclude all attributes from the XFA Data DOM. This does not prevent the processing of attributes that have special meanings; they are still processed but are not represented by dataValue nodes in the XFA Data DOM. When the data is unloaded the XFA processor reinserts the attributes preserved in the XML Data DOM. It may reinsert them in a different order than their order in the original XML data document. This is permissible because the XML Specification [XML1.0] dictates that the order of elements is not significant.

Apart from the above exclusions, the behavior specified by the XFA Configuration DOM overrides the default behavior for the entire XML data document.

The excludeNS Element

This specification provides an extended mapping rule to exclude document content from data loader processing by providing one or more XML namespaces that refer to the content intended for exclusion. By default, as described in the section "Namespaces" on page 120, the data loader excludes content belonging to a number of predetermined namespaces.

The XFA Configuration DOM may include an excludeNS element that overrides the default behavior. See "Config Specification" on page 761 for the full schema. The behavior specified by the XFA Configuration DOM overrides the default behavior for the entire XML data document.

The content of the excludeNS element is a white space separated list of Uniform Resource Identifiers as described by [URI]. The data loader excludes elements belonging to any namespace associated with any of these URIs. The data loader also excludes any attributes belonging to any such namespace. Finally, it excludes all attributes belonging to any element, which is itself excluded.

Note that a namespace specified with a namespace prefix is not inherited, whereas the default namespace is inherited. Thus if an element declares a default namespace that is excluded, elements contained within it are excluded by default. However any of the contained elements may declare a different namespace and so escape exclusion. When a containing element is excluded but its contained element is included, the node corresponding to the contained element is appended to the XFA Data DOM at the same point where the node corresponding to the containing element would have been appended had it not been excluded.

In the example below showing a fragment of an XML data document, the default name prefix is declared to be "http://www.example.org/orchard/". This is inherited by most elements and their attributes, however some override the default with the namespace "http://www.example.org/field/", which is signified by the namespace prefix field.

Example 15.3 Data including some portions with non-default namespace

```
<property xml:lang="en">
  <farm xmlns="http://www.example.org/orchard/"
    xmlns:field="http://www.example.org/field/">
    <tree class="pome">apple</tree>
    <tree class="citrus">lemon</tree>
    <field:plant class="tuber">potato</field:plant>
    <tree field:role="border">poplar</tree>
  </farm>
</property>
```

Assume that the XFA Configuration DOM has been set to load attributes into the XFA Data DOM, as described in "The attributes Element" on page 427. Without the `excludeNS` option, the resulting subtree of the XFA Data DOM contains:

```
[dataGroup (property)
  [dataGroup (farm) xlmns="http://www.example.org/orchard/"]
    [dataValue (tree) = "apple"
      xlmns="http://www.example.org/orchard/"]
      [dataValue (class) = "pome" contains="metadata"
        xlmns="http://www.example.org/orchard/"]
    [dataValue (tree) = "lemon"
      xlmns="http://www.example.org/orchard/"]
      [dataValue (class) = "citrus" contains="metadata"
        xlmns="http://www.example.org/orchard/"]
    [dataValue (plant) = "potato" namePrefix="field"
      xlmns="http://www.example.org/field/"]
      [dataValue (class) = "tuber" contains="metadata"
        xlmns="http://www.example.org/orchard/"]
    [dataValue (tree) = "poplar"
      xlmns="http://www.example.org/orchard/"]
      [dataValue (role) = "border" namePrefix="field"
        contains="metadata" xlmns="http://www.example.org/field/"]
```

Note that the `xml:lang` attribute of the `property` element is excluded by the default namespace exclusion rule described in "Namespaces" on page 120. The default namespace exclusions are mandatory and not controlled by the XFA Configuration DOM.

If the `excludeNS` option was used to exclude `http://www.example.org/field/`, the XFA Data DOM would include only those portions shown below in bold.

Example 15.4 Same data as the previous example with highlighting to show inclusions

```
<property xml:lang="en">
 <farm xmlns="http://www.example.org/orchard/"
    xmlns:field="http://www.example.org/field/">
   <tree class="pome">apple</tree>
   <tree class="citrus">lemon</tree>
   <field:plant class="tuber">potato</field:plant>
   <tree field:role="border">poplar</tree>
 </farm>
</property>
```

The namespace declarations on the `farm` element are processed by the data loader but not placed directly into the `dataValue` node corresponding to the element where they are found, as described in "About the XFA Data DOM" on page 109. However, since the `farm` element type does not include a

namespace prefix it adopts the default namespace declared by one of its attributes. The resulting subtree of the XFA Data DOM would contain:

```
[dataGroup (property)]
  [dataGroup (farm) xmlns="http://www.example.org/orchard/"]
    [dataValue (tree) = "apple"
      xlmns="http://www.example.org/orchard/"]
      [dataValue (class) = "pome" contains="metadata"
        xlmns="http://www.example.org/orchard/"]
    [dataValue (tree) = "lemon"
      xlmns="http://www.example.org/orchard/"]
      [dataValue (class) = "citrus" contains="metadata"
        xlmns="http://www.example.org/orchard/"]
    [dataValue (tree) = "poplar"
      xlmns="http://www.example.org/orchard/"]
```

If instead the namespace `"http://www.example.org/orchard/"` was excluded, the XFA Data DOM would include the portions shown below in bold.

Example 15.5 Same data as the previous example with highlighting to show inclusions

```
<property xml:lang="en">
  <farm xmlns="http://www.example.org/orchard/"
    xmlns:field="http://www.example.org/field/">
    <tree class="pome">apple</tree>
    <tree class="citrus">lemon</tree>
    <field:plant class="tuber">potato</field:plant>
    <tree field:role="border">poplar</tree>
  </farm>
</property>
```

The resulting subtree of the XFA Data DOM would contain:

```
[dataGroup (property)]
  [dataValue (plant) ="plant" namePrefix="field"
    xmlns="http://www.example.org/field"
```

Finally, if both `"http://www.example.org/orchard/"` and `"http://www.example.org/field"` were excluded, only the `property` element would remain. As an empty element it would by default be mapped to a `dataValue` node.

```
[dataValue (property) = ""]
```

The groupParent Element

XFA is designed to deal efficiently with data grouped into records. A record consists of a `dataGroup` node with `dataValue` nodes and possibly lower-level `dataGroup` nodes under it. Most XML data documents are already grouped this way, but some are just flat streams of elements. The flat streams nonetheless usually consist of sets of adjacent related elements comprising logical records. This transform provides for grouping logical records under `dataGroup` nodes, hence turning them into XFA-style records. Once grouped the data can then be processed record-by-record rather than all at once, which reduces resource consumption for documents containing many records. This transform also gives more control over the mapping of records into `dataGroup` nodes than the data binding process alone.

For example, a data file contains information about a series of books as a flat series of elements:

```
<items>
```

```
  <ISBN>15536455</ISBN>
  <title>Introduction to XML</title>
  <firstname>Charles</firstname>
  <lastname>Porter</lastname>
  <ISBN>15536456</ISBN>
  <title>XML Power</title>
  <firstname>John</firstname>
  <lastname>Smith</lastname>
</items>
```

Each group of related records begins with an `ISBN` element and ends with a `lastname` element. It would be appropriate to regroup the data so that each group of related records is descended from a unique `book dataGroup` node. This regrouping can be done by the loader. The result is that `dataGroup` nodes are inserted into the Data DOM which are not directly represented in the XML data document. The `dataValue` nodes from the XML data document are placed as children of the inserted groups as follows:

```
[items]
  [book]
    [ISBN = "15536455"]
    [title = "Introduction to XML"]
    [firstname = "Charles"]
    [lastname ="Porter"]
  [book]
    [ISBN = "15536456"]
    [title = "XML Power"]
    [firstname = "John"]
    [lastname = "Smith"]
```

The transformation that accomplishes this regrouping uses the `groupParent` element. The syntax is as follows.

Example 15.6 Transform declaration for grouping example

```
<transform name="ISBN, title, firstname, lastname">
  <groupParent>book</groupParent>
</transform>
```

When the loader encounters an `ISBN` element, it starts a new `book dataGroup` node and begins placing `dataGroup` nodes under it. It continues doing so until it encounters an element that is either not in the current transform set, or is in the set but has already occurred once within the record. Hence in the example it starts a new book `dataGroup` node (`book[0]`) when it encounters the first `ISBN` element, continues the same `dataGroup` node for the subsequent `title`, `firstname`, and `lastname` elements, then closes that `dataGroup` node because the next element (`ISBN`) has already appeared in the current record. Instead it starts a new `dataGroup` node (`book[1]`) and makes the next four `dataValue` nodes children of that `dataGroup` node.

This algorithm makes it possible to correctly group `dataValue` nodes even if individual data values within a group are sometimes missing from the data, or in a different order than they are listed in the transform set. For example suppose the data had been as follows.

Example 15.7 Original data for grouping example

```
<items>
  <ISBN>15536455</ISBN>
  <title>Introduction to XML</title>
  <lastname>Porter</lastname>
```

```
  <title>XML Power</title>
  <ISBN>15536456</ISBN>
  <firstname>John</firstname>
  <lastname>Smith</lastname>
</items>
```

Given the same configuration option, the resulting Data DOM would contain:

```
[items]
  [book]
    [ISBN = "15536455"]
    [title = "Introduction to XML"]
    [lastname ="Porter"]
  [book]
    [title = "XML Power"]
    [ISBN = "15536456"]
    [firstname = "John"]
    [lastname = "Smith"]
```

Note that when *any* `groupParent` transform is present in the configuration, the loader operates in a mode which accepts *only* record data. Any element other apart from the outermost enclosing element that is not listed in a `groupParent` transform set is ignored, that is, not loaded into the Data DOM. By contrast if the outermost enclosing element is not named in a `groupParent` transform set it is loaded in the normal (default) way. For example, suppose the data document above had additional elements as follows.

Example 15.8 Modified data for grouping example

```
<items>
  <reportDate>17 December 2004</reportDate>
  <ISBN>15536455</ISBN>
  <title>Introduction to XML</title>
  <lastname>Porter</lastname>
  <pubDate>August 2001</pubDate>
  <title>XML Power</title>
  <ISBN>15536456</ISBN>
  <pubDate>April 2002</pubDate>
  <firstname>John</firstname>
  <lastname>Smith</lastname>
</items>
```

Given the same configuration option as the previous examples, the resulting Data DOM would contain:

```
[items]
  [book]
    [ISBN = "15536455"]
    [title = "Introduction to XML"]
    [lastname ="Porter"]
  [book]
    [title = "XML Power"]
    [ISBN = "15536456"]
    [firstname = "John"]
    [lastname = "Smith"]
```

In the example, the `items` element is not associated with any `groupParent` but it is the outermost element so it is loaded as a `dataGroup` node in the normal way. However, the `reportDate` element and the two `pubDate` elements are not loaded because they are not associated with any `groupParent` and

they are not the outermost element in the data document. This shows how the transform can be used to exclude unwanted data.

Sometimes a data value appears more than once within a single record. This can be handled simply by repeating the name of the data value in the transformation set. The loader accepts one child by that name for each occurrence of the name in the transformation set. For example, suppose each group of book data may contain up to two `comment` elements, as follows.

Example 15.9 Data for grouping with up to two comment elements per group

```
<items>
  <ISBN>15536455</ISBN>
  <title>Introduction to XML</title>
  <firstname>Charles</firstname>
  <lastname>Porter</lastname>
  <comment>new</comment>
  <ISBN>15536456</ISBN>
  <title>XML Power</title>
  <firstname>John</firstname>
  <lastname>Smith</lastname>
  <comment>used</comment>
  <comment>good condition</comment>
</items>
```

The grouping transform is as follows:

Example 15.10 Transform declaration for up to two comment elements per group

```
<transform name="ISBN, title, firstname, lastname, comment, comment">
  <groupParent>book</groupParent>
</transform>
```

When loaded into the Data DOM, the result is:

```
[items]
  [book]
    [ISBN = "15536455"]
    [title = "Introduction to XML"]
    [firstname = "Charles"]
    [lastname ="Porter"]
    [comment ="new"]
  [book]
    [ISBN = "15536456"]
    [title = "XML Power"]
    [firstname = "John"]
    [lastname = "Smith"]
    [comment = "used"]
    [comment = "good condition"]
```

The regrouping facility can also deal with a mixture of record types. Suppose the data file contained the following mixture of book and CD records.

Example 15.11 Data for grouping with a mixture of group types

```
<items>
  <ISBN>15536455</ISBN>
  <title>Introduction to XML</title>
```

```
  <firstname>Charles</firstname>
  <lastname>Porter</lastname>
  <cdid>4344-31020-2</cdid>
  <title>Big Calm</title>
  <artist>Morcheeba</artist>
  <ISBN>15536456</ISBN>
  <title>XML Power</title>
  <firstname>John</firstname>
  <lastname>Smith</lastname>
</items>
```

A separate regrouping transform must be defined for the `cd dataGroup` node, as follows.

Example 15.12 Transform declarations for mixed group types

```
<transform name="ISBN, title, firstname, lastname">
  <groupParent>book</groupParent>
</transform>
<transform name="cdid, title, artist">
  <groupParent>cd</groupParent>
</transform>
```

The result of applying both these transforms when loading the Data DOM is:

```
[items]
  [book]
    [ISBN = "15536455"]
    [title = "Introduction to XML"]
    [firstname = "Charles"]
    [lastname ="Porter"]
  [cd]
    [cdid = "4344-31020-2"]
    [title = "Big Calm"]
    [artist = "Morcheeba"]
  [book]
    [ISBN = "15536456"]
    [title = "XML Power"]
    [firstname = "John"]
    [lastname = "Smith"]
```

Note that `title` appears as a member of both the `book` group and the `cd` group. This is acceptable and does not cause any problems as long as `title` is not the first element in its logical record. If it is the first element in its record, the loader assigns it to the group parent defined by the first transform in document order that has the element tag in its transform set and that defines a group parent. Note that it does not matter whether or not `title` is the first name in its transform set. Order in the transform set is never significant.

Grouping transforms can nest to any level. This is done by including the name of the parent for the inner group in the transform set for the outer group. For example we wish to group the `firstname` and `lastname` elements under an `author dataGroup` node, which is itself a child of the `book dataGroup` node. This is expressed as follows:

```
<transform name="ISBN, title, author">
  <groupParent>book</groupParent>
</transform>
<transform name="firstname, lastname">
  <groupParent>author</groupParent>
```

```
</transform>
```

Assume the book data is:

```
<items>
  <ISBN>15536455</ISBN>
  <title>Introduction to XML</title>
  <firstname>Charles</firstname>
  <lastname>Porter</lastname>
  <ISBN>15536456</ISBN>
  <title>XML Power</title>
  <firstname>John</firstname>
  <lastname>Smith</lastname>
</items>
```

When the book data is loaded using these transformations, the result is:

```
[items]
  [book]
    [ISBN = "15536455"]
    [title = "Introduction to XML"]
    [author]
      [firstname = "Charles"]
      [lastname ="Porter"]
  [book]
    [ISBN = "15536456"]
    [title = "XML Power"]
    [author]
      [firstname = "John"]
      [lastname = "Smith"]
```

There is one important difference between grouping data values and grouping data groups. `dataGroup` nodes are allowed to repeat an unlimited number of times as siblings under the same parent, even though the name of the data group appears only once in its transform set. This rule is appropriate for most data documents. For example, consider the set of transforms defined for the previous example. The `author dataGroup` node contains at most two data values, `firstname` and `lastname`. It would not be appropriate to accept another `firstname` or `lastname` into the record containing the author's name. Similarly, in this database, each book can have only one ISBN and one title. However, a book can have any number of authors. For example, given the following data:

```
<items>
  <ISBN>15536455</ISBN>
  <title>Introduction to XML</title>
  <firstname>Charles</firstname>
  <lastname>Porter</lastname>
  <firstname>Elizabeth</firstname>
  <lastname>Matthews</lastname>
  <ISBN>15536456</ISBN>
  <title>XML Power</title>
  <firstname>John</firstname>
  <lastname>Smith</lastname>
</items>
```

The result upon loading is:

```
[items]
  [book]
```

```
    [ISBN = "15536455"]
    [title = "Introduction to XML"]
    [author]
      [firstname = "Charles"]
      [lastname ="Porter"]
    [author]
      [firstname = "Elizabeth"]
      [lastname ="Matthews"]
  [book]
    [ISBN = "15536456"]
    [title = "XML Power"]
    [author]
      [firstname = "John"]
      [lastname = "Smith"]
```

There is a limitation to the type of data that this transform can handle. The data must not contain nested `dataValue` nodes. (This implies that it must not contain rich text.) To deal with this use, the `dissolveStructure` option of the `presence` transform, described under "The presence Element" on page 442, to ensure the data is flattened before the `groupParent` transform processes it.

The ifEmpty Element

This section describes an extended mapping rule that can be used to modify the handling of elements that are empty in the XML data document. This handling affects the XFA Data DOM and may optionally alter the XML data DOM.

By default, as described in "Data Values Containing Empty Elements" on page 124, the data loader represents empty elements in the XML data document with `dataValue` nodes in the XFA Data DOM. This extended mapping rule provides for alternate behaviors in which the `dataValue` node is removed from the Data DOM(s) or is converted to a `dataGroup` node.

For purposes of this specification:

- A `dataGroup` node is considered empty if and only if it has no children.
- A `dataValue` node is considered empty if and only if it has no children and its `value` property is equal to the empty string (`""`).

As described in "Data Values Containing Mixed Content" on page 123, a `dataValue` node representing mixed content has a `value` property equal to the ordered concatenation of the `value` properties of its content-containing children. Hence it can have a `value` property equal to the empty string if all of its children also have `value` properties equal to the empty string. However, even in this case it is not considered empty because it has children.

Similarly a `dataValue` node representing an element with no content has a `value` property equal to the empty string, but if it has a child representing an attribute it is not considered empty. Attributes are not loaded by default but they may be loaded under control of a configuration option, as described in "The attributes Element" on page 427.

Consider the following XML data document:

```
<item>
  <book>
    <ISBN registered="no"></ISBN>
    <title>Introduction to XML</title>
    <author>
```

```
      <firstname></firstname>
      <lastname></lastname>
    </author>

  </book>
</item>
```

The elements `firstname` and `lastname` are considered empty. The `author`, `book` and `item` elements are not empty because each of them has children. Assuming attributes were not loaded the element `ISBN` is empty, however if attributes were loaded it is not empty.

The XFA Configuration DOM may include one or more `transform` elements, each of which may include an `ifEmpty` element that overrides the default behavior. See "Config Specification" on page 761 for the full schema. If an `ifEmpty` element is present and the corresponding node is empty, the behavior specified by the `ifEmpty` element overrides the default behavior within the scope of the `transform` element.

The `ifEmpty` element must contain one of `ignore`, `remove`, `dataGroup`, or `dataValue`, where:

- `ignore` causes the data loader to remove the node representing the element from the XFA Data DOM, unless it is the root node. This does not affect the XML data DOM.
- `remove` causes the data loader to remove the node representing the element from the XFA Data DOM, unless it is the root node. In addition, if that node was removed from the XFA Data DOM, the data loader removes the node corresponding to the element from the XML data DOM, so that if a new XML data document is subsequently generated it does not contain the element.
- `dataGroup` causes the data loader to replace the `dataValue` node in the XFA Data DOM with a `dataGroup` node, unless the node's parent is a `dataValue` node. This does not affect the XML data DOM.
- `dataValue` causes the data loader to retain the `dataValue` node representing the element.

Invoking the extended mapping rule with `dataValue` produces the same results as the default behavior for empty elements as described in section"Data Values Containing Empty Elements" on page 124.

As described above, an `ifEmpty` operation will be suppressed if it is trying to remove the root node or if it is trying to make a `dataGroup` node the child of a `dataValue` node. When an operation is suppressed in this way, the data loader should not issue an error message because the same operation may operate legitimately on elements with the same name elsewhere in the XML data document, hence this is expected to be a common occurrence.

Consider the following XML data document:

```
<item>
  <book>
    <ISBN></ISBN>
    <title>Introduction to XML</title>
    <author>
      <firstname></firstname>
      <lastname></lastname>
    </author>

  </book>
</item>
```

With `dataValue` empty element handling, expressed in the XFA configuration document as:

```
<transform ref="ISBN">
  <ifEmpty>dataValue</ifEmpty>
</transform>
<transform ref="firstname">
  <ifEmpty>dataValue</ifEmpty>
</transform>
<transform ref="author">
  <ifEmpty>dataValue</ifEmpty>
</transform>
```

the `author` `dataGroup` node is not changed to a `dataValue` node because it has children, hence is not empty. The result of the mapping is as follows:

```
[dataGroup (item)]
  [dataGroup (book)]
    [dataValue (ISBN)  = ""]
    [dataValue (title) = "Introduction to XML"]
    [dataGroup (author)]
      [dataValue (firstname) = ""]
      [dataValue (lastname)  = ""]
```

The above mapping is identical to that produced by default empty element handling, however it has a higher priority. Default empty element handling can be overridden by forced data group mapping as described below in "The xfa:dataNode Attribute" on page 456. By contrast `dataValue` empty element handling takes precedence over forced data group mapping. This is a consequence of coming earlier in the processing sequence.

With `dataGroup` empty element handling, expressed in the XFA configuration document as:

```
<transform ref="ISBN">
  <ifEmpty>dataGroup</ifEmpty>
</transform>
<transform ref="firstname">
  <ifEmpty>dataGroup</ifEmpty>
</transform>
<transform ref="author">
  <ifEmpty>dataGroup</ifEmpty>
</transform>
```

the result of the mapping in the XFA Data DOM is as follows:

```
[dataGroup (item)]
  [dataGroup (book)]
    [dataGroup (ISBN)]
    [dataValue (title) = "Introduction to XML"]
    [dataGroup (author)]
      [dataGroup (firstname)]
      [dataValue (lastname)  = ""]
```

With `ignore` empty element handling, expressed in the XFA configuration document as:

```
<transform ref="ISBN">
  <ifEmpty>ignore</ifEmpty>
</transform>
<transform ref="firstname">
  <ifEmpty>ignore</ifEmpty>
</transform>
<transform ref="author">
```

```
  <ifEmpty>ignore</ifEmpty>
</transform>
```

the `author dataGroup` node is not ignored because it has a child and is therefore not empty. Hence the result of the mapping in the XFA Data DOM is as follows:

```
[dataGroup (item)]
  [dataGroup (book)]
    [dataValue (title) = "Introduction to XML"]
    [dataGroup (author)]
      [dataValue (lastname)  = ""]
```

If on the other hand the XFA configuration document contains:

```
<transform ref="ISBN">
  <ifEmpty>ignore</ifEmpty>
</transform>
<transform ref="firstname">
  <ifEmpty>ignore</ifEmpty>
</transform>
<transform ref="lastname">
  <ifEmpty>ignore</ifEmpty>
</transform>
<transform ref="author">
  <ifEmpty>ignore</ifEmpty>
</transform>
```

the `firstname` and `lastname dataValue` nodes are both deleted from the XFA Data DOM, leaving the `author dataGroup` node empty, so it in turn is deleted. This results in the following mapping in the XFA Data DOM:

```
[dataGroup (item)]
  [dataGroup (book)]
    [dataValue (title) = "Introduction to XML"]
```

The data loader ensures that the order of declaration of `ifEmpty` rules in the configuration document does not affect the outcome of empty node processing. One way to do this is to perform an `ifEmpty` pass over the XFA Data DOM in bottom-up order, that is to perform the `ifEmpty` processing on the return back from a leaf node towards the root.

Alternatively `ignore` can be applied to the whole document as follows:

```
<transform ref="">
  <ifEmpty>ignore</ifEmpty>
</transform>
```

and in this case the recursive deletion of all empty nodes results in the following mapping in the XFA Data DOM:

```
[dataGroup (item)]
  [dataGroup (book)]
    [dataValue (title) = "Introduction to XML"]
```

With `remove` in place of `ignore`, the result of the mapping in the XFA Data DOM is the same in every case as for `ignore` (above), but for every node deleted from the XFA Data DOM the corresponding node in the XML data DOM is also deleted.

The nameAttr Element

Some XML data documents use the same element type for different families of elements, with an attribute indicating the element family. This style of XML is referred to in the following paragraphs as *inverted* XML. (*Inverted* is not meant pejoratively.) For example, a document that conventionally would contain:

```
<?xml version="1.0"?>
<directory>
   <address>
    <street>10 King</street>
  </address>
</directory>
```

could be expressed in inverted XML as:

```
<?xml version="1.0"?>
<directory>
  <item nodename="address">
    <item nodename="street">10 King</item>
  </item>
</directory>
```

This section describes an extended mapping rule that can be used to alter the handling of data expressed as inverted XML. This affects the XFA Data DOM but has no effect on the XML data DOM.

By default, as described in "Default Data Mapping Rules" on page 117, the `name` property of each node in the XFA Data DOM is copied from the local part of the element type of the corresponding element in the XML data document. This extended mapping rule provides for behaviors in which the `name` property of a nominated element is taken from the value of its nominated attribute.

The XFA Configuration DOM may include one or more `transform` elements, each of which may enclose a `nameAttr` element that overrides the default behavior. See "Config Specification" on page 761 for the full schema. The behavior specified by the `nameAttr` element overrides the default behavior for elements matching the `ref` property of the `transform` element. For each such element that has an attribute with the given name that has a non-empty value, the data loader copies the value of the nominated attribute into the `name` property of the associated node in the XFA Data DOM. For such elements the data loader does not load the nominated attribute as a `dataValue` node even when attribute loading is enabled, as described in "The attributes Element" on page 427; this applies even if the attribute has an empty value.

For the example data above, the following fragment in the XFA configuration document would cause the inverted XML to be represented in the XFA Data DOM the same way as the conventional XML:

```
<transform ref="item">
    <nameAttr>nodename</nameAttr>
</transform>
```

Although this renaming is performed before some other transformations it does not affect which later transformations apply to the renamed node. For example, the following configuration fragment:

```
<transform ref="item">
  <nameAttr>nodename</nameAttr>
  <whitespace>normalize</whitespace>
</transform>
```

causes the data from the `address` elements in the above example to be renamed. It also causes those same nodes to be normalized in whitespace, even though their names no longer match the `ref` value in the `transform` element.

The XFA configuration document must not declare multiple different `nameAttr` mappings for the same value of `ref`. For example, the following fragment from an XFA configuration document illegally declares two different `nameAttr` mappings for `foo`:

```
<transform ref="foo">
  <nameAttr>x</nameAttr>
</transform>
<transform ref="foo">
  <nameAttr>y</nameAttr>
</transform>
```

This is forbidden because it can lead to paradox. For example, consider the above illegal configuration fragment and the following XML data document:

```
<foo x="abc" y="def">some content</foo>
```

If the configuration was legal, the data loader would be required to map the data value simultaneously to both names `abc` and `def`, which is impossible.

If the value supplied by the nominated attribute is not a valid XML node name, the behavior of the data loader is implementation-defined.

Note that this mapping is data-dependent in that the data determines what node names result. Consequently it is not reversible; inverted XML can be loaded into the XFA Data DOM but cannot be copied back into the XML data DOM.

The picture Element

During data binding and after an item of data has been bound to a particular field in the form, the supplied data may be interpreted using a `bind` picture clause. The `bind` picture clause describes the formatting of the data. For example, a `bind` picture clause of `"$999.99"` indicates that the data contains a currency symbol, followed by three digits, followed by a radix indicator, followed by two more digits.

The XFA processor uses `bind` picture clauses to parse the numeric content and make the value available in canonical format to scripting engines. It also uses the same picture clause on output to convert the internal canonical number into the appropriate human-readable format. In the example, if the data string is `"€439,02"` the canonical format is `"439.02"`.

Bind pictures may be supplied by the template; however, if a particular field definition in the template does not supply a bind picture clause, the data binding process looks for a default bind picture clause supplied in the XFA configuration document. If a default clause is supplied, the data binding processor uses it. In either case, the `rawValue` property of the field is set to the supplied value (in the example `"€439,02"`) and the `value` property is set to the canonical string (in the example `"439.02"`). If there is no default clause, both properties are set to the supplied value.

The default picture clause is specified using a `picture` element. The `picture` element must be the child of a `transform` element. The example default picture clause could be declared as follows:

```
<transform ref="prix">
  <picture>$999.99</picture>
</transform>
```

In this case, since the value of the `ref` attribute is `prix`, the default picture clause applies only to data elements named `prix`, after any renaming imposed by other extended mapping rules. As always with picture clauses, if the data does not match the picture clause the picture clause has no effect. For example if the data was `"gratis"` both the `rawValue` and `value` properties of the field would be set to

"gratis". For more information about picture clauses see "Picture Clause Specification" on page 991. For an introduction to localization and canonicalization, see "Localization and Canonicalization" on page 138.

The presence Element

This section describes an extended mapping rule that can be used to flatten or remove parts of the hierarchy of data in the XFA Data DOM and optionally the XML data DOM.

The XFA Configuration DOM may include one or more transform elements, each of which may contain a presence element that overrides the default behavior. See "Config Specification" on page 761 for the full schema. The behavior specified by the presence element overrides the default behavior within the scope described for each keyword below.

The content of the presence element must be one of the keywords described below:

presence value	Directs the data loader to …
preserve	Preserves the original hierarchy.
ignore	Removes the node matching the ref property and also its descendents from the XFA Data DOM but not from the XML data DOM, unless the node is the root node in which case the original hierarchy is preserved.
dissolve	Removes the dataGroup node matching the ref property from the XFA and XML data DOMs, promoting its immediate children to children of its parent, unless the node is the root node in which case it is preserved. Descendents are not affected by this operation.
dissolveStructure	Removes all dataGroup nodes that are descendants of the node matching the ref property from the XFA and XML data DOMs. The matching node itself is not removed. All dataValue nodes that are descendants of the node are promoted to children of the node.

Invoking the extended mapping rule with preserve produces the same results as the default mapping behavior for element names as described in "About the XFA Data DOM" on page 109.

Note the different scope for each keyword. In summary, ignore affects the specified node and its descendents, whereas dissolve affects just the specified node and dissolveStructure affects just the descendents.

Consider the following XML data document:

```
<item>
  <book>
    <ISBN>15536455</ISBN>
    <title>Introduction to XML</title>
    <author>
      <firstname>Charles</firstname>
      <lastname>Porter</lastname>
    </author>

  </book>
</item>
```

With no suppression of data group mapping, expressed in the XFA configuration grammar ["Config Specification" on page 761] as:

```
<transform ref="book">
  <presence>preserve</presence>
</transform>
```

the result of the mapping is identical to the default mapping as follows:

```
[dataGroup (item)]
  [dataGroup (book)]
    [dataValue (ISBN)  = "15536455"]
    [dataValue (title) = "Introduction to XML"]
    [dataGroup (author)]
      [dataValue (firstname) = "Charles"]
      [dataValue (lastname)  = "Porter"]
```

The following example suppresses the `author dataGroup` node and its substructure, expressed in the XFA configuration document as:

```
<transform ref="author">
  <presence>ignore</presence>
</transform>
```

The result of the mapping is as follows:

```
[dataGroup (item)]
  [dataGroup (book)]
    [dataValue (ISBN)  = "15536455"]
    [dataValue (title) = "Introduction to XML"]
```

The following example suppresses the data group mapping of the `book` element, expressed in the XFA configuration document as:

```
<transform ref="book">
  <presence>dissolve</presence>
</transform>
```

The result of the mapping is as follows:

```
[dataGroup (item)]
  [dataValue (ISBN)  = "15536455"]
  [dataValue (title) = "Introduction to XML"]
  [dataGroup (author)]
    [dataValue (firstname) = "Charles"]
    [dataValue (lastname)  = "Porter"]
```

Because `dissolve` also modifies the XML data DOM, when the data unloader creates a new XML data document the new document reflects the `dissolve` operation. In this case the resulting XML data document contains:

```
<item>
  <ISBN>15536455</ISBN>
  <title>Introduction to XML</title>
  <author>
    <firstname>Charles</firstname>
    <lastname>Porter</lastname>
  </author>
</item>
```

The following example suppresses all data group mapping within the book element, expressed in the XFA configuration document as:

```
<transform ref="book">
  <presence>dissolveStructure</presence>
</transform>
```

The result of the mapping is as follows:

```
[dataGroup (item)]
  [dataGroup (book)]
    [dataValue (ISBN)  = "15536455"]
    [dataValue (title) = "Introduction to XML"]
    [dataValue (firstname) = "Charles"]
    [dataValue (lastname)  = "Porter"]
```

Because dissolveStructure also modifies the XML data DOM, when the data unloader creates a new XML data document the new document reflects the dissolveStructure operation. In this case the resulting XML data document contains:

```
<item>
  <book>
    <ISBN>15536455</ISBN>
    <title>Introduction to XML</title>
    <firstname>Charles</firstname>
    <lastname>Porter</lastname>
  </book>
</item>
```

The data loader does not issue a warning when ignore or dissolve is suppressed at the root node, because the same element type may legitimately appear other places in the XML data document.

The range Element

This section describes an extended mapping rule to specify a subset of records for processing.

By default the data loader processes all records in the XML data document. However, the XFA Configuration DOM may contain a range element, which specifies a set of record indices. See "Config Specification" on page 761 for the full schema. When this element is provided and is non-empty, records with indices not included in the list are not processed. Hence the mere existence of the list reverses the default behavior. The behavior specified by the XFA Configuration DOM overrides the default behavior for the entire XML data document.

The content of the range element must be a comma-separated list of record numbers and/or record number ranges, where:

- A record number is a positive base ten integer specifying a record index that is to be included in the set. The index of the first record in the XML data document is 0, of the second is 1, and so on.
- A record number range is a positive base ten integer followed by a hyphen character ("-") followed by another positive base ten integer, specifying a range of record indices that are to be included in the set. The range includes the two numbers given. The two integers may be in either order, smallest-first or largest-first.

All records with indices in the set are processed. The set of indices may extend beyond the index of the last record in the XML data document (indeed it is expected that this will frequently be the case).

Use of this mapping rule does not affect the order in which records are processed, only which ones are included. Therefore the order of processing will be the same as described elsewhere in this specification.

The record Element

This section describes an extended mapping rule to nominate the data group elements within the document range that represent records.

By default, as described in "Record Elements" on page 122, the data loader considers the document range to include one record of data represented by the first data group. This data handling option provides a mechanism for controlling the partitioning of the document into one or more records by nominating data group elements where partitioning occurs. The extent of the document range is not affected by record elements, rather the content within the document range is partitioned by record elements.

Notionally a record is a complete logical unit, which the application processes separately. For example a tax department collects tax returns from many individuals. All of the tax returns could be expressed as a single large XML data document with each individual return comprising one record. The division of a document into records may have side-effects in the way they are processed by the XFA application. It is expected that the application will process records sequentially and separately. For example, it is common to sequentially merge each record with a template, then lay out and print the merged record before moving on to the next record. In this operation the output for each record starts on a new page, even if there is space remaining on the last page of the previous record. In addition, portions of the XFA Data DOM that lie outside any record may passively supply data but in this operation do not force any output to occur.

The XFA Configuration DOM may include a `record` element that overrides the default behavior. See "Config Specification" on page 761 for the full schema. The behavior specified by the XFA Configuration DOM overrides the default behavior for the entire XML data document.

The `record` element nominates `dataGroup` nodes, which are marked as records by setting their `isRecord` properties to `true`. The content of the `record` element must be either a node level or an element type, where:

A node level is an integer that specifies the level in the XML data DOM hierarchy for which the data loader is required to mark each corresponding `dataGroup` node in the XFA Data DOM as a record. The node level is relative to the start node, as follows: the start node is located at level 0, its children are at 1, its grandchildren at 2 and so on.

An element type (tag name) of an element present within the XML data document where the data loader is required to mark each corresponding `dataGroup` node at the same level in the hierarchy with the specified element type (tag name) as a record.

The second case requires some clarification. If an element type is supplied, the data loader traverses the nodes in the XML data DOM beneath the start node in depth-first left-to-right (document) order, until it encounters a node that corresponds to a `dataGroup` node in the XFA Data DOM and for which the local part of the element type matches the supplied string (case-sensitive). The corresponding `dataGroup` node in the XFA Data DOM is marked as a record. Subsequent nodes at the same depth in the XML data DOM (but not necessarily siblings) that also correspond to `dataGroup` nodes and have the same `name` property are also marked as records.

The following XML data document shows a situation in which there are two records but they are not siblings. Assume that in the configuration document the `record` element contains `book`. The result is that the subsections shown in bold print are processed as records.

```
<order>
```

```
  <item>
    <book>
     <ISBN>15536455</ISBN>
     <title>Introduction to XML</title>
    </book>
  </item>
  <item>
    <book>
     <ISBN>15536456</ISBN>
     <title>XML Power</title>
    </book>
  </item>
</order>
```

Note that there are two cases in which a node in the XML data DOM might have the correct name and level but not cause a record to be marked. First, the node in the XML data DOM might be mapped into a `dataValue` node in the XFA Data DOM, but `dataValue` nodes cannot be marked as records because they have no `isRecord` property. Second, the node in the XML data DOM might be excluded from the XFA Data DOM because of its namespace, as described in "The excludeNS Element" on page 428, in which case there is nothing to mark as a record.

It is also worth noting that namespace exclusion may raise the level of a node in the XFA Data DOM relative to the corresponding node in the XML data DOM, when a container element is excluded but not its contained element, as described in "The excludeNS Element" on page 428. The level that matters in determining which `dataGroup` nodes are records is the level in the XML data DOM, *not* the level in the XFA Data DOM. This is appropriate because the record structure is normally expressed in the structure of the XML data document as it is supplied before any exclusions.

Consider the following XML data document:

```
<order>
  <number>1</number>
  <shipto>
    <reference><customer>c001</customer></reference>
  </shipto>
  <contact>Tim Bell</contact>
  <date><day>14</day><month>11</month>
       <year>1998</year></date>
  <item>
    <book>
      <ISBN>15536455</ISBN>
      <title>Introduction to XML</title>
      <author>
        <firstname>Charles</firstname>
        <lastname>Porter</lastname>
      </author>
      <quantity>1</quantity>
      <unitprice>25.00</unitprice>
      <discount>.40</discount>
    </book>
  </item>
  <item>
    <book>
      <ISBN>15536456</ISBN>
      <title>XML Power</title>
```

```
      <author>
        <firstname>John</firstname>
        <lastname>Smith</lastname>
      </author>
      <quantity>2</quantity>
      <unitprice>30.00</unitprice>
      <discount>.40</discount>
    </book>
  </item>
  <notes>You owe $85.00, please pay up!</notes>
</order>
```

With no `record` element in the XFA Configuration DOM, the `order` element is considered to be the one and only record. The `dataGroup` node representing the single record is represented in bold type:

```
[dataGroup (order)]
  [dataValue (number) = "1"]
  [dataGroup (shipTo)]
    [dataGroup (reference)]
      [dataValue (customer) = "c001"]
  [dataValue (contact) = "Tim Bell"]
  [dataGroup (date)]
    [dataValue (day)   = "14"]
    [dataValue (month) = "11"]
    [dataValue (year)  = "1998"]
  [dataGroup (item)]
    [dataGroup (book)]
      [dataValue (ISBN)  = "15536455"]
      [dataValue (title) = "Introduction to XML"]
      [dataGroup (author)]
        [dataValue (firstname) = "Charles"]
        [dataValue (lastname)  = "Porter"]
      [dataValue (quantity)  = "1"]
      [dataValue (unitprice) = "25.00"]
      [dataValue (discount)  = ".40"]
  [dataGroup (item)]
    [dataGroup (book)]
      [dataValue (ISBN)  = "15536456"]
      [dataValue (title) = "XML Power"]
      [dataGroup (author)]
        [dataValue (firstname) = "John"]
        [dataValue (lastname)  = "Smith"]
      [dataValue (quantity)  = "2"]
      [dataValue (unitprice) = "30.00"]
      [dataValue (discount)  = ".40"]
  [dataValue (notes) = "You owe $85.00, please pay up!"]
```

The following example assumes that the `record` element contains `item`.The result of mapping the same XML data document is as follows, with the `dataGroup` nodes representing records appearing in bold type:

```
[dataGroup (order)]
  [dataValue (number) = "1"]
  [dataGroup (shipTo)]
    [dataGroup (reference)]
      [dataValue (customer) = "c001"]
```

```
 [dataValue (contact) = "Tim Bell"]
 [dataGroup (date)]
   [dataValue (day)   = "14"]
   [dataValue (month) = "11"]
   [dataValue (year)  = "1998"]
[dataGroup (item)]
   [dataGroup (book)]
     [dataValue (ISBN)  = "15536455"]
     [dataValue (title) = "Introduction to XML"]
     [dataGroup (author)]
       [dataValue (firstname) = "Charles"]
       [dataValue (lastname)  = "Porter"]
     [dataValue (quantity)  = "1"]
     [dataValue (unitprice) = "25.00"]
     [dataValue (discount)  = ".40"]
[dataGroup (item)]
   [dataGroup (book)]
     [dataValue (ISBN)  = "15536456"]
     [dataValue (title) = "XML Power"]
     [dataGroup (author)]
       [dataValue (firstname) = "John"]
       [dataValue (lastname)  = "Smith"]
     [dataValue (quantity)  = "2"]
     [dataValue (unitprice) = "30.00"]
     [dataValue (discount)  = ".40"]
 [dataValue (notes) = "You owe $85.00, please pay up!"]
```

The expression of data records via a node level is likely not the typical usage of this extended mapping rule, but does have particular utility in specific cases similar to the one illustrated below. Using the previous XML data document as an example, with a `record` element containing `1`, the result of the mapping is as follows:

```
[dataGroup (order)]
  [dataValue (number) = "1"]
 [dataGroup (shipTo)]
   [dataGroup (reference)]
     [dataValue (customer) = "c001"]
  [dataValue (contact) = "Tim Bell"]
 [dataGroup (date)]
   [dataValue (day)   = "14"]
   [dataValue (month) = "11"]
   [dataValue (year)  = "1998"]
 [dataGroup (item)]
   [dataGroup (book)]
     [dataValue (ISBN)  = "15536455"]
     [dataValue (title) = "Introduction to XML"]
     [dataGroup (author)]
       [dataValue (firstname) = "Charles"]
       [dataValue (lastname)  = "Porter"]
     [dataValue (quantity)  = "1"]
     [dataValue (unitprice) = "25.00"]
     [dataValue (discount)  = ".40"]
 [dataGroup (item)]
   [dataGroup (book)]
     [dataValue (ISBN)  = "15536456"]
```

```
      [dataValue (title) = "XML Power"]
      [dataGroup (author)]
        [dataValue (firstname) = "John"]
        [dataValue (lastname)  = "Smith"]
      [dataValue (quantity)  = "2"]
      [dataValue (unitprice) = "30.00"]
      [dataValue (discount)  = ".40"]
  [dataValue (notes) = "You owe $85.00, please pay up!"]
```

As a result of the above mapping, the XML data document is partitioned into four records: `shipTo`, `date`, `item`, `item`. The two `item` records represent the same grouping of data as order items. The other two records `shipTo` and `date` don't represent the same grouping of data as order items, and they don't even relate directly to each other. Given this XML data document, such a mapping is only useful if the processing application is able to discriminate among the `dataGroup` nodes that are of interest. This example illustrates how the expression of data records via a node level can easily produce a mapping of heterogeneous data records.

In cases where the XML data document makes use of different element types for roughly the same grouping of data, the ability to express data records via a node level is very useful, as illustrated by the following example with a `record` element containing `1`.

```
<order>
  <bookitem>
    <ISBN>15536455</ISBN>
    <title>Introduction to XML</title>
    <author>
      <firstname>Charles</firstname>
      <lastname>Porter</lastname>
    </author>
    <quantity>1</quantity>
    <unitprice>25.00</unitprice>
    <discount>.40</discount>
  </bookitem>
  <musicitem>
      <cdid>4344-31020-2</cdid>
      <title>Big Calm</title>
      <artist>Morcheeba</artist>
      <quantity>1</quantity>
      <unitprice>19.00</unitprice>
  </musicitem>
</order>
```

The result of the mapping,with the `dataGroup` nodes representing records appearing in bold type, is as follows:

```
[dataGroup (order)]
 [dataGroup (bookitem)]
    [dataValue (ISBN)  = "15536455"]
    [dataValue (title) = "Introduction to XML"]
    [dataGroup (author)]
      [dataValue (firstname) = "Charles"]
      [dataValue (lastname)  = "Porter"]
    [dataValue (quantity)    = "1"]
    [dataValue (unitprice)   = "25.00"]
    [dataValue (discount)    = ".40"]
 [dataGroup (musicitem)]
```

```
    [dataValue (cdid)      = "4344-31020-2"]
    [dataValue (title)     = "Big Calm"]
    [dataValue (artist)    = "Morcheeba"]
    [dataValue (quantity)  = "1"]
    [dataValue (unitprice) = "19.00"]
```

The XML data document may have content that is outside any record. Although such content is not marked as part of a record, it is nevertheless loaded into the XFA Data DOM. Although it is not inside any record it can still be used in special circumstances, for example if a calculation explicitly makes reference to it. For example, the previous example could be modified with some extra-record data as follows:

```
<order>
  <customername>Delta Books</customername>
  <customerorder>300179</customerorder>
  <bookitem>
    <ISBN>15536455</ISBN>
    <title>Introduction to XML</title>
    <author>
      <firstname>Charles</firstname>
      <lastname>Porter</lastname>
    </author>
    <quantity>1</quantity>
    <unitprice>25.00</unitprice>
    <discount>.40</discount>
  </bookitem>
  <musicitem>
      <cdid>4344-31020-2</cdid>
      <title>Big Calm</title>
      <artist>Morcheeba</artist>
      <quantity>1</quantity>
      <unitprice>19.00</unitprice>
  </musicitem>
</order>
```

When this is loaded into the XFA Data DOM, the additional data is not part of any record but is available for use.

```
[dataGroup (order)]
  [dataValue (customername) = "Delta Books"]
  [dataValue (customerorder) = "300179"]
 [dataGroup (bookitem)]
    [dataValue (ISBN)  = "15536455"]
    [dataValue (title) = "Introduction to XML"]
    [dataGroup (author)]
      [dataValue (firstname) = "Charles"]
      [dataValue (lastname)  = "Porter"]
    [dataValue (quantity)    = "1"]
    [dataValue (unitprice)   = "25.00"]
    [dataValue (discount)    = ".40"]
 [dataGroup (musicitem)]
    [dataValue (cdid)      = "4344-31020-2"]
    [dataValue (title)     = "Big Calm"]
    [dataValue (artist)    = "Morcheeba"]
    [dataValue (quantity)  = "1"]
    [dataValue (unitprice) = "19.00"]
```

The rename Element

This section describes an extended mapping rule that can be used to cause the substitution of new names for names of elements from the XML data document. This substitution affects the XFA Data DOM but does not affect the XML data DOM.

By default, as described in "Default Data Mapping Rules" on page 117, the data loader copies the local parts of element-types from elements into the corresponding `name` properties of nodes in the Data DOM. This extended mapping rule provides for an alternate behavior in which the local (non-namespace) part of each element type is matched against a set of (name, substitute) pairs and, if it is in the set, the substitute string is copied into the `name` property of the node in its place. Matching is case-sensitive in keeping with XML norms. If the local part of the element type does not match any member of the set, it is used verbatim, as in the default case.

The XFA Configuration DOM may include one or more `transform` elements, each of which may include one `rename` elements that override the default behavior. See "Config Specification" on page 761 for the full schema. The behavior specified by the `rename` overrides the default behavior for every element in the XML data document with the local part of the element type matching the `ref` property of the `transform`. It also overrides the default behavior for every attribute in the XML data document with the local part of the name matching the `ref` property of the `transform`, when attributes are being mapped to `dataValue` nodes as described in "The attributes Element" on page 427.

If the `rename` element is non-empty, the data loader sets the `name` property of the corresponding node in the XFA Data DOM to the content of the `rename` element. The content of the `rename` element must be a valid local name as specified by [XMLNAMES].

Note that name mapping only applies to the local part of a name. It is not affected by and does not affect namespace designators.

Consider the following XML data document:

```
<item>
  <book>
    <ISBN>15536455</ISBN>
    <title>Introduction to XML</title>
    <author>
      <firstname>Charles</firstname>
      <lastname>Porter</lastname>
    </author>

  </book>
</item>
```

With name mapping for `author` and `title` elements, expressed in the XFA configuration document as:

```
<transform ref="author">
  <rename>writer</rename>
</transform>
<transform ref="title">
  <rename>bookName</rename>
</transform>
```

the result of the mapping in the XFA Data DOM is as follows:

```
[dataGroup (item)]
  [dataGroup (book)]
```

```
[dataGroup (ISBN)]
[dataValue (bookName) = "Introduction to XML"]
[dataGroup (writer)]
  [dataValue (firstname) = "Charles"]
  [dataValue (lastname)  = "Porter"]
```

Although renaming is performed before some other transformations it does not affect which later transformations apply to the renamed node. For example, the following configuration fragment:

```
<transform ref="address">
  <rename>MailingAddress</rename>
  <whitespace>normalize</whitespace>
</transform>
```

causes data from `address` elements to be renamed to `MailingAddress` in the Data DOM. It also causes those same nodes to be normalized in whitespace, even though their names no longer match the `ref` value in the `transform` element.

Almost any valid XML tag or attribute name (as defined in [XML]) can be used in the XML data document without a name mapping. The only restriction of XFA names which does not also apply to XML names is that XFA names may not contain an embedded `":"` (colon). Fortunately XML such as the following, while legal, is rarely encountered.

```
<outer xmlns:abc="http://example.org/ns/abc/">
   <abc:foo:bar>xxx</png:foo:bar>
</outer>
```

In the example the inner data element has the name `foo:bar`. To map this into theXFA Data DOM it would be necessary to rename the data value. For example,

```
<transform ref="foo:bar">
   <rename>foo_bar</rename>
</transform>
```

The startNode Element

This section describes an extended mapping rule to nominate an element within the XML data document that is the starting point for the data loader. The data loader processes only the fragment of the XML data document consisting of that element and its content.

By default, as described in "Start Element" on page 118, the data loader starts at the outermost element of the document. This causes it to attempt to map the whole of the XML data document. When this extended mapping rule is used, the data loader starts processing at the nominated element. Other document range constraints, such as namespace constraints, still apply. As a result the document range is constrained to be not greater than the fragment within the start element.

The XFA Configuration DOM may include a `startNode` element which overrides the default behavior. See "Config Specification" on page 761 for the full schema. The behavior specified by the XFA Configuration DOM overrides the default behavior for the entire XML data document. The content of the `startNode` element is a string of the form `"xfasom(`*somExpr*`)"` where *somExpr* is a restricted SOM expression. The general syntax of SOM expressions is defined in "Scripting Object Model" on page 73. The expression in the `startNode` element must be a fully-qualified path of element types (tag names) starting with the root of the XML data document and referring to a single element, as illustrated in the following example:

```
<order>
  <number>1</number>
  <shipto>
```

```
    <reference><customer>c001</customer></reference>
  </shipto>
  <contact>Tim Bell</contact>
  <date><day>14</day><month>11</month>
       <year>1998</year></date>
  <item>
    <book>
      <ISBN>15536455</ISBN>
      <title>Introduction to XML</title>
      <author>
        <firstname>Charles</firstname>
        <lastname>Porter</lastname>
      </author>
      <quantity>1</quantity>
      <unitprice>25.00</unitprice>
      <discount>.40</discount>
    </book>
  </item>
  <item>
    <book>
      <ISBN>15536456</ISBN>
      <title>XML Power</title>
      <author>
        <firstname>John</firstname>
        <lastname>Smith</lastname>
      </author>
      <quantity>2</quantity>
      <unitprice>30.00</unitprice>
      <discount>.40</discount>
    </book>
  </item>
  <notes>You owe $85.00, please pay up!</notes>
</order>
```

Assume that the start node has been set with `"xfasom(order.item[1].book)"`. Recall that XFA SOM expressions use zero-based indexing, so `"item[1]"` refers to the *second* instance of `item`. The result of the mapping is as follows:

```
[dataGroup (book)]
  [dataValue (ISBN)  = "15536456"]
  [dataValue (title) = "XML Power"]
  [dataGroup (author)]
    [dataValue (firstname) = "John"]
    [dataValue (lastname)  = "Smith"]
  [dataValue (quantity)  = "2"]
  [dataValue (unitprice) = "30.00"]
  [dataValue (discount)  = ".40"]
```

With a start element expressed as `"xfasom(order.item)"`, the result of the mapping is as follows:

```
[dataGroup (item)]
  [dataGroup (book)]
    [dataValue (ISBN)  = "15536455"]
    [dataValue (title) = "Introduction to XML"]
    [dataGroup (author)]
      [dataValue (firstname) = "Charles"]
```

```
    [dataValue (lastname)  = "Porter"]
  [dataValue (quantity)  = "1"]
  [dataValue (unitprice) = "25.00"]
  [dataValue (discount)  = ".40"]
```

It should be noted that an identical mapping is produced with a start element expressed as "xfasom(order.item[0])".

With a start element expressed as "xfasom(order.item[1])", the result of the mapping is as follows:

```
[dataGroup (item)]
  [dataGroup (book)]
    [dataValue (ISBN)  = "15536456"]
    [dataValue (title) = "XML Power"]
    [dataGroup (author)]
      [dataValue (firstname) = "John"]
      [dataValue (lastname)  = "Smith"]
    [dataValue (quantity)  = "2"]
    [dataValue (unitprice) = "30.00"]
    [dataValue (discount)  = ".40"]
```

The whitespace Element

This section describes an extended mapping rule that can be used to alter the handling of white space in the XML data document. This handling affects the XFA Data DOM and also the XML data DOM.

By default, as described in "White Space Handling" on page 131, white space is preserved in the `value` property of `dataValue` nodes. This extended mapping rule provides for alternate behaviors in which white space is trimmed from both ends, trimmed only on the left, trimmed only on the right, or normalized. Note that both element content and attribute values are represented by `dataValue` nodes, so this operation applies equally to both.

The XFA Configuration DOM may include one or more `transform` elements, each of which may contain a `whitespace` element that overrides the default behavior. See "Config Specification" on page 761 for the full schema. When applied to a `dataGroup` node, this transform has no effect, either upon the `dataGroup` node or upon its children. When applied to a `dataValue` node, this transform overrides the default behavior for the `dataValue` node and its descendents. However, its descendents may have different white space transforms applied to them, and such transforms are applied depth-first as described in "Transforms" on page 425.

The content of the `whitespace` element must be one of the following keywords:

whitespace value	Directs the data loader to …
`trim`	Trim white space from both ends but preserve embedded white space, both in the XFA Data DOM and in the XML data DOM.
`rtrim`	Trim trailing white space but preserve leading and embedded white space, both in the XFA Data DOM and in the XML data DOM.
`ltrim`	Trim leading white space but preserve embedded and trailing white space, both in the XFA Data DOM and in the XML data DOM.

whitespace value	Directs the data loader to …
normalize	Trim leading and trailing white space and replace each instance of embedded white space with a single SPACE character (U+0020), both in the XFA Data DOM and in the XML data DOM.
preserve	Preserve all white space.

When the document contains mixed content, the operation is performed on the complete text of the value property of the outermost data value, then the inner data values are modified as necessary to remain consistent. When the operation is normalize, where within mixed content there is embedded white space at the end of one data value, whether or not the next data value starts with white space, the replacement SPACE character is assigned to the first data value; but where a data value ends in non-white space and the next data value starts with white space, the replacement SPACE character is assigned to the second data value.

For example, consider the following XML data:

```
<book>
  <ISBN>15536455</ISBN>
  <title>Introduction to XML</title>
  <desc>Primer on <keyword>   XML  </keyword>  technology.</desc>
</book>
```

The resulting XFA Data DOM, after default white space handling (preserve), is as follows:

```
[dataGroup (book)]
  [dataValue (ISBN)      = "15536455"]
  [dataValue (title)     = "Introduction to XML"]
  [dataValue (desc)      = "Primer on    XML  technology."]
    [dataValue ()       = "Primer on "]
    [dataValue (keyword) = "   XML  "]
    [dataValue ()       = "  technology."]
```

After the normalize operation no more than one blank separates each word:

```
[dataGroup (book)]
  [dataValue (ISBN)      = "15536455"]
  [dataValue (title)     = "Introduction to XML"]
  [dataValue (desc)      = "Primer on XML technology."]
    [dataValue ()       = "Primer on "]
    [dataValue (keyword) = "XML"]
    [dataValue ()       = " technology."]
```

In the following example two different transforms are defined, rtrim for one element and ltrim for the other.

```
<transform ref="desc">
  <whitespace>rtrim</whitespace>
</transform>
<transform ref="keyword">
  <whitespace>ltrim</whitespace>
</transform>
```

In the XML data document an element that is nominated for rtrim encloses an element nominated for ltrim.

```
<desc>Primer on<keyword> XML </keyword> </desc>
```

The result is that the inner element is first subject to `ltrim` on its own, and then to `rtrim` as content of the outer element. Hence the value of the inner element starts off as `" XML "`, is trimmed on the left to become `"XML "`, and then trimmed on the right as part of the string `"Primer on XML "`, resulting finally in:

```
[dataValue (desc) = "Primer onXML"]
  [dataValue (keyword) = "XML"]
```

The xfa:dataNode Attribute

This specification provides an extended mapping rule that can be used to force the creation of `dataValue` nodes and `dataGroup` nodes from a particular element within the document range. For instance, this is useful in circumstances where the structure of the original XML data document is not considered to be useful information; only the data value content is desired.

Unlike all the other extended mapping rules, this rule is invoked from within the XML data document. The rule may be invoked for any element within the document range by placing a particular attribute in the element. The name of the attribute is `dataNode` and it must belong to the namespace `"http://www.xfa.org/schema/xfa-data/1.0/"`. The attribute is defined as:

`dataNode="dataValue"|"dataGroup"`

where:

dataNode value	Directs the data loader to …
`dataValue`	Map the associated element to a `dataValue` node according the rules defined by this specification.
`dataGroup`	Map the associated element to a `dataGroup` node according the rules defined by this specification.

The data loader does not permit this extended mapping rule to be used to violate the relationships between `dataGroup` nodes and `dataValue` nodes, as described by this specification. For instance, as a result of a forced mapping of an element to a `dataValue` node, the element's contained elements are not considered as candidate data groups, because `dataValue` nodes can be ancestors only to other `dataValue` nodes. Similarly, a forced mapping attempt to a `dataGroup` node does not succeed where the `dataGroup` node would be descended from a `dataValue` node, again because `dataValue` nodes can be ancestors only to other `dataValue` nodes. Any attempt to force the mapping of an element that would violate the relationships between `dataGroup` nodes and `dataValue` nodes is detected by the data loader and the request ignored.

The following examples illustrate the usage of this extended mapping rule.

Consider the following example:

```
<book xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
  <ISBN xfa:dataNode="dataGroup">15536455</ISBN>
  <title>Introduction to XML</title>
  <author xfa:dataNode="dataValue"
><firstname>Charles</firstname
><lastname>Porter</lastname
></author>
  <desc xfa:dataNode="dataGroup">Basic primer on <keyword>XML</keyword
> technology.</desc>
</book>
```

The result of mapping this XML data document is as follows:

```
[dataGroup (book)]
  [dataGroup (ISBN)]
    [dataValue () = "15536455"]
  [dataValue (title)  = "Introduction to XML"]
  [dataValue (author) = "CharlesPorter]
    [dataValue (firstname) = "Charles"]
    [dataValue (lastname)  = "Porter"]
  [dataGroup (desc)]
    [dataValue () = "Basic primer on "]
    [dataValue (keyword) = "XML"]
    [dataValue () = " technology."]
```

In the above example, the XML for the `author` element has been modified from previous similar examples which had the `firstname` and `lastname` on separate lines and indented to aid legibility. This white space was removed for this example in order to produce the mapping above. See the section "White Space Handling" on page 131 for more information on white space handling inside data values.

When this extended mapping rule causes an element containing character data, which would otherwise be mapped to a `dataValue` node, to be mapped instead to a `dataGroup` node, the data loader inserts an unnamed `dataValue` node as child of the `dataGroup` node to hold the character data. For example, given the following XML data document:

```
<book xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
  <ISBN>15536455</ISBN>
  <title xfa:dataNode="dataGroup">Introduction to XML</title>
</book>
```

After loading with default mapping rules the XFA Data DOM contains:

```
[dataGroup (book)]
  [dataValue (ISBN) = "15536455"]
  [dataGroup (title)]
    [dataValue () = "Introduction to XML"
```

A common use for this extended mapping rule is to ensure that an empty element that represents a data group is not mapped to a `dataValue` node as would occur based on the default mapping rules described in the section "Data Values Containing Empty Elements" on page 124. Consider the following example:

```
<book>
  <ISBN>15536455</ISBN>
  <title>Introduction to XML</title>
  <author/>
</book>
```

We know from previous examples that the element `author` is a data group element that usually encloses data value elements `firstname` and `lastname`; however, in this specific example, the `author` element is empty and therefore would, by default, map to a `dataValue` node.

To ensure that the `author` element maps to a `dataGroup` node, the following example uses this extended mapping rule:

```
<book xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
  <ISBN>15536455</ISBN>
  <title>Introduction to XML</title>
  <author xfa:dataNode="dataGroup"/>
</book>
```

Note: This extended mapping rule only overrides the default mapping rule for non-empty elements. It does not override the extended mapping rule for empty elements, which is described in the section "The ifEmpty Element" on page 436 if the extended mapping rule for empty elements is declared in the XFA configuration document.

XSLT Transformations

XSLT is a special-purpose language defined by [XSLT]. An XSLT program or *stylesheet* can be used to transform an input XML document into an output XML or non-XML document. XFA supports the use of XSLT to transform arbitrary XML input data into a temporary XML document, which is then loaded by the XFA loader. XFA also supports the use of XSLT to transform the output XML document into a final arbitrary XML or non-XML document. Note that XSLT is not reversible, hence if it is necessary to round-trip two different XSLT stylesheets are required, one for the input transformation and one for the output transformation.

XSLT Preprocessing

This section describes a facility that can be used to modify the incoming data under control of an XSLT script [XSLT]. This transformation makes changes to the XML data DOM before the XFA Data DOM is loaded by the data loader. Thus, if a new XML data document is created it reflects the result of the XSLT preprocessing.

Note that it is possible to incorporate a processing instruction into the XML data document that causes an XSLT transformation to be applied before (or as) the data is loaded into the XML data DOM. This is quite separate from the facility described here. The facility described here does not require the addition of a processing instruction, or any other modification, to the XML data document.

The XFA Configuration DOM may include an `xsl` element. If present, the `xsl` element must contain a `uri` element that nominates the XSLT script. See "Config Specification" on page 761 for the full schema. When the `xsl` element is supplied, the data loader executes the script and uses its output in place of the original XML data document.

The XFA Configuration DOM may include a `debug` element that nominates a place in which to save a copy of the data after the XSLT transformation but before any other transformations. If the `debug` element is supplied and is non-empty, and the XSLT transformation is performed, the data loader copies the output of the XSLT transformation into the nominated place.

For example, the following fragment from an XFA configuration document causes the script `"massage.xslt"` to be used for preprocessing the XML data document. The preprocessed document is copied to `"massage.xml"` before being loaded by the data loader.

```
<xsl>
  <debug>
    <uri>massage.xml</uri>
  </debug>
  <uri>massage.xslt</uri>
</xsl>
```

XSLT Postprocessing

This section describes an extended mapping rule that can be used to modify data being written to a new XML data document under control of an XSLT script [XSLT].

The XFA Configuration DOM may include an `outputXSL` element. If present, the `outputXSL` element must contain a `uri` element that nominates the XSLT script. See "Config Specification" on page 761 for the full schema. When the `outputXSL` element is supplied, the data loader executes the script after applying all other transformations to the data.

Note that `debug` may not be used inside `outputXSL`.

For example, an XFA Configuration DOM includes the following fragment:

```
<outputXSL>
  <uri>out.xslt</uri>
</outputXSL>
```

This causes the output XML document from the data loader to be passed to an XSLT interpreter, along with the local file `"out.xslt"`, which contains an XSLT style sheet. The style sheet supplies all required additional configuration such as the destination of the transformed document.

16 Security and Reliability

It is important to understand what security XFA does not support. XFA does not guarantee that a form will look the same to everyone who looks at it. Indeed the `relevant` attribute exists purely to make the form look and/or act differently under different circumstances, for example when printed instead of viewed on a display. In addition XFA allows scripting, even in static forms, and scripts can be constructed to be deceptive. In short XFA cannot guarantee that a form does what it appears to do. What XFA can guarantee is described in the topics "Tracking and Controlling Templates Through Unique Identifiers" on page 460, "Protecting an XFA Server from Attack" on page 462, and "Signed Forms and Signed Submissions" on page 464.

Similarly XFA does not guarantee that an archived form will still be accessible and identical when it is retrieved in the future. This is true of PDF generally. However there is a subset of PDF called PDF/A which is reliably archivable. PDF/A omits all XFA content except, optionally, the XML Data Document. When a form using XFA is converted to PDF/A for archiving both the boilerplate and field content are flattened into a PDF appearance stream. This guarantees the appearance of the form but it also voids all digital signatures and removes any evidence of why the form looks like it does. Sometimes a lower level of archivability is acceptable in exchange for keeping the XFA structure. This has the additional benefit of optimizing the portability of the XFA form. This is discussed in "Structuring Forms for Portability and Archivability" on page 480.

Tracking and Controlling Templates Through Unique Identifiers

This section describes how templates can be uniquely identified. It also explains the requirements XFA template designing applications and XFA processing applications must satisfy to retain such identification.

Uniquely identifying an XFA template allows its use to be tracked and controlled, whether the template is used for form fill-in or is modified by other template designers for another purpose.

IMPORTANT: All applications that produce XFA templates (XFA template designers) or that process forms based on XFA templates (XFA processing applications) are required to respect template-creator unique identifiers and time stamps.

Unique Identifiers and Time Stamps

The XDP representation of XFA include a Universally Unique Identifier (UUID) and a time stamp. The UUID is kept with the template throughout its life and the time stamp is updated whenever the template is modified. The presence of the UUID and time stamp allows XFA template designing and XFA form processing applications to track and control XFA templates.

The UUID and time stamp are included in any serialized representation of the template. In XDP format, this information appears in the XDP root element as the attributes `UUID` and `timeStamp` ("XDP Specification" on page 873).

If an XFA form is stored in PDF format, the UUID and time stamp are lost.

Processing Requirements for Template Designing Applications

If an XFA template designing application opens a template that omits a UUID, it must create one for it. If it opens a template that includes a UUID, it must not change the value of that UUID.

XFA template designing applications must update a template's time stamp whenever the template is changed.

Processing Requirements for XFA Processing Applications

Although XFA processing applications cannot change a template, they can be directed to submit an XFA template to a server. In such a situation, the XFA processing application must include in its submittal the UUID and time stamp from the original template.

The following template segment includes an submit button that causes the data and template to be submitted to the server.

Example 16.1 Submitting event that sends the template and data to the target server

```
<subform … >
   <field … >
      <event activity="click" … >
         <submit format="xdp" xdpContent="template datasets" … />
      </event>
   </field>
</subform>
```

If an XFA template includes a UUID and time stamp and that template is submitted to a server, the UUID and time stamp are included in the XDP or PDF created for that template.

Protecting an XFA Server from Attack

Respecting External References in Image Data and Rich Text

External references may appear in data supplied to the XFA processing application in the following forms:

- Referenced images, where the reference is represented as an `href` specification ("Image Data" on page 132)
- Embedded references in rich text, where the reference is represented as an `xfa:embed`, where the adjacent `xfa:embedType` is set to "URI" ("Embedded Object Specifications" on page 1044)

Whether such external references are resolved depends on the trust given to the URI described in that reference.

- Trusted. If the `href` reference is trusted, the image data may be included in the XFA Data DOM regardless of where the reference points.
- Not trusted. If it is not trusted, the XFA processor verifies that the referenced location is inside the current package, i.e. inside the XDP or PDF that supplied the template. If it is not inside the current package the reference is blocked.

Referenced images in data are described in "Image Data" on page 132.

Caution: Image references that point outside the package, even to a trusted URI, are undesireable from the standpoint of reliability and portability. See "Structuring Forms for Portability and Archivability" on page 480 for more information.

Discarding Unexpected Submitted Packets

The XFA submit mechanism provides the option of submitting the template and configuration information along with the data. However templates may contain scripts that execute on the server side. However XFA does not provide a native mechanism for establishing that a submitted template is trustworthy. Hence in any environment in which submissions are accepted from untrusted clients, care should be taken to ensure that any submitted template is discarded and a local copy of the template used instead.

Potentially similar problems could arise from accepting a configuration packet from an untrusted client. The best thing to do when dealing with untrusted clients is to discard every submitted packet that is not expected.

Encrypting Submitted Data

Starting with XFA 2.5, data submitted via HTTP or e-mail may be encrypted. This is more secure than SSL/TSP because it is not subject to man-in-the-middle attacks. The SSL/TSP approach only ensures that the conversation between client and server is private, not that the server is who it says it is.

Caution: For maximum security the form must also be certified (that is, signed by the form creator). This prevents a form of man-in-the-middle-attack in which the blank form is intercepted by a third party and the third party tricks the client into submitting data to a forged URI.

Signing Submitted Data

Starting with XFA 2.5, data submitted via HTTP or e-mail may be signed with one or more private keys. This provides the host with a way to ensure that the data was provided by a trusted source and has not been tampered with along the way. Encrypting and signing may be combined for maximum security.

Signed Forms and Signed Submissions

Digital signatures can be applied to forms to provide various levels of security. Digital signatures, like handwritten signatures, allow signers to identify themselves and to make statements about a document. Such statements include authorship of data in the form or approval of part or all of a form. The technology used to digitally sign documents helps to ensure that both the form signer and the form recipients can be clear about what was signed and whether the document was altered since it was signed.

A digital signature can be used to authenticate the identity of a user and the document's contents. It can store information about the signer and the state of the document when it was signed. The signature may be purely mathematical, such as a public/private-key encrypted document digest, or it may be a biometric form of identification, such as a handwritten signature, fingerprint, or retinal scan. The level of security and integrity associated with a digital signature depends upon the handlers and algorithms used to generate the signature and the parts of the form reflected in the signature.

Another application of digital signatures is to authenticate data which is submitted from a client to a server. Starting with XFA 2.5 such submitted data may be signed.

Digital signatures are an important component of secure XML applications, although by themselves they are not sufficient to address all application security/trust concerns, particularly with respect to using signed XML (or other data formats) as a basis of human-to-human communication and agreement.

Types of Digital Signatures

This section introduces digital signatures and describes how a template can be designed with clickable features that initiate the creation of digital signatures.

Note: The clickable features that initiate the creation of digital signatures are separate from the signatures themselves.

XFA supports the following signature mechanisms:

- *XML digital signature*. One or more signatures can be inserted into a form using the mechanism defined by the W3C for an XML Digital Signature [XMLDSIG-CORE]. This mechanism is selective in regard to what portion of the form is included in the signature. It can be used to sign any or every portion of the form which is expressed in XML, including the template, the configuration document, and/or the data.

  The clickable feature that produces an XML digital signature is an event with a `signData` property.

- *PDF digital signature*. A form which is embedded inside PDF can use the PDF signing mechanism [PDF]. The PDF signing mechanism may sign the whole of the XFA form and in addition may sign non-XFA content of the form. Hence a PDF signature can generate a document of record, which is described in the next section.

  The clickable feature that produces a PDF digital signature is a signature widget.

A single form may contain multiple XML digital signatures and multiple PDF digital signatures, although such use is not expected to be useful.

Using Digital Signatures to Achieve Different Levels of Security

XML digital signatures are used to achieve various levels of security. This section discusses those different purposes and how XML and PDF digital signatures can be used to achieve them.

There are several different types of signature purposes, each of which imposes its own requirements. The different types of signatures are summarized by the following table. The following sections explain how digital signatures can be used to achieve these purposes. Similar information on PDF digital signatures is available in the *PDF Reference* [PDF] and in *A primer on electronic document security* [ElectronicSecurity].

Signature purposes

Purpose	Use	How achieved
Integrity	Verify that data has not been corrupted in transit or processing. For example, when a digital signature is applied to a quarterly financial statement, recipients have more assurance that the financial information has not been altered since it was sent. See "Integrity" on page 468.	• Signature based on relevant parts of the form and optionally a private key
Authenticity Achieving this purpose results in a "trusted document" or a "document of record".	Verify a signer's digital identity. For example, a digitally signed quarterly financial statement allows recipients to verify the identity of the sender and assures them that the financial information has not been altered since it was sent. See "Authenticity" on page 470.	• Signature based on selected portions of the template and configuration and on pre-rendered PDF and a private key • Verification using a public key • Assurance of the sender's identity
Non- repudiability Achieving this purpose results in a "certified document".	Establish unequivocally that the person signing the document did in fact see and sign the document, or to establish that the recipient did in fact receive the document. See "Non- Repudiability" on page 470.	Same as for Authenticity with the following addition: • Trusted third-party software prevents the signer of the document from denying that they signed the document
Usage rights Achieving this purpose results in a "ubiquitized document". (a PDF capability)	If signature permissions have been issued by a bona fide granting authority, enable additional rights (such as the ability to sign) in special viewing applications such as Acrobat. See "Usage Rights Signatures (Ubiquitized Documents)" on page 471..	• Establish the identity of the granting authority • Specify additional rights to be granted by the special viewing application

Differences Between XML and PDF Digital Signatures

There are substantial differences in the capabilities of XML and PDF digital signatures.

This specification defines XML digital signatures that support only data integrity; however, XML digital signatures could conceivably be designed to achieve the same level of integrity and signer authentication as PDF signatures.

Note: These comments apply only to situations in which the form itself is signed. It is also possible to generate a separate document wrapped in a PDF envelope or an XML envelope and sign the contents of the envelope. For example, this is what happens when signed data is submitted from a

client to a host. In such cases the contents of the envelope are not limited by this specification so any appropriate signature mechanism can be used.

Using certified signatures to restrict changes

Unlike XML digital signatures, PDF signatures support PDF certified signatures. A certified signature allows the document author to specify which changes are allowed in the form. A PDF viewing application such as Acrobat then detects and prevents disallowed changes. A certified signature must be the first signature applied to a form.

What part of the document can be signed

XML digital signatures can include part or all of the XFA form; however, they cannot include resources such as fonts, referenced images, or other attachments. In contrast, PDF signatures can include such resources.

Tracking changes during a form's lifetime

XFA does not provide a mechanism for tracking the changes made during the form's lifecycle. In contrast, the PDF architecture can identify what changes have been made to a document, including when the document was signed.

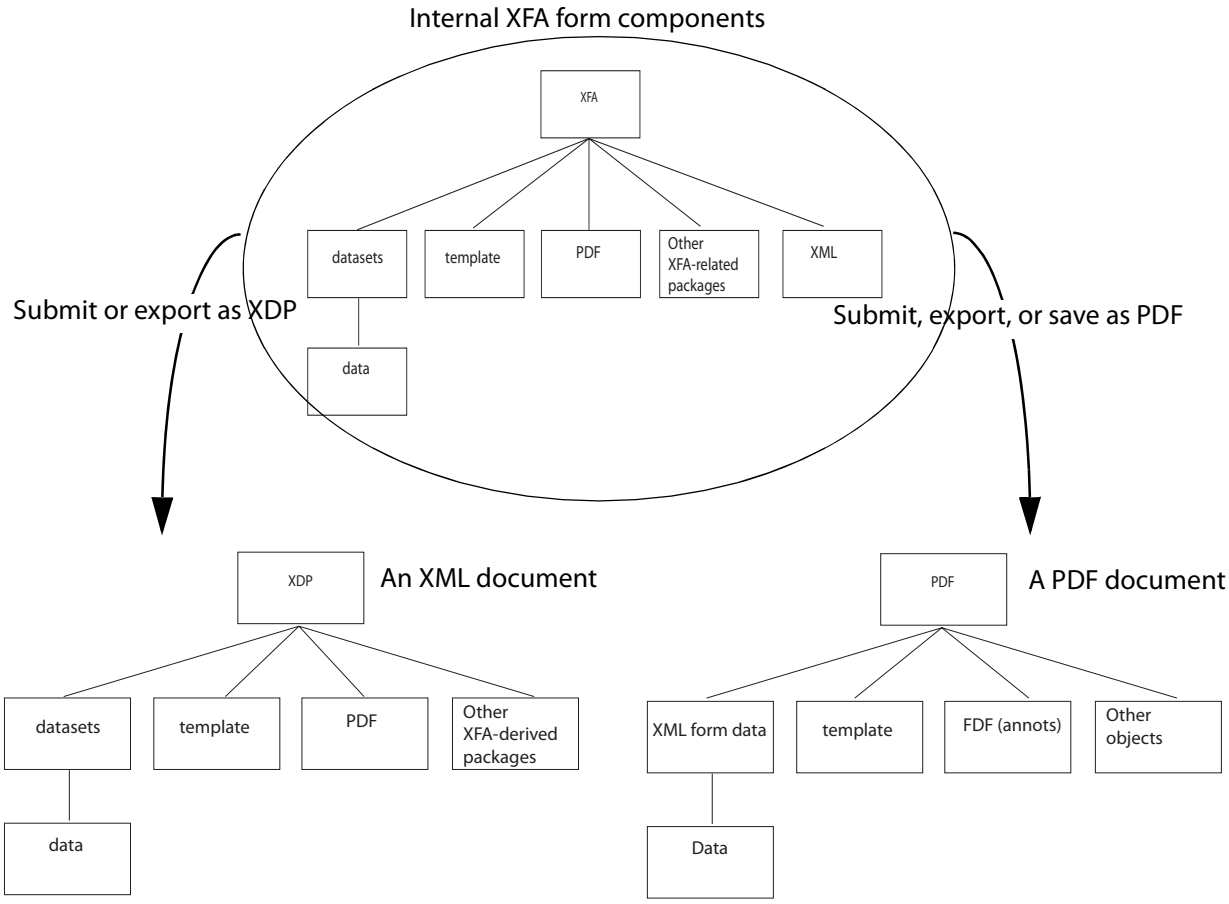
Refresher on the Relationship Between XFA and PDF

An understanding of the relationship between the XFA grammar, the PDF language and the applications typically used to create and process them is important in understanding how XML and PDF digital signatures affect security. Combinations of XML and PDF digital signatures may be applied to the same form; however, the different signature types have different processing requirements.

XFA and PDF digital signatures may be used in a form, regardless of the packaging (PDF or XDP), provided the objects referenced in the signature manifest are included in the package.

An XFA form can enclose a PDF object or can be enclosed by a PDF object. These different structures dictate whether the form is an XML document or a PDF document. The following diagram shows how an XFA form can be packaged.

Packaging of XFA form components into XDP or PDF



The following scenarios describe the addition and use of digital signatures in forms.

➤ Adding a PDF digital signatures to an existing template

In the following scenario, PDF digital signatures are added to the PDF that contains the XFA. Such a PDF digital signature can be used to produce a document of record, by referencing the XFA object from the manifest.

The form created by the following scenario can be opened only with a PDF-processing application, such as Acrobat. That is, it cannot be opened with XML-processing applications because it is not conforming XML.

1. A form designer creates the form using an XFA creating application such as Adobe LiveCycle Designer.
2. The designer creates a submission button that causes the entire form to be included in the submission. This is the format in which the form data and parts of the form are submitted to a server when the user completes the form and selects the form's submit button. (This statement assumes a rather simple form submission process.)
3. The designer saves the form as PDF.
4. The designer opens the form using a PDF-capable application such as Acrobat.
5. The designer applies a PDF digital signature widget. (Acrobat Professional is required for this operation.)

6. The designer saves the resulting form, again as PDF.

The following scenario applies when the form is being filled out.

1. The form is opened using a PDF-capable application that also supports XFA, such as Acrobat.
2. In the case of a trusted document, the application declares the trustability of the form.
3. The person fills out the form.
4. The person submits the form by selecting a submit button. In response, the XFA processing application bundles the designated parts of the form into a PDF document and submits that document to a server. (This statement assumes a rather simple form submission process.)

➤ **Adding an XML digital signatures to a template**

In this example, an XML digital signature is added to the template. Such a signature would be used for data integrity, by referencing the data object from the signature manifest.

1. A form designer creates a template. One of the elements in the template is a button associated with XML digital signature action (`signData`).
2. The designer defines the submission format as XDP. This is the format in which the form data and parts of the form are submitted to a server. After completing the form, the user selects a submit button. In response to this selection, the XFA processing application bundles the designated parts of the form into an XDP package and submits that package to a server. (This statement assumes a a simple form submission process.)
3. The designer saves the form as PDF or XDP.

The following scenario applies when the form is being filled out.

1. The form is opened using an XFA processing application.
2. The person fills out the form.
3. The user submits the form by selecting a submit button. In response, the XFA processing application bundles the designated parts of the form into an XDP package and submits that package to a server. (This statement assumes a simple form submission process.)

Integrity

Digital signatures enable recipients to verify the integrity of an electronic document used in one-way or round-trip workflows. For example, when a digital signature is applied to a quarterly financial statement, recipients have more assurance that the financial information has not been altered since it was sent. *A primer on electronic document security* [ElectronicSecurity] describes methods for maintaining integrity.

The following types of signatures are commonly used to support document integrity:

- Ordinary signatures, which can associate a signer with part or all of the document. For example, a user's signature may indicate approval of the data in certain fields of the form.
- Modification Detection and Prevention (MDP) signatures, which specify what changes are permitted to be made the document and what changes invalidate the author's signature.
- Usage rights (UR) signatures, which identify the authorizing agent and enable capabilities in special PDF-viewing applications.

The main differences between these signatures lies in what portions of the form are covered by the signature and what supplemental information is provided about each signer or signature. XFA provides all the necessary grammar to express any of these types of signature. However what types of signature are supported by a particular application is entirely application-defined. Individual XFA applications may support any, all, or none of the above types of signatures. For example, a non-interactive application might verify signatures but would probably never generate signatures.

It is normal and expected that different signatures may cover overlapping portions of the form. For example, Susan fills out a request for vacation. Her signature signs the subform that she filled out. Her boss Henry then approves the request. Henry's signature signs the entire form, including the subform that Susan filled out. Any change to that subform invalidates both signatures.

Using XML digital signatures for integrity

XML digital signatures can establish the integrity of a form, by incorporating[1] relevant objects in the signature. For example, if there is a concern only about the integrity of a form's data, the signature would incorporate only the form's data. If there is a concern about other aspects of the form, the signature would incorporate those other aspects, too.

An XML digital signature can incorporate the PDF object used in a form, but this is useful only for archiving. It is not useful in a workflow where other individuals subsequently validate the original signature. This limitation applies only to forms whose signature manifests include a PDF object. This limitation exists because PDF objects contain volatile information, such as date and time. If a PDF processing application such as Acrobat reopens and saves forms whose signatures include the PDF object, those signatures are voided, even if no changes are made.

It is possible to sign a template to indicate that the form can be trusted. When doing so the form creator must be careful with `setProperty`. Suppose that a form uses `setProperty` to copy user data into a URI. One user could enter a toxic URI and then forward the resulting form to someone else, perhaps via e-mail. In this way a signed template from a trusted source could be subverted. The `setProperty` property is described in "The setProperty property" on page 184.

Using PDF digital signatures for integrity

The PDF signature mechanism is described in the PDF reference, [PDF].

A PDF digital signature can incorporate the XFA stream used in a form, provided the XDP is packaged inside the PDF, but this is useful only for archiving. It is not useful in a workflow where other individuals subsequently validate the original signature. This limitation applies only to forms whose signature manifests include the XFA stream. This limitation exists because the XFA stream is in XML and there may be changes made to an XML stream which are defined as not significant by the XML standard [XML1.0]. For example white space preceding the closing '>' or '/>' of a tag is by definition not significant. The digital signature for an XML stream should not be invalidated by the addition, deletion, or modification of any such insignificant content. However the PDF signing mechanism is not XML-aware so PDF signatures are invalidated by such changes. Therefore simply writing the XFA stream out again using a different XFA processor (or a different version of the same processor) may invalidate the PDF signature even though there is no substantive difference in the form.

Starting with XFA 2.5, a client can apply one or more to the data that it submits to a host. This allows the host to verify that the data has not been modified in transit. This is stronger than simply submitting via SSL/TSP because it is not susceptible to man-in-the-middle attacks. It also ensures that the signature travels all the way to the host application rather than being stripped off at the communications layer.

1. The term "incorporating" refers to the creation of hash code (or other representative binary number) that reflects the portions of the form specified in the signData manifest element. This code is then stored in the signature property created when the form is actually signed.

Authenticity

Achieving this purpose results in a "trusted document" or a "document of record".

Authenticity provides confidence that a document or part of a document does not take on a different appearance after being signed. The XFA grammar and the PDF language provide a number of capabilities that can make the rendered appearance of a form or PDF document vary. These capabilities could potentially be used to construct a document that misleads the recipient of a document, intentionally or unintentionally. These situations are relevant when considering the legal implications of a signed XFA form or PDF document. Therefore, it is necessary to have a mechanism by which a document recipient can determine whether the document can be trusted.

Using XML digital signatures for authenticity

XML digital signatures can establish the authenticity of a form, by incorporating in the signature relevant parts of the form (including the template) and certificates that identify the sender, and by using private-key encryption.

Using PDF Digital signatures for authenticity

Authenticity includes ensuring the integrity of the form and verifying the identity of the sender. With forms intended for fill-in, authenticity may be required in a form that is then fill-in and signed. For example, an accounting firm might send a financial report to another agency for comments and signatures. The accounting firm would want to ensure the recipient of its identity (authenticity) and prevent the financial report from being modified (integrity), with the exception of fields set aside for text comments and the signature of the recipient. After adding comments to the designed fields, the recipient would sign the document. The signature would be associated with the current state of the document. Although further modifications to the comment fields would be allowed, they would not be associated with the signature field.

PDF MDP signatures support the kind of form fill-in and signature described in the above paragraph.

Starting with XFA 2.5, a client can apply one or more signatures when sending data to a host. This allows the host to verify the identity of the sender by validating the signature against the corresponding public key.

It is possible for a template to invoke prototypes from external documents. These external prototypes can in turn invoke prototypes from other documents and so on. To ensure the authenticity of the document the PDF processor resolves all prototype references before generating the signature. The resulting PDF has vestigial prototype references (it still contains the URLs of the external prototypes) but it no longer has any dependence upon the external documents.

Non- Repudiability

Non-repudiation is a document security service that prevents the signer of the document from denying that they signed the document. Such a service is often driven by authentication and time-stamping from a trusted third-party.

Non- repudiable security is the same as document of record, with the additional verification that the person signing the form cannot deny signing the form. Using PDF signatures to establish non-repudiability is described in the *PDF Reference* [PDF] and in *A primer on electronic document security* [ElectronicSecurity].

Using XML digital signatures to establish non- repudiable documents is beyond the scope of this specification.

Usage Rights Signatures (Ubiquitized Documents)

Usage rights signatures are a PDF feature that enables additional interactive features that are not available by default in a particular viewer application (such as Adobe Reader). Such a signature is used to validate that the permissions have been granted by a bona fide granting authority and to determine which additional rights should be enabled if the signature is valid. If the signature is invalid because the document has been modified in a way that is not permitted or if the identity of the signer has not granted the extended permissions and additional rights are not granted.

Usage rights signatures are applied as described in "Adding a PDF digital signatures to an existing template" on page 467. XML digital signatures do not specify usage rights.

XML Digital Signatures

XFA specifies the structures used to support XML digital signatures. One structure specifies the signature-related operation the XFA processing application should perform and the other contains the result of a signing operation — an XML digital signature. An XFA processing application produces an XML digital signature when the person filling out a form activates an event that contains instructions for producing a signature.

The structure that specifies the signature-related operation is an XFA template element (`signData`). This structure provides operations for signing a form. It also provides operations for verifying and clearing existing signatures. The operation to perform is determined by the `operation` subproperty. When the value of `operation` is `sign` the effect is to create the signature. When the value is `verify` the effect is to verify the data against the signature and generate a message if it does not match. Scripts can also manipulate XML digital signatures using methods of the `xfa.signature` object.

The structure that contains the result of a signing operation (`Signature`) is an XML element that resides outside the template namespace. [XMLDSIG-CORE] defines how digital signatures are produced and how they are represented in a `Signature` property, with the following additions: XFA augments the `Signature` object with information that allows XFA processing applications to verify and clear the signature. This addition information is discussed "Template Provides Instructions on Signature Operation" on page 474.

Signing a Form

An XFA processing application produces an XML digital signature in response to a user activating an event that contains a `signData` property with an `operation` property of `"sign"`. Such an event is usually activated by the user clicking a button. In response to the event activation, an XFA processing application performs steps such as the following, although the exact steps are application-dependent:

1. Initiate a dialog with the person filling out the form to determine which of the user's private certificates should be used to produce the signature. Typically certificates are used only when the application and handler specify a signature algorithm that supports a public key-based signature algorithm.

Note: The following steps are paraphrased from Section 3.1.1, "Reference Generation" and Section 3.1.2, "Signature Generation" in [XMLDSIG-CORE], with additional information pertinent to the XFA `signData` element.

2. For each object being signed, apply the application-determined transforms, calculate the digest value over the resulting data object, and create a reference element. The XFA template signData element

specifies the objects being signed, as described "Manifest: Data and other content digested by the signature handler" on page 474.

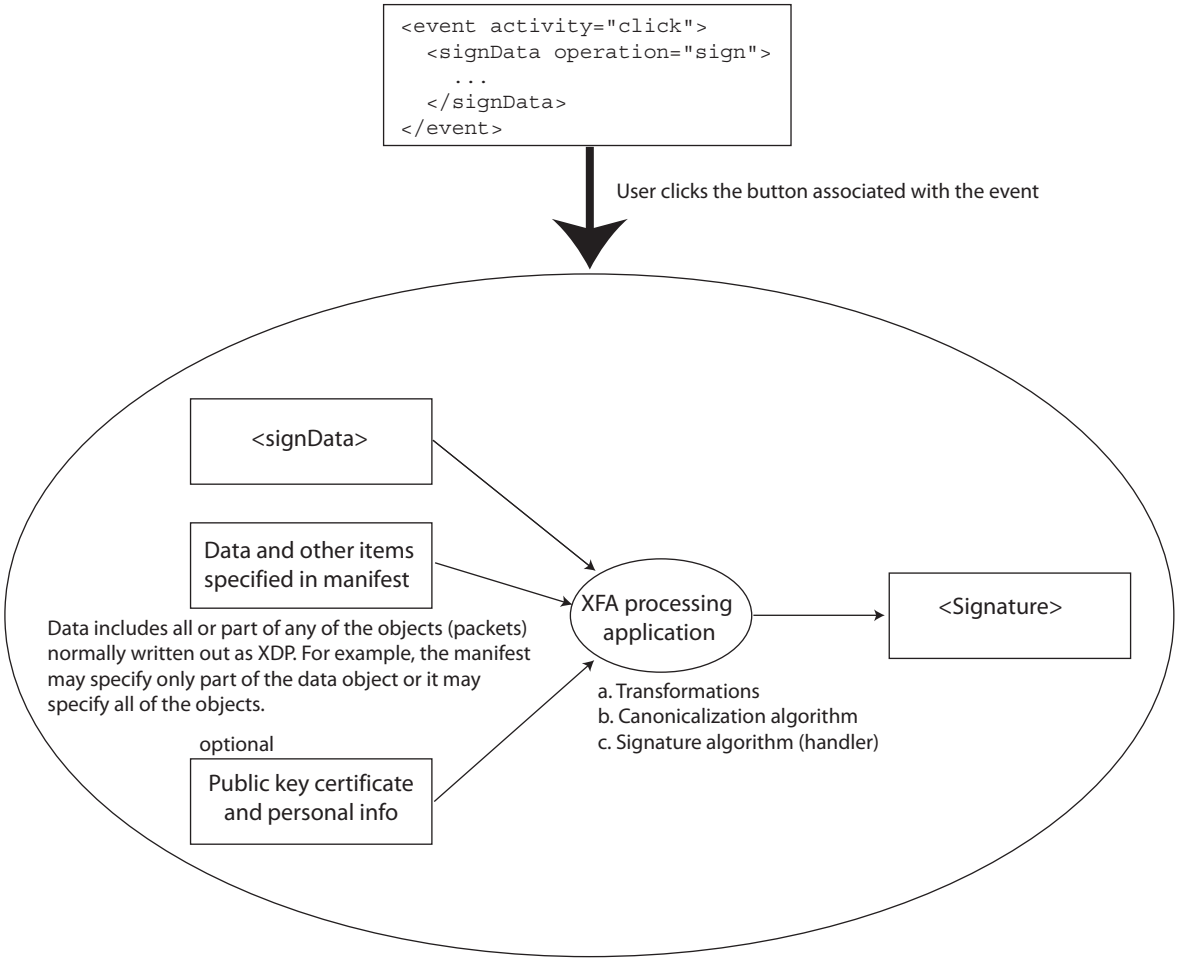
3. Create the XML object (`SignedInfo`) that describes how the signature is being produced. This object includes a description of the canonicalization method, the signature method, any transform algorithms, and the digest method.

4. Canonicalize over the objects specified in Step 2. The canonicalization algorithm is application-defined. Canonicalization is used because not every alteration to an XML document has meaning. For example, it is irrelevant whether an element start tag is separated from the following attribute name by a single space, multiple spaces, a line feed, or any other valid white space. Naively signing every character in the document would mean that any change at all to the document, even a meaningless change, would void the signature. Rather a canonical copy of the document is extracted and the signature is generated or verified based on the canonicalized copy. The W3C specification Exclusive XML Canonicalization [EXCLUSIVE-XML-CANONICALIZATION] specifies a variety of canonicalization algorithms.

5. Calculate the signature value over the objects specified in Step 2. This is done by invoking the signature handler specified or by invoking an application-defined algorithm. If the `signData` element specifies a handler that is required, the XFA processing application is obliged to use the specified handler. A *signature handler* is usually third-party software that handles the raw signing operation. The XFA processing application invokes it after applying other transformations and after obtaining other information, such as certificates, used by the signature handler. The signature handler computes a hash value based upon a combination of the data included in the manifest and a (usually private) signing key. This value is the digital signature.

6. Construct the Signature element that includes `SignedInfo`, the signature value, and other information that allows the signature to later be verified or cleared. The signature object includes the computed hash and URIs identifying the certificates and certificate policies needed to verify the signature. It also includes a manifest that identifies each of the XML elements included in the signature. Although this manifest is derived from one originally specified as XFA-SOM expressions within an XFA `manifest` element, within the signature the manifest is expressed using [XPATH] expressions as required by the [XMLDSIG-CORE] specification. The signature also includes additional information, not mandated by [XMLDSIG-CORE], which is described below.

Role of <signData> and <Signature> in producing a digital signatures



Removing a Signature

An XFA processing application produces an XML digital signature in response to a user activating an event that contains a `signData` property with a `operation` property of "`clear`". Such an event is usually activated by the user clicking a button.

In response to such an event being activated, the XFA processing application removes the signature signature by simply by stripping out the signature object. This can be done by anyone with access to the document. Hence, signatures are hard to apply (i.e. they require possession of private keys to apply) but they are easy to remove.

Verifying a Signature

An XFA processing application produces an XML digital signature in response to a user activating an event that contains a `signData` property with a `operation` property of "`verify`". Such an event is usually activated by the user clicking a button.

In response to such an event being activated, the XFA processing application invokes the signature handler, specifying that signature verification is desired and supplying a pointer to the signature object.

Template Provides Instructions on Signature Operation

The `signData` property specifies a signature-related operation, which may be used to produce a signature (`sign`), verify an existing signature (`verify`), or clear an existing signature (`clear`). If the operation is to produce a signature, the `signData` property specifies how the digital signature should be produced, including the signature handler, the signature destination, and the signature manifest (what part of the XFA document is being signed). If the operation is to verify or clear an existing signature, only the location of the signature (as the `ref` property) need be provided. The signature being cleared or verified is self-describing.

Signature filter: Handler, algorithms, and certificates to use to produce a signature

The `signData filter` property specifies the signature handler and certificates for use in producing the signature. It also specifies a list of potential reasons a document is being signed.

The architecture specified by [XMLDSIG-CORE] allows for different signatures to use different signature algorithms. XFA processors support at least the RSA-SHA1 and DSA-SHA1 methods. The method used is recorded in the digital signature so that the signature can be processed by generic software.

A digital certificate allows a document's recipient to know whether or not a specific public key really belongs to a specific individual. Digital certificates bind a person (or entity) to a public key. Certificate authorities (CA) issue these certificates and recipients must trust the CA who issued the certificate. X.509 is the widely accepted certificate standard that Adobe uses.

Most of the properties in `filter` include a `type` subproperty. This property indicates whether the XFA processing application is restricted to using the indicated item or selecting from the indicated items. The following example requires the XFA processing application to use the signature method "`Adobe.PPKList`" and to restrict the signing certificates to one of the seed values provided. These settings limit the individuals who can use this signature to the department head and the supervisor.

Example 16.2 Signature instructions that restrict who can sign

```
<event>
   <signData operation="sign" …>
      <manifest … />
      <filter>
         <handler type="required">Adobe.PPKList</handler>
         <certificates url="">
            <signing type="required">
               <!-- Department heads certificate -->
               <certificate>MIB4jCCAUugAwkdE13 … </certificate>
               <!-- Supervisors certificate -->
               <certificate>MIB4jCCAUugAwkdE13 … </certificate>
            </signing>
         </certificates>
      </filter>
   </signData>
</event>
```

Manifest: Data and other content digested by the signature handler

The `signData manifest` property provides a list of SOM expressions that specify which parts of the form should be reflected in the digital signature. If the manifest is non-empty, the signature handler uses the referenced items in its production of the XML digital signature. The SOM expressions may reference

part or all of any of the packets written out to XDP ("XDP Specification" on page 873). Such packets include dataSets, config, and localeSet.

XML digital signatures adopt the mechanism specified by [XMLDSIG-CORE], which is an XML-specific mechanism. The signature handler digests the XML prior to being written out as XDP. This distinction is important relative to data, which has one form in the XFA Data DOM and another (possibly) different form in the XML Data DOM. Such differences exist for the following reasons:

- XSLT transformations which may be applied when loading or saving the XML Data Document. Use of these transformations is described in "XSLT Transformations" on page 458.
- Before rich text in the XML Data DOM is brought into the XFA Data DOM, it is converted into plain text. This separation is described in "Representing Rich Text in the XFA Data DOM" on page 190.
- Hypertext references to images in rich text are resolved in the XFA Form DOM, but not in the XML Data DOM. As a result, the data in such references is omitted even if the entire form is included in the manifest.

It is an error to specify a manifest for an XML digital signature that includes a node that is not written out. For example, the manifest must name neither data nodes that are marked transient nor other form properties that are never written out.

Signature destination

The `signData ref` property specifies the location where the `Signature` element is to be stored (if the `signData` operation is `"create"`) or has been stored (if the `signData` operation is `"clear"` or `"verify"`).

The signature can be place anywhere that a SOM expression can reach. However it is recommended that `Signature` elements be placed within the `datasets` element, but outside the `data` element which is the child of `datasets`. Multiple sibling `Signature` elements can be accommodated. The signatures are distinguished by XML ID, not by element name, in keeping with the dictates of [XMLDSIG-CORE]. See "The datasets Element (an XDP Packet)" on page 880.

Occasionally, it is necessary to place a signature in a location that will not travel with the data. In this case `Signature` elements may be placed as XDP packets, that is, as children of the XDP's root `xdp` element. See "The signature Element (an XDP Packet)" on page 882.

XFA-Specific Information in the XML Digital Signature

XML digital signatures generated by XFA processors contain additional information beyond that required by [XMLDSIG-CORE]. The additional information uses namespaces other than the namespace for XML signatures so it does not interfere with generic signature processing. The additional information that is included is:

- Date and time of signing
- A reason for signing.

The reason for signing must be selected from a list of possible reasons which is descended from the `reasons` subproperty of the `filter` property of the `signData` object. For example, assume the template contains the following fragment.

Example 16.3 Template fragment using a list of reasons

```
<field ...>
 <event ...>
   <signData operation ="sign" ...>
```

```
    ...
    <filter>
      <reasons ...>
        <reason>Requested</reason>
        <reason>Approved</reason>
      <reasons>
    </filter>
  </signData>
 </event>
</field>
```

Then the additional information inserted into the XML digital signature could look like this:

```
<Signature Id="mySIg" xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    ...
    <Object>
      <SignatureProperties>
        <SignatureProperty ...>
          <x:xmpmeta xmlns:x='adobe:ns:meta/'>
            <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
              <rdf:Description rdf:about=""
                xmlns:xmp="ns.adobe.com/xap/1.0/"
                xmlns:dc="http://purl.org/dc/elements/1.1/"
                xmlns:xfa="http://www.xfa.org/schema/xfa-template/2.5/">
                <xmp:CreateDate>2005-08-15T17:10:04Z</xmp:CreateDate>
                <dc:description>Approved</dc:description>
              </rdf:Description>
            </rdf:RDF>
          </x:xmpmeta>
        </SignatureProperty>
      </SignatureProperties>
    </Object>
  </SignedInfo>
</Signature>
```

Example

The following form presents buttons for signing, verifying, and unsigning the form. The signature includes the root element of the data document and all of its content, but nothing else. When the signature is present it is enclosed in an element called `signatures` which is placed within the `datasets` element. Note the use of a prototype to avoid repeating information in each `signData` element.

Example 16.4 Full signing example template

```
<template xmlns="http://www.xfa.org/schema/xfa-template/2.1/">
  <subform name="form1" ...>
    <pageSet>
      …
    </pageSet>
    <subform …>
      <field …>…</field>
      …
      <field name="Sign" …>
        <ui>
```

```
            <button/>
          </ui>
          <caption>
            <value>
              <text>Sign the data</text>
            </value>
          </caption>
          <bind match="none"/>
          <event activity="click">
            <signData operation="sign" use="#mySignData"/>
          </event>
        </field>
        <field name="Verify" ...>
          <ui>
            <button/>
          </ui>
          <caption>
            <value>
              <text>Verify the signature</text>
            </value>
          </caption>
          <bind match="none"/>
          <event activity="click">
            <signData operation="verify" use="#mySignData"/>
          </event>
        </field>
        <field name="Clear" …>
          <ui>
            <button/>
          </ui>
          <caption>
            <value>
              <text>Remove the signature</text>
            </value>
          </caption>
          <bind match="none"/>
          <event activity="click">
            <signData operation="clear" use="#mySignData"/>
          </event>
        </field>
      </subform>
      <proto>
        <signData id="mySignData" target="mySignature" ref="!signatures">
          <manifest>
            <ref>$data.form1</ref>
          </manifest>
          <filter>
            <handler version="…" type="required">…</handler>
            <reasons type="required">
              <reason>…</reason>
              …
            </reasons>
            <certificates url="MyCertURL">
              <signing type="optional">
                <certificate>…</certificate>
```

```
                …
              </signing>
              <issuers type="required">
                <certificate>…</certificate>
                …
              </issuers>
              <oids type="optional">
                <oid>…</oid>
                …
              </oids>
            </certificates>
          </filter>
        </signData>
      </proto>
    </subform>
</template>
```

PDF Signatures

PDF signatures can be applied to the XFA form itself or to a separate document which accompanies submitted data.

PDF Signatures Applied to the Form Itself

When aPDF signature is applied to the XFA form itself it is a document-of-record signature because it always includes all portions of the XFA template, configuration, and data and all portions of the PDF that bear upon the signed field(s).

Prior to XFA 2.5, PDF signatures always included all fields of the form. Starting with XFA 2.5 a PDF signature may be restricted to a subset of the fields.

A PDF signature is placed upon a form by the user clicking on a signature widget. To this end, XFA defines a signature widget that is used only for PDF signatures. The widget itself displays the signed or unsigned state of the document. Since there can potentially be more than one signature widget on a document, each widget independently displays its own signed state.

The signature widget is associated with an XFA field object but it never contains user data. By default the signature computation includes all fields, as well as the Template and Config DOMs, creating a document of record for the entire form. However the signature field may specify a manifest which limits the signature to a subset of the fields in the form. This provides the XFA equivalent of the PDF object digest for signatures, except that the grammar for the XFA manifest is more flexible.

Applying a PDF signature to a form does not prevent subsequent alterations to the form; however, if the signed portion of the form is altered, the signature dictionary stored in the document no longer matches a freshly calculated signature value. Hence, analysis can determine that the form was tampered with after signing.

When the PDF signature covers all fields in the form its computation includes the entire XFA form embedded in the PDF and most of the non-XFA content in the PDF as well. Some portions of the non-XFA content are omitted as specified in the PDF standard [PDF].

Unlike an XML digital signature, a PDF signature signs the XFA form exactly as it is currently expressed, rather than signing a normalized copy. This means that it is not possible to make even meaningless

changes to the XFA form without voiding the signature. For example, changing a space to a tab in between an element tag and the following attribute name voids the signature, even though it does not change the meaning of the XML.

PDF Signatures Accompanying Submitted Data

Data may be submitted in a PDF envelope that also bears one or more signatures. This envelope is an independent document, entirely separate from the XFA form. The envelope is constructed, filled, and signed during the submit operation. Once the client has sent it to the host the client deletes the envelope and its content.

Because the PDF envelope is constructed when needed and deleted immediately afterward the usual problems with signing XML inside PDF do not apply. It does not matter that PDF and XML signature filters differ because the PDF signature applies to the exact stream of bytes carried within the PDF. Signatures that apply only to portions of the data can be delineated by simple byte ranges within the contained stream of bytes. This is within the capability of the PDF signature mechanism.

Structuring Forms for Portability and Archivability

Portability is a basic premise of PDF, hence the name *Portable* Document Format. PDF documents can be used on-line or off-line, and they can distributed electronically or as hard copy.

By contrast XDP is not intended for portability but for ease of interoperability with other software. That is why it is based on XML.

In order to be portable a PDF document must be self-contained. It must not depend on network access. People expect to copy a PDF file to a CD-ROM or a USB key and be able to use it even when they don't have network access.

XDP documents are not expected to have this degree of portability. The primary use for XDP documents is either within a server or in transit between a client and a server. Hence an XDP file may include URIs pointing to network resources. For example, it may invoke external prototypes which are located on an HTTP server.

When converting an XFA form from XDP format to PDF, the following rules must be followed to ensure that the PDF file is properly portable.

- Resolve external prototypes.
- Put external images inline.
- Include definitions for all required locales.
- Include all non-default configuration settings.

Fields containing rich text may contain `image` elements which reference external images. Such elements are not required to be respected as rich text markup, however XFA processors may still parse the `image` element and inspect the target of the URI. However to preclude portability problems XFA processors refrain from following such URIs outside the package, as described in "Image Data" on page 132.

The requirements for archiving are similar to the requirements for portability but more stringent. For example, a non-archival document might rely on the default settings for a well-known locale. However an archival document must specify definitions for all the locales it uses because locale definitions change from time to time. When the name of the national currency changes, and possibly at the same time its value, the archived document must not automatically pick up the new currency while retaining the old numbers!

Because XFA is intended for use in financial transactions we recommend adhering to the tougher archival standard for all XFA forms packaged as PDF files. Ideally one would wish to include all known locales in every form but this is not practical because each locale definition adds about 3 kilobytes and there are hundreds of locales defined. Instead the form should include definitions for all locales in which the form might be used.

Part 2: XFA Grammar Specifications

This part provides a set of specifications that describe the elements and attributes of each of the grammars that comprise XFA. Each chapter describes one of these grammars.

17 Template Specification

This chapter is the language specification for the XFA template syntax. The reference portion of this specification with syntax descriptions begins on page 488.

Guide to the Template Specification

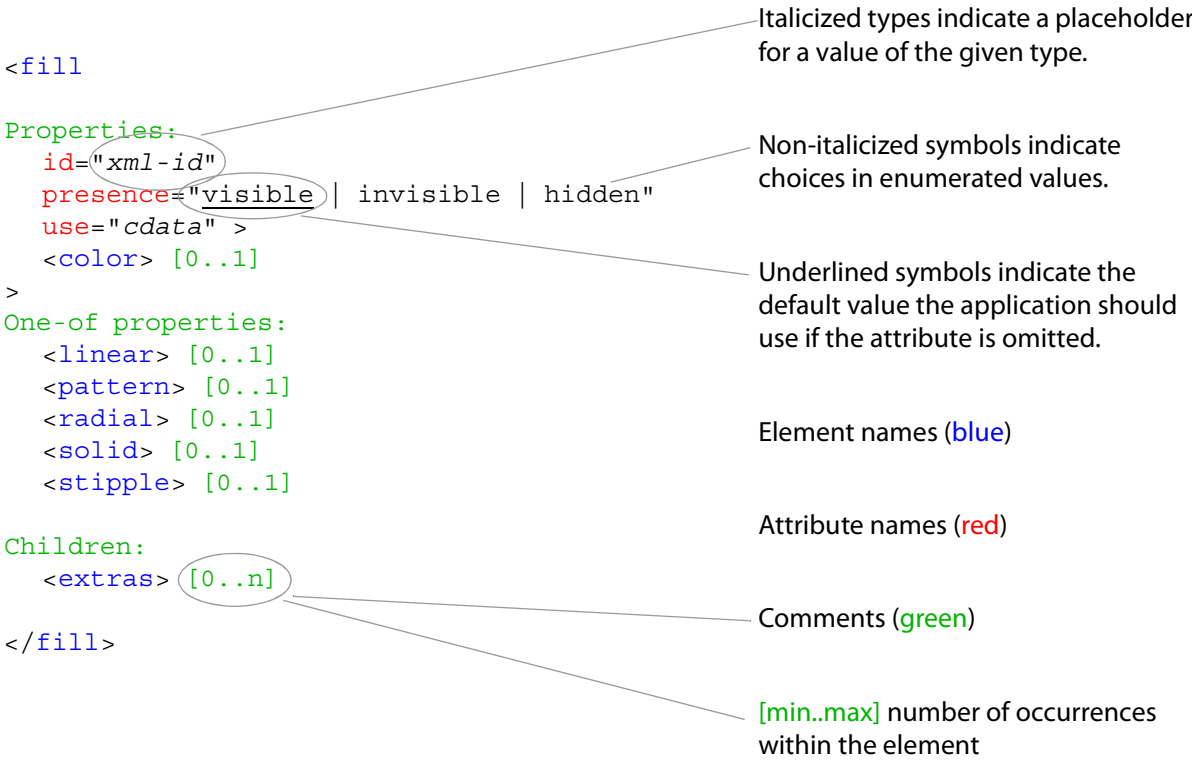
This chapter provides information that will help you understand the material presented in the template syntax reference. It describes typographic and formatting conventions and concepts represented by each element description, including properties, one-of properties, and children. It also discusses default properties and property occurrence.

How to Read an Element Specification

"Template Reference" on page 488 contains a subsection for each element in the XFA-Template language. All of those subsections follow the same format; indeed, they are machine-generated.

Each element description starts with an XML syntax definition — a human readable schema for the element.

The element description comprises nested subsections that describe each of the element's attributes and child elements. These attributes and child elements are partitioned into the groups: properties, one-of properties and children, as is apparent in the following example. These groups are described in the following subsections.



Properties

As in object-oriented programming, properties describe the objects to which they are attached.

A property represents a logical grouping of information that may be represented as a single attribute or as as a tree structure of elements. A property includes all the information contained in the elements and attributes used to represent it.

Properties may be unstructured or structured; XFA-Template uses attributes to describe unstructured properties and child elements to describe structured properties. For example, the `fill` element's attributes (see above syntax) are all unstructured properties, while its `color` element is a structured property.

All properties must be in the XFA template namespace to be included in the template DOM. That is, the XFA template grammar cannot be extended through the use of custom namespaces. However, the XFA template grammar provides `extras` elements that can be used for extensions.

The element descriptions in the template syntax reference differentiate between (regular) properties and one-of properties, as shown in the example on the previous page.

Regular Properties

Regular properties can be added to the element without regard to other properties in the element. The element descriptions in this specification use the term *Property* to identify such regular properties.

In the case of elements, occurrence constraints must be honored.

One-of Properties

There are occasions where an element supports mutually-exclusive properties. For unstructured properties, an attribute enumeration represents the mutually-exclusive values, and these are not distinguished from regular properties. However, for structured properties, the entire structures are likely mutually-exclusive.

The element descriptions in this specification use the term *One-of property* to identify mutually-exclusive, structured properties. The element must hold at most one of the allowed one-of property child elements.

In the `fill` element example at the beginning of this chapter, the `linear`, `pattern`, `radial`, `solid` and `stipple` elements are mutually-exclusive, having been identified as One-of properties.

Property Defaults

The processing application must supply defaults for properties omitted from an element, using the following guidelines:

- Regular properties. The processing application must provide the default values indicated in the element descriptions in this specification.
- One-of properties. The processing application must provide one of the properties as a default. That is, the absence of any one-of child elements implies the application must provide a default.

Children

Elements in the Children category don't represent properties at all. They represent tangible objects that often have the capability to contain each other and often are indeed called “containers”. Examples of such children include the `field` and `subform` elements. A `subform` element has a variety of attributes and child elements that represent the properties of the `subform` itself. Additionally, the `subform` may enclose a number of child elements that express children of the subform, such as fields, draws, or other subforms.

The distinction between child elements that are (structured) properties and those that are “true” children is intentional. While all properties could be expressed as attributes, the attribute proliferation required to describe a deep structure would be overwhelming. Property child elements tend to be singly occurring, or occurring in known numbers (e.g., four edges in a border).

Element Occurrence

Singly Occurring Elements

Elements that are defined as singly occurring [0..1] are permitted to be defined only once within the scope of the enclosing element. Unless stated otherwise all elements are singly occurring. Singly occurring elements usually each represent a property of the enclosing element, rather than an object aggregated by

the enclosing element. Observe the following example of a filled white rectangle, with rounded corners and alternating solid and dashed edges:

```
<Draw>
  <Fill>
    <Color Value="255,255,255">
    <Solid/>
  </Fill>
  <Value>
    <Rectangle>
      <Corner Join="Round"/>
      <Edge Stroke="Solid"/>
      <Edge Stroke="Dashed"/>
    </Rectangle>
  </Value>
</Draw>
```

In the example above, we see that the `Edge` element has been multiply specified in the rectangle; the `Edge` and `Corner` elements are both specified as multiple occurrence XFA element types, and so each element contributes to the definition of some part of the rectangle. The `Fill` element is specified as a single occurrence XFA element type, and therefore only one of the occurrences of the element will contribute to the definition of the object.

Observe the following adaptation of the previous example of a white rectangle:

```
<Draw>
  <Fill>
    <!-- A white color fill -->
    <Color Value="255,255,255">
    <Solid/>
  </Fill>
  <Value>
    <Rectangle>
      <Corner Join="Round"/>
      <Edge Stroke="Solid"/>
      <Edge Stroke="Dashed"/>
    </Rectangle>
  </Value>
  <Fill>
    <!-- A black color fill -->
    <Color Value="0,0,0">
    <Solid/>
  </Fill>
</Draw>
```

In the example above, the draw element incorrectly contains two `Fill` elements. If the processing application encounters such an XFA Template that expresses an excessive number of a given element, the processing application may consider this an error or continue processing. If the application chooses to continue processing, it must accept only the first occurrence of the given element. Therefore, in the previous example the rectangle would have a fill of white (color value of 255,255,255).

Multiply Occurring Elements

Elements that are defined as multiply occurring are permitted to be defined more than once within the scope of the enclosing element. Multiply occurring elements are used to represent array-type properties or sub-objects aggregated by the enclosing element.

Observe the following example of a filled black rectangle, with rounded corners and alternating solid and dashed edges:

```
<Draw>
  <Fill>
    <!-- A white color fill -->
    <Color Value="255,255,255">
  </Fill>
  <Value>
    <Rectangle>
      <Corner Join="Round"/>
      <Edge Stroke="Solid"/>
      <Edge Stroke="Dashed"/>
    </Rectangle>
  </Value>
  <Fill>
    <!-- A black color fill -->
    <Color Value="0,0,0">
  </Fill>
</Draw>
```

In the example above, we see that the `Edge` element has been multiply specified in the rectangle; the `Edge` and `corner` elements are both specified as multiple occurrence XFA element types, and so each element contributes to the definition of some part of the rectangle.

The `Fill` element is defined as a singly occurring XFA element type, and therefore only the first occurrence of the element contributes to the definition of the object; hence, the rectangle shall be filled with a color of white.

When more multiply occurring elements are present than required, the element shall choose its required number of elements from the beginning of the set. Observe the following adaptation of the previous example:

```
<Draw>
  <Fill>
    <!-- A white color fill -->
    <Color Value="255,255,255">
  </Fill>
  <Value>
    <Rectangle>
      <Corner Join="Round"/>
      <Edge Stroke="Solid"/>
      <Edge Stroke="Dashed"/>
      <Edge Stroke="Solid"/>
      <Edge Stroke="Dashed"/>
      <Edge Stroke="Dotted"/>
      <Edge Stroke="Dotted"/>
    </Rectangle>
  </Value>
  <Fill>
    <!-- A black color fill -->
    <Color Value="0,0,0">
  </Fill>
</Draw>
```

In the example above, we see that the `Edge` element has been multiply specified in the rectangle for a total of six `Edge` elements. The element specification for the `Rectangle` element describes that there may be up to four edges specified for a rectangle. Therefore, only the first four occurrences of the `Edge` element shall be accepted; the two `Edge` elements with the values `Stroke` of `Dotted` shall not contribute to the rectangle.

Template Reference

The arc element

A curve that can be used for describing either an arc or an ellipse.

```
<arc

Properties:
    circular="0 | 1"
    hand="even | left | right"
    id="xml-id"
    startAngle="0 | angle"
    sweepAngle="360 | angle"
    use="cdata"
    usehref="cdata"
>
    <edge>  [0..1]
    <fill>  [0..1]
</arc>
```

The arc element is used within the following other elements:
proto value

Unlike borders and rectangles, the path of an arc follows a counter-clockwise direction. This has implications for handedness. In particular, an arc with a left-handed edge will render the edge's thickness just inside the path, while left-handed borders and rectangles render the thickness just outside the path. Similarly, an arc with a right-handed edge will render the edge's thickness just outside the path, while right-handed borders and rectangles render the thickness just inside the path.

The circular property

Specifies whether the arc will be adjusted to a circular path.

0

The arc will not be adjusted to a circular path.

1

The arc will be adjusted to a circular path.

The default value of this property is 0.

Setting this property to 1 causes the arc to become circular, even if the content region into which the arc is being placed is not square. When forced into a circle, the radius is equal to the smaller dimension of the content region.

The edge property

Specifies the appearance of the rendered arc path.

The default value corresponds to an edge that produces a 0.5pt black stroke.

For more information see "The edge element".

The fill property

Specifies the appearance of the area enclosed by the arc.

By default, an arc does not have a fill property, producing an arc with a transparent interior.

If the arc has a sweep angle less than 360 degrees, the processing application must fill an area formed by the path of the arc and a chord joining the arc's start and end points.

For more information see "The fill element".

The hand property

Description of the handedness of a line or edge.

`even`

Center the displayed line on the underlying vector or arc.

`left`

Position the displayed line immediately to the left of the underlying vector or arc, when following that line from its start point to its end point.

`right`

Position the displayed line immediately to the right of the underlying vector or arc, when following that line from its start point to its end point.

The id property

A unique identifier that may be used to identify this element as a target.

The startAngle property

Specifies the angle where the beginning of the arc shall render.

The sweepAngle property

Specifies the length of the rendered arc as an angle.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The area element

A container representing a geographical grouping of other containers.

```
<area

Properties:
    colSpan="1 | integer"
    id="xml-id"
    name="xml-id"
    relevant="cdata"
    use="cdata"
    usehref="cdata"
    x="0in | measurement"
    y="0in | measurement"
>
    <desc>  [0..1]
    <extras>  [0..1]

Children:
    <area>  [0..n]
    <draw>  [0..n]
    <exclGroup>  [0..n]
    <exObject>  [0..n]
    <field>  [0..n]
    <subform>  [0..n]
    <subformSet>  [0..n]
</area>
```

The area element is used within the following other elements:
area pageArea proto subform

The area child

A container representing a geographical grouping of other containers.

For more information see "The area element".

The colSpan property

Number of columns spanned by this object, when used inside a subform with a layout type of `row`. Defaults to 1.

The desc property

An element to hold human-readable metadata.

For more information see "The desc element".

The draw child

A container element that contains non-interactive data content.

For more information see "The draw element".

The exclGroup child

A container element that describes a mutual exclusion relationship between a set of containers.

For more information see "The exclGroup element".

The exObject child

An element that describes a single program or implementation-dependent foreign object.

For more information see "The exObject element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The field child

A container element that describes a single interactive container capable of capturing and presenting data content.

For more information see "The field element".

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The relevant property

Specifies the views for which the enclosing object is relevant. The views themselves are specified in the config object.

Views are supplied as a space-separated list of viewnames: `relevant="[+|-]viewname [[+|-]viewname[...]]"`. A token of the form *viewname* or *+viewname* indicates the enclosing element should be included in that particular view. A token of the form `-viewname` indicates the element should be excluded from that particular view.

If a container is excluded, it is not considered in the data binding process.

See also Concealing Containers Depending on View and Config Specification.

The subform child

A container element that describes a single subform capable of enclosing other containers.

For more information see "The subform element".

The subformSet child

An element that describes a set of related subform objects.

For more information see "The subformSet element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The x property

X coordinate of the container's anchor point relative to the top-left corner of the parent container's nominal content region when placed with positioned layout. Defaults to 0.

The y property

Y coordinate of the container's anchor point relative to the top-left corner of the parent container's nominal content region when placed with positioned layout. Defaults to 0.

The assist element

An element that supplies additional information about a container for users of interactive applications.

```
<assist

Properties:
    id="xml-id"
    role="cdata"
    use="cdata"
    usehref="cdata"
>
    <speak>  [0..1]
    <toolTip>  [0..1]
</assist>
```

The assist element is used within the following other elements:
draw exclGroup field proto subform

The assist element provides a means to specify the tool tip and behavior for a spoken prompt.

The id property

A unique identifier that may be used to identify this element as a target.

The role property

Specifies the role played by the parent container. Such a role specification may be used by speech-enabled XFA processing applications to provide information. For example, this attribute may be assigned values borrowed from HTML, such as role="TH" (table headings) and role="TR" (table rows).

The speak property

An audible prompt describing the contents of a container. This element is ignored by non-interactive applications.

For more information see "The speak element".

The toolTip property

An element that supplies text for a tool tip. This element is ignored by non-interactive applications.

For more information see "The toolTip element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The barcode element

An element that represents a barcode.

```
<barcode

Properties:
    charEncoding="UTF-8 | cdata"
    checksum="none | auto | 1mod10 | 2mod10 |
              1mod10_1mod11"
    dataColumnCount="cdata"
    dataLength="cdata"
    dataPrep="none | flateCompress"
    dataRowCount="cdata"
    endChar="cdata"
    errorCorrectionLevel="0 | cdata"
    id="xml-id"
    moduleHeight="5mm | measurement"
    moduleWidth="0.25mm | measurement"
    printCheckDigit="0 | 1"
    rowColumnRatio="cdata"
    startChar="cdata"
    textLocation="below | none | above | aboveEmbedded |
                  belowEmbedded"
    truncate="0 | 1"
    type="cdata"
    use="cdata"
    usehref="cdata"
    wideNarrowRatio="3:1 | cdata"
>
    <encrypt>  [0..1]
    <extras>  [0..1]
</barcode>
```

The barcode element is used within the following other elements:
proto ui

The barcode element supplies the information required to display a barcode. This includes the type of the barcode and a set of options which varies from one type of barcode to another. For more information about using this element see the chapter Using Barcodes.

The charEncoding property

The data written out as a barcode is serialized into a sequence of bytes as specified by this attribute. This has no effect upon the data in the DOM or upon loading data into the DOM.

Note that the value of this property is case-insensitive. For that reason it is defined in the schema as cdata rather than as a list of XML keywords. However the value must match one of the following keywords in a case-insensitive manner.

UTF-8

The characters are encoded using Unicode code points as defined by [Unicode-3.2], and UTF-8 serialization as defined by *ISO/IEC 10646* [ISO-10646]. There is no byte order mark.

none

No special encoding is specified. The characters are encoded using the ambient encoding for the operating system.

ISO-8859-1

The characters are encoded using ISO-8859-1 [ISO-8859-1], also known as Latin-1.

ISO-8859-2

The characters are encoded using ISO-8859-2 [ISO-8859-2].

ISO-8859-7

The characters are encoded using ISO-8859-7 [ISO-8859-7].

Shift-JIS

The characters are encoded using JIS X 0208, more commonly known as Shift-JIS [Shift-JIS].

KSC-5601

The characters are encoded using the *Code for Information Interchange (Hangul and Hanja)* [KSC5601].

Big-Five

The characters are encoded using Traditional Chinese (Big-Five). **Note:** there is no official standard for Big-Five and several variants are in use. XFA uses the variant implemented by Microsoft as code page 950 [Code-Page-950].

GB-2312

The characters are encoded using Simplified Chinese [GB2312].

UTF-16

The characters are encoded using Unicode code points as defined by [Unicode-3.2], and UTF-16 serialization as defined by *ISO/IEC 10646* [ISO-10646]. There is no byte order mark.

UCS-2

The characters are encoded using Unicode code points as defined by [Unicode 3.2], and UCS-2 serialization as defined by *ISO/IEC 10646* [ISO-10646]. There is no byte order mark.

fontSpecific

The characters are encoded in a font-specific way. Each character is represented by one 8-bit byte. The font referred to is font property of the enclosing field or draw.

The checksum property

Algorithm for the checksum to insert into the barcode. For some barcode types this attribute is ignored. For others all or only a subset of the following values is supported.

none

Do not insert a checksum. This is the default and is always allowed (but may be ignored).

auto

Insert the default checksum for the barcode format. Always allowed.

1mod10

Insert a "1 modulo 10" checksum.

2mod10

Insert a "2 modulo 10" checksum.

1mod10_1mod11

Insert a "1 modulo 10" checksum followed by a "1 modulo 11" checksum.

"1 modulo 10", "2 modulo 10", and "1 modulo 11" are defined in barcode standards documents for the barcodes to which they apply.

The dataColumnCount property

(2-d barcodes only.) Optional number of data columns to encode for supported barcodes. The template supplies this property in conjunction with dataRowCount to specify a fixed row and column bar code. The template must not supply the `dataColumnCount` property unless the `dataRowCount` property is also supplied. When these properties are used the size of the bar code is fixed. If the supplied data does not fill the barcode it is padded out with padding symbols.

The dataLength property

(1-d barcodes only.) The expected maximum number of characters for this instance of the barcode.

For software barcodes, when `moduleWidth` is not specified, this property must be supplied by the template. The XFA processor uses this value and the field width, plus its knowledge of the barcode format, to compute the width of a narrow bar. The width of a wide bar is derived from the width of a narrow bar. When `moduleWidth` is specified this property, if present, is ignored by the XFA processor.

For hardware barcodes this parameter is ignored. Because the XFA processor does not know the details of the barcode format, it cannot use this information to determine the bar width.

The dataPrep property

(Recommended for 2-d barcodes only.) Preprocessing applied to the data before it is written out as a barcode. This does not affect the data in the DOMs, nor does it affect what the user sees when the field has focus in interactive contexts.

none

Use the data just as supplied. This is the default.

flateCompress

Write out a header consisting of a byte with decimal value 129 (0x81 hex) followed by another byte with decimal value 1. Then write the data compressed using the Flate algorithm, as defined by the Internet Engineering Task Force (IETF) in [RFC 1951]. No predictor algorithm is used. It is an error to specify this option with a `type` that cannot encode arbitrary binary data.

The dataRowCount property

(2-d barcodes only.) Optional number of data rows to encode for supported barcodes.

The template supplies this property in conjunction with `dataColumnCount` in order to specify a fixed row and column barcode. The dataRowCount property must not be present unless the `dataColumnCount` property is also present. When these properties are used the size of the barcode is

fixed. If the supplied data does not fill the barcode the remaining cells are padded out with padding symbols.

The encrypt property

An element that controls encryption of barcode or submit data.

For more information see "The encrypt element".

The endChar property

Optional ending control character to append to barcode data. This property is ignored by the XFA processor if the barcode pattern does not allow it.

The errorCorrectionLevel property

(2-d barcodes only.) Optional error correction level to apply to supported barcodes. For PDF417 the valid values are integers in the range 0 through 8, inclusive.

For barcode types that accept this property the XFA processor ignores the `checksum` property.

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The moduleHeight property

Module height.

A module is a set of bars encoding one symbol. Usually a symbol corresponds to a character of supplied data. This property determines the height of the bars in the module. The allowable range of heights varies from one barcode pattern to another. The template must not specify a height outside the allowable range.

When this property is not supplied, the default behavior depends on the type of barcode. 1-D barcodes grow to the height of the enclosing field, limited by the allowable height range. 2-D bar codes default to a module height of 5mm.

The moduleWidth property

The property has different meanings for different classes of bar codes.

For 1-d software barcodes the XFA processor sets the width of the narrow bars to the value of this property. The width of the wide bars is derived from that of the narrow bars. The allowable range of widths varies from one barcode format to another. The template must not specify a value outside the allowable range. If `moduleWidth` is supplied the XFA processor ignores the `dataLength` property. Conversely `moduleWidth` has no default, so when `dataLength` is not supplied then `moduleWidth` must be supplied.

For 1-d hardware barcodes `moduleWidth` either has no effect or has the same effect as for a software barcode, depending upon the printer and barcode. However for hardware barcodes the template may fall

back upon the default value for this property. The default is 0.25mm. The allowable range for the value varies between printers and between barcodes.

For 2-d barcodes the value of this property determines the module width. A module is a set of bars encoding one symbol. Usually a symbol corresponds to a character of supplied data. The allowable range of widths varies from one barcode format to another. The template must not specify a value outside the allowable range. The default value for this property (0.25mm) is not useful for 2-d barcodes.

The printCheckDigit property

Specifies whether the check digit(s) is/are printed in the human-readable text.

When the XFA processor is not generating a checksum it ignores this property.

0

Do not print the check digit in the human-readable text, only in the barcode itself. This is the default.

1

Append the check digit to the end of the human-readable text.

The rowColumnRatio property

(2-d barcodes only.) Optional ratio of rows to columns for supported 2-D barcodes.

The XFA processor ignores this property if `dataRowCount` and `dataColumnCount` are specified.

When `rowColumnRatio` is supplied the XFA processor allows the barcode to grow to the number of rows required to hold the supplied data. If the last row is not filled by the supplied data it is padded out with padding symbols.

The startChar property

Optional starting control character to prepend to barcode data.

This property is ignored by the XFA processor if the barcode pattern does not allow it.

The textLocation property

Location, if any, of human-readable text. May be one of:

below

Text is placed below the barcode. This is the default.

above

Text is placed above the barcode.

belowEmbedded

Text is partially embedded in the bottom of the barcode. The baseline of the text is aligned with the bottom of the bars.

aboveEmbedded

Text is partially embedded at the top of the barcode. The top of the text is aligned with the top of the bars.

none

No text is displayed.

The truncate property

Truncates the right edge of the barcode for supported formats. Of the barcodes in the standard types list, this applies only to PDF417. The XFA processor ignores this property for barcode formats to which it does not apply.

0

The right-hand synchronization mark must be included. This is the default.

1

The right-hand synchronization mark must be omitted.

The type property

A string that identifies the barcode pattern.

This property must be supplied. The set of supported values for this property is implementation-defined and may also be specific to the display device. The following values have been defined for this property as indicating particular barcode types:

codabar

Codabar, as defined in ANSI/AIM BC3-1995, USS Codabar [Codabar].

code2Of5Industrial

Code 2 of 5 Industrial; no official standard.

code2Of5Interleaved

Code 2 of 5 Interleaved, as defined in ANSI/AIM BC2-1995, USS Interleaved 2-of-5 [Code2Of5Interleaved].

code2Of5Matrix

Code 2 of 5 Matrix; no official standard.

code2Of5Standard

Code 2 of 5 Standard; no official standard.

code3Of9

Code 39 (also known as code 3 of 9), as defined in ANSI/AIM BC1-1995, USS Code 39 [Code39].

code3Of9extended

Code 39 extended; no official standard.

code11

Code 11 (USD-8); no official standard.

code49

Code 49, as defined in ANSI/AIM BC6-1995, USS Code 49 [Code49].

code93

Code 93, as defined in ANSI/AIM BC5-1995, USS Code 93 [Code93].

code128

Code 128, as defined in ANSI/AIM BC4-1995, ISS Code 128 [Code128-1995].

code128A

Code 128 A, as defined in ANSI/AIM BC4-1995, ISS Code 128 [Code128-1995].

code128B

Code 128 B, as defined in ANSI/AIM BC4-1995, ISS Code 128 [Code128-1995].

code128C

Code 128 C, as defined in ANSI/AIM BC4-1995, ISS Code 128 [Code128-1995].

code128SSCC

Code 128 serial shipping container code, as defined in ANSI/AIM BC4-1995, ISS Code 128 [Code128-1995].

ean8

EAN-8, as defined in ISO/EEC 15420 [ISO-15420]

ean8add2

EAN-8 with 2-digit Addendum, as defined in ISO/EEC 15420 [ISO-15420]

ean8add5

EAN-8 with 5-digit Addendum, as defined in ISO/EEC 15420 [ISO-15420]

ean13

EAN-13, as defined in ISO/EEC 15420 [ISO-15420]

ean13pwcd

EAN-13 with Price/Weight customer data, as defined in ISO/EEC 15420 [ISO-15420]

ean13add2

EAN-13 with 2-digit Addendum, as defined in ISO/EEC 15420 [ISO-15420]

ean13add5

EAN-13 with 5-digit Addendum, as defined in ISO/EEC 15420 [ISO-15420]

fim

United States Postal Service FIM (Facing Identification Mark), as described in First-Class Mail [USPS-C100].

logmars

Logmars (Logistics Applications of Automated Marking and Reading Symbols) as defined by U.S. Military Standard MIL-STD-1189B [LOGMARS].

maxicode

UPS Maxicode, as defined in ANSI/AIM BC10-ISS Maxicode [Maxicode].

msi

MSI (modified Plessey); may have once had a formal specification but not any longer.

pdf417

PDF417, as defined in USS PDF417 [PDF417].

pdf417macro

PDF417, but allowing the data to span multiple PDF417 bar codes. The barcode(s) are marked so that the barcode reader knows when it still has additional barcodes to read, and can if necessary prompt the operator. This facility is defined in "USS PDF417" [PDF417].

plessey

Plessey; no official standard.

postAUSCust2

Australian Postal Customer 2, as defined in Customer Barcoding Technical Specifications [APO-Barcode].

postAUSCust3

Australian Postal Customer 3, as defined in Customer Barcoding Technical Specifications [APO-Barcode].

postAUSReplyPaid

Australian Postal Reply Paid, as defined in Customer Barcoding Technical Specifications [APO-Barcode].

postAUSStandard

Australian Postal Standard, as defined in Customer Barcoding Technical Specifications [APO-Barcode].

postUKRM4SCC

United Kingdom RM4SCC (Royal Mail 4-State Customer Code), as defined in the How to Use Mailsort Guide [RM4SCC].

postUSDPBC

United States Postal Service Delivery Point Bar Code, as defined in DMM C840 Barcoding Standards for Letters and Flats [USPS-C840].

postUSStandard

United States Postal Service POSTNET barcode (Zip+4), as defined in DMM C840 Barcoding Standards for Letters and Flats [USPS-C840].

postUSZip

United States Postal Service POSTNET barcode (5 digit Zip), as defined in DMM C840 Barcoding Standards for Letters and Flats [USPS-C840].

qr

QR Code, as defined in ISS - QR Code [QRCode].

telepen

Telepen, as defined in USS Telepen [Telepen].

ucc128

UCC/EAN 128, as defined in International Symbology Specification - Code 128 (1999) [Code128-1999].

ucc128random

UCC/EAN 128 Random Weight, as defined in International Symbology Specification - Code 128 (1999) [Code128-1999].

ucc128sscc

UCC/EAN 128 serial shipping container code (SSCC), as defined in International Symbology Specification - Code 128 (1999) [Code128-1999].

upcA

UPC-A, as defined in ISO/EEC 15420 [ISO-15420].

upcAadd2

UPC-A with 2-digit Addendum, as defined in ISO/EEC 15420 [ISO-15420].

upcAadd5

UPC-A with 5-digit Addendum, as defined in ISO/EEC 15420 [ISO-15420].

upcApwcd

UPC-A with Price/Weight customer data, as defined in ISO/EEC 15420 [ISO-15420].

upcE

UPC-E, as defined in ISO/EEC 15420 [ISO-15420].

upcEadd2

UPC-E with 2-digit Addendum, as defined in ISO/EEC 15420 [ISO-15420].

upcEadd5

UPC-E with 5-digit Addendum, as defined in ISO/EEC 15420 [ISO-15420].

upcean2

UPC/EAN with 2-digit Addendum, as defined in ISO/EEC 15420 [ISO-15420].

upcean5

UPC/EAN with 5-digit Addendum, as defined in ISO/EEC 15420 [ISO-15420].

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The wideNarrowRatio property

Ratio of wide bar to narrow bar in supported barcodes.

The allowable range of ratios varies between barcode formats and also, for hardware barcodes, the output device. The template must not specify a value outside the allowable range. The XFA processor ignores this property for barcode formats which do not allow a variable ratio of wide to narrow bar widths. The default value for this property is 3:1.

The syntax for the value of this property is *wide*[:*narrow*] where:

wide is a positive number representing the numerator of the ratio, and

narrow is an optional positive number representing the denominator of the ratio. If *narrow* is not supplied it defaults to `1`.

The bind element

An element that controls the behavior during merge operations of its enclosing element.

```
<bind

Properties:
    match="once | none | global | dataRef"
    ref="cdata"
>
    <picture>  [0..1]
</bind>
```

The bind element is used within the following other elements:
exclGroup field subform

The match property

Controls the role played the by enclosing element in a data-binding (merge) operation.

once

The node representing the enclosing element will bind to a node in the XFA Data DOM in accordance with the standard matching rules.

none

The node representing the enclosing element is transient. It will not be bound to any node in the XFA Data DOM.

global

The containing field is global. If the normal matching rules fail to provide a match for it, the data-binding process will look outside the current record for data to bind to the field.

dataRef

The containing field will bind to the node in the XFA Data DOM specified by the accompanying `ref` attribute.

See Basic Data Binding to Produce the XFA Form DOM for more information about, and an authoritative definition of, the effects of this property.

The picture property

A rendering element that describes input parsing and output formatting information.

For more information see "The picture element".

The ref property

An XFA SOM expression defining the node in the XFA Data DOM to which the enclosing container will bind. This is used only when the `match` attribute has the value `dataRef`.

See the *XFA-Scripting Object Model Expression Specification* [XFA-SOM] for more information about XFA SOM expressions.

The bindItems element

An element that extracts data into an item list.

```
<bindItems

Properties:
    connection="cdata"
    labelRef="cdata"
    ref="cdata"
    valueRef="cdata"
>
</bindItems>
```

The bindItems element is used within the following other elements:
field proto

This element builds the items list for a choicelist or a set of check boxes or radio buttons. However unlike the `items` element this element gets the data from the Data DOM or from a connection to a web service.

The connection property

An optional attribute that, if present, supplies the name of a connection for a web service. If this attribute is supplied it alters the meaning of the ref property such that it is interpreted according to the same rules as the ref attribute of the `connect` element.

The labelRef property

An optional attribute that, if present, tells where to find the data value to use as a label for each item. The value of this property is a SOM expression which is relative to a node selected by the ref attribute. If this attribute is not supplied or empty each item is labelled with its value.

The ref property

A SOM expression that selects a set of nodes, each of which corresponds to an item in the list. If there is a connection attribute then the value of this property is a SOM expression interpreted in relation to that connection according to the same rules as the ref attribute of the `connect` element. However if there is no connection attribute then the value of this attribute is an ordinary SOM expression interpreted relative to the data node to which its containing object is bound.

The valueRef property

An attribute that tells where to find the data value for each item. The value of this property is a SOM expression which is relative to a node selected by the ref attribute.

The bookend element

An element controlling content that is inserted to "bookend" the contents of the parent object.

```
<bookend

Properties:
     id="xml-id"
     leader="cdata"
     trailer="cdata"
     use="cdata"
     usehref="cdata"
>
</bookend>
```

The bookend element is used within the following other elements:
proto subform subformSet

The id property

A unique identifier that may be used to identify this element as a target.

The leader property

The value of this property is either a SOM expression (which can not start with '#') or a '#' followed by an XML ID. The SOM expression or XML ID points to a subform or subform set to be laid down before the content of the parent object. When this property is empty or blank no leader is laid down.

Note that this replaces the `bookendTrailer` attribute on the deprecated `break` element.

The trailer property

The value of this property is either a SOM expression (which can not start with '#') or a '#' followed by an XML ID. The SOM expression or XML ID points to a subform or subform set to be laid down after the content of the parent object. When this property is empty or blank no trailer is laid down.

Note that this replaces the `bookendLeader` attribute on the deprecated `break` element.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The boolean element

A content element describing single unit of data content representing a Boolean logical value.

```
<boolean

Properties:
    id="xml-id"
    name="xml-id"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</boolean>
```

The boolean element is used within the following other elements:
desc exObject extras items proto value variables

Content

The content must be one of the following:

`0`

The content represents a logical value of false.

`1`

The content represents a logical value of true.

When no content is present, the content shall be interpreted as representing a null value, irrespective of the value of the associated `nullType` property in the data description.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The border element

A box model element that describes the border surrounding an object.

```
<border

Properties:
    break="close | open"
    hand="even | left | right"
    id="xml-id"
    presence="visible | invisible | hidden"
    relevant="cdata"
    use="cdata"
    usehref="cdata"
>
    <corner>  [0..4]
    <edge>  [0..4]
    <extras>  [0..1]
    <fill>  [0..1]
    <margin>  [0..1]
</border>
```

The border element is used within the following other elements:
checkButton choiceList dateTimeEdit draw exclGroup field imageEdit numericEdit passwordEdit proto signature subform textEdit

The edges of a border are rendered in a clockwise fashion, starting from the top left corner. This has implications for the border's handedness. In particular, a left-handed stroke will appear immediately outside the rectangle's edge, while a right-handed edge will appear immediately inside. Such behavior is consistent with rectangles, but not arcs.

The break property

(DEPRECATED) An element that describes the constraints on moving to a new page or content area before or after rendering an object.

The corner property

A formatting element that describes the appearance of a vertex between two edges

For more information see "The corner element".

The edge property

A formatting element that describes an arc, line, or one side of a border or rectangle.

For more information see "The edge element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The fill property

A formatting element that applies a color and optional rendered designs to the region enclosed by an object.

For more information see "The fill element".

The hand property

Description of the handedness of a line or edge.

even

Center the displayed line on the underlying vector or arc.

left

Position the displayed line immediately to the left of the underlying vector or arc, when following that line from its start point to its end point.

right

Position the displayed line immediately to the right of the underlying vector or arc, when following that line from its start point to its end point.

The id property

A unique identifier that may be used to identify this element as a target.

The margin property

A box model element that specifies one or more insets for an object.

For more information see "The margin element".

The presence property

Visibility control.

visible

Make it visible.

invisible

Make it transparent. Although invisible it still takes up space.

hidden

Hide it. It is not displayed and does not take up space.

The relevant property

Specifies the views for which the enclosing object is relevant. The views themselves are specified in the config object.

Views are supplied as a space-separated list of viewnames: `relevant="[+|-]viewname [[+|-]viewname[...]]"`. A token of the form *viewname* or *+viewname* indicates the enclosing element should be included in that particular view. A token of the form `-viewname` indicates the element should be excluded from that particular view.

If a container is excluded, it is not considered in the data binding process.

See also Concealing Containers Depending on View and Config Specification.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The break element

(DEPRECATED) An element that describes the constraints on moving to a new page or content area before or after rendering an object.

```
<break

Properties:
    after="auto | contentArea | pageArea | pageEven |
           pageOdd"
    afterTarget="cdata"
    before="auto | contentArea | pageArea | pageEven |
            pageOdd"
    beforeTarget="cdata"
    bookendLeader="cdata"
    bookendTrailer="cdata"
    id="xml-id"
    overflowLeader="cdata"
    overflowTarget="cdata"
    overflowTrailer="cdata"
    startNew="0 | 1"
    use="cdata"
    usehref="cdata"
>
    <extras>  [0..1]
</break>
```

The break element is used within the following other elements:
proto subform subformSet

As of XFA 2.4 this element has been deprecated. New designs should use the overflow, bookend, breakBefore, and/or breakAfter elements instead.

The after property

This property specifies the constraints on moving to a new page or content area after rendering the subform.

The behaviors described below can be further refined by optionally specifying a destination page or content area via the afterTarget attribute.

auto

The determination of a transition to a new page or content area will be delegated to the processing application. No transition to a new page or content area will be forced.

contentArea

Rendering will transition the next available content area.

pageArea

Rendering will transition to a new page.

The afterTarget property

Specifies the explicit destination page or content area for the after property. The content of this property is a '#' character followed by an XML ID.

The value of property is expected to be compatible with the value of the after property. For instance, it would be considered an error for the after property to have a value of pageArea and the afterTarget property to reference a content area, or vice versa.

The before property

Specifies the constraints on moving to a new page or content area before rendering the subform.

The behaviors described below can be further refined by optionally specifying a destination page or content area via the beforeTarget attribute. The startNew attribute also modifies some of these behaviors.

auto

The determination of a transition to a new page or content area will be delegated to the processing application. No transition to a new page or content area will be forced.

contentArea

Rendering will transition the next available content area. See also the startNew attribute.

pageArea

Rendering will transition to a new page.See also the startNew attribute.

The beforeTarget property

This property specifies the explicit destination page or contentArea for the before property. The content of this property is a '#' character followed by an XML ID.

The value of the beforeTarget property is expected to be compatible with the value of the before property. For instance, it would be considered an error for the before property to have a value of pageArea and the beforeTarget property to reference a content area, or vice versa.

The bookendLeader property

Identifies a subform which is to be placed into the current content area or page before any other content. The content of this property is a '#' character followed by an XML ID.

If both bookendLeader and bookendTrailer are supplied the two subforms bracket the content in the manner of bookends.

The bookendTrailer property

Identifies a subform which is to be placed into the current content area or page after any other content. The content of this property is a '#' character followed by an XML ID.

If both bookendLeader and bookendTrailer are supplied the two subforms bracket the content in the manner of bookends.

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The overflowLeader property

Identifies a subform which is to be placed at the top of the content area or page when it is entered as a result of an overflow. The content of this property is either a '#' character followed by an XML ID.

The overflowTarget property

Specifies the explicit destination page or contentArea that shall be the transition target when the current content area or page area has been overflowed. The content of this property is a '#' character followed by an XML ID.

The overflowTrailer property

Identifies a subform which is to be placed at the bottom of the content area or page when it overflows. The vertical space required for the overflow trailer must be reserved. The content of this property is a '#' character followed by an XML ID.

The startNew property

Determines whether it is necessary to start a new content area or page even when the current content area or page has the required name.This attribute has no effect unless the `before` attribute has the value `contentArea` or `pageArea`.

`0`

Do not start a new content area or page area if the current one has the specified name.

`1`

Always start a new content area or page.

The name of the content area or page is supplied by the accompanying `beforeTarget` attribute.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The breakAfter element

An element that controls actions to be taken after laying down the contents of the parent object.

```
<breakAfter

Properties:
    id="xml-id"
    leader="cdata"
    startNew="0 | 1"
    target="cdata"
    targetType="auto | contentArea | pageArea | pageEven |
                pageOdd"
    trailer="cdata"
    use="cdata"
    usehref="cdata"
>
    <script>  [0..1]
</breakAfter>
```

The breakAfter element is used within the following other elements:
proto subform subformSet

When layout of the parent object is complete and this element contains a non-empty script the script is evaluated. If the script returns false no break occurs and layout proceeds using the current layout container. However if the script returns true, or if there is no script, or the script is empty, a break occurs and various actions occur under control of the other properties of this element.

The id property

A unique identifier that may be used to identify this element as a target.

The leader property

The value of this property is either a SOM expression (which can not start with '#') or a '#' followed by an XML ID. The SOM expression or XML ID points to a subform or subform set to be laid down after all other actions of the break are complete (for example transitioning to a new page). When this property is empty or blank no leader is laid down.

The script property

An automation element that contains a script.

For more information see "The script element".

The startNew property

Controls whether or not to start a new layout container if the current layout container matches the target specification.

0

If the current layout container matches the target specification continue filling the same container. Only start a new layout container when the current one does not match the target specification.

1

Always start a new layout container.

The target property

The value of this property is either a SOM expression (which can not start with '#') or a '#' followed by an XML ID. The SOM expression or XML ID points to a layout container which may or may not be the current layout container. When this property is empty or blank layout continues in the existing container.

Note that this replaces the `afterTarget` attribute on the deprecated `break` element.

The targetType property

Controls movement to new a layout container after laying out the content of the parent object.

auto

Layout continues using the current layout container.

contentArea

Layout transitions to the next available content area.

pageArea

Layout transitions to a new page.

Note that this replaces the `after` attribute on the deprecated `break` element.

The trailer property

The value of this property is either a SOM expression (which can not start with '#') or a '#' followed by an XML ID. The SOM expression or XML ID points to a subform or subform set to be laid down after the content of the parent object but before any other actions of the break such as transitioning to a new page. When this property is empty or blank no trailer is laid down.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The breakBefore element

An element that controls actions to be taken before laying down the contents of the parent object.

```
<breakBefore

Properties:
    id="xml-id"
    leader="cdata"
    startNew="0 | 1"
    target="cdata"
    targetType="auto | contentArea | pageArea | pageEven |
                pageOdd"
    trailer="cdata"
    use="cdata"
    usehref="cdata"
>
    <script>  [0..1]
</breakBefore>
```

The breakBefore element is used within the following other elements:
proto subform subformSet

When layout of the parent object is about to start and this element contains a non-empty script the script is evaluated. If the script returns false no break occurs and layout proceeds using the current layout container. However if the script returns true, or if there is no script, or the script is empty, a break occurs and various actions occur under control of the other properties of this element.

The id property

A unique identifier that may be used to identify this element as a target.

The leader property

The value of this property is either a SOM expression (which can not start with '#') or a '#' followed by an XML ID. The SOM expression or XML ID points to a subform or subform set to be laid down before the content of the parent object but after other actions of the break such as starting a new page. When this property is empty or blank no leader is laid down.

The script property

An automation element that contains a script.

For more information see "The script element".

The startNew property

Controls whether or not to start a new layout container if the current layout container matches the target specification.

0

If the current layout container matches the target specification continue filling the same container. Only start a new layout container when the current one does not match the target specification.

1

Always start a new layout container.

The target property

The value of this property is either a SOM expression (which can not start with '#') or a '#' followed by an XML ID. The SOM expression or XML ID points to a layout container which may or may not be the current layout container. When this property is empty or blank layout continues in the existing container.

Note that this replaces the `beforeTarget` attribute on the deprecated `break` element.

The targetType property

Controls movement to new a layout container before laying out the content of the parent object.

auto

Layout continues using the current layout container.

contentArea

Layout transitions to the next available content area.

pageArea

Layout transitions to a new page.

Note that this replaces the `before` attribute on the deprecated `break` element.

The trailer property

The value of this property is either a SOM expression (which can not start with '#') or a '#' followed by an XML ID. The SOM expression or XML ID points to a subform or subform set to be laid down before the content of the parent object and before any other actions of the break such as staring a new page. When this property is empty or blank no trailer is laid down.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The button element

A user interface element that describes a push-button widget.

```
<button

Properties:
    highlight="push | none | inverted | outline"
    id="xml-id"
    use="cdata"
    usehref="cdata"
>
    <extras>  [0..1]
</button>
```

The button element is used within the following other elements:
proto ui

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The highlight property

Controls the graphic treatment of the button when activated in an interactive environment.

push

The button presents an appearance similar to a mechanical button being pushed. It has three appearances, up (inactive), down (active) and rollover (highlighted but retaining its current state unless and until toggled by a mouse click).

inverted

The button (within the frame) is inverted in shading and/or color when activated. There is no special rollover appearance.

none

There is no special change in graphic appearance when activated. The only change is that specified by the mark attribute of the associated checkButton element. There is no special rollover appearance.

outline

The frame of the button is inverted in shading and/or color when activated. There is no special rollover appearance.

When (and only when)push mode is selected, alternate captions may be specified for the down and rollover states. The alternate captions are specified using an items list containing named items. The item named "down", if present, is used in the down state and the item named "rollover", if present, is used in the rollover state. For example,

```
<field>
   <ui>
      <button/>
   </ui>
   <caption>
      <value><text> Up Text </text></value>
   </caption>
   <items>
      <text name="down"> Down Text </text>
      <text name="rollover"> Rollover Text </text>
   </items>
</field>
```

The id property

A unique identifier that may be used to identify this element as a target.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The calculate element

An automation element that controls the calculation of its container's value.

```
<calculate

Properties:
    id="xml-id"
    override="disabled | warning | error | ignore"
    use="cdata"
    usehref="cdata"
>
    <extras>  [0..1]
    <message>  [0..1]
    <script>  [0..1]
</calculate>
```

The calculate element is used within the following other elements:
exclGroup field proto subform

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The message property

A automation element that holds one or more sub-elements containing messages used with validations and calculations.

For more information see "The message element".

The override property

Determines whether the calculated value can be overridden by the user in an interactive context, or disables calculation in any context.

As one would expect, every field object has a calculate property, whether or not the XML representation of the field contains a calculate element. Unusually for XFA, the default value for the calculate object's override property differs depending upon whether the XML representation includes or does not include a calculate element.

`disabled`

The calculation is disabled. This is the default when the XML representation of the field *does not* include a calculate element. In an interactive context the user is free to enter data into the field. The effect of this override value is independent of user action. The `disabled` value allows an event script to dynamically enable/disable a calculate element.

error

The calculation is enabled. This is the default when the XML representation of the field *does* include a calculate element. The processing application must not change the calculated value with any user-supplied values. User attempts to directly set the value derived by a calculate object having an error override causes the processing application to present an error message. To avoid the need for such error messages, form creators may wish to define such fields as read-only.

ignore

The calculated value is mandatory. The processing application ignores any attempt by the user to set the value of the form object.

warning

The calculation is enabled, and the calculated value is recommended over user-supplied values. If the user takes action to directly set the value of the form object, the processing application presents a warning message. The message informs the user that the form object is recommended to have a calculated value, and provides the user with two choices: dismiss or override. Users select dismiss to indicate they wish to leave the calculated value undisturbed. Users select override to indicate they understand the form's recommendation, but have chosen to override the calculated value with a value of their choosing. The application does not issue any warnings or prompts on subsequent gain of focus by the same object.

The script property

An automation element that contains a script.

For more information see "The script element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The caption element

A box model element that describes a descriptive label associated with an object.

```
<caption

Properties:
    id="xml-id"
    placement="left | top | right | bottom |
               inline"
    presence="visible | invisible | hidden"
    reserve="-1 | measurement"
    use="cdata"
    usehref="cdata"
>
    <extras>  [0..1]
    <font>  [0..1]
    <margin>  [0..1]
    <para>  [0..1]
    <value>  [0..1]
</caption>
```

The caption element is used within the following other elements:
draw exclGroup field proto

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The font property

A formatting element that describes a font.

For more information see "The font element".

The id property

A unique identifier that may be used to identify this element as a target.

The margin property

A box model element that specifies one or more insets for an object.

For more information see "The margin element".

The para property

A formatting element that specifies default paragraph and alignment properties to be applied to the content of an enclosing container.

For more information see "The para element".

The placement property

Specifies the placement of the caption.

`left`

The caption is located to the left of the content in a rectangular region that spans the height of the margined nominal extent.

`right`

The caption is located to the right of the content in a rectangular region that spans the height of the margined nominal extent.

`top`

The caption is located above the content in a rectangular region that spans the width of the margined nominal extent.

`bottom`

The caption is located below the content in a rectangular region that spans the width of the margined nominal extent.

`inline`

The caption appears inline with, and prior to, the text content.

The presence property

Visibility control.

`visible`

Make it visible.

`invisible`

Make it transparent. Although invisible it still takes up space.

`hidden`

Hide it. It is not displayed and does not take up space.

The reserve property

A measurement value that specifies the height or width of the caption.

The effect of this property is determined by the `placement` property. When the caption is placed at the left or right the `reserve` property specifies the width of the caption region. When the caption is placed at the top or bottom the `reserve` property specifies the height. When the caption is placed inline the `reserve` property is ignored.

When this attribute is omitted or has a value of "0" the height or width (as appropriate) is just enought to hold the content of the caption and text auto-wrapping does not occur.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The value property

A content element that encloses a single unit of data content.

For more information see "The value element".

The certificate element

An element that holds a suitable Base64 DER-encoded X.509v3 certificate.

```
<certificate

Properties:
    id="xml-id"
    name="xml-id"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</certificate>
```

The certificate element is used within the following other elements:
encrypt issuers proto signing

A certificate binds a person or entity to a specific public key. 509v3 certificates are described in RFC 3280, Internet X.509 Public Key Infrastructure, Certificate and Certificate Revocation List (CRL) Profile [RFC3280].

Content

A Base64 DER-encoded X.509v3 certificate.

Depending upon the context this element can contain a certificate holding either a public key or a private key. When used for encryption, as a property of a barcode element, the certificate holds a public key. When used for authenticating a signing certificate, as a property of an issuers element, it also holds a public key. But when used for signing, as a property of a signing element, it holds a private key.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The certificates element

An element that holds a collection of certificate filters used to identify the signer.

```
<certificates

Properties:
    credentialServerPolicy="optional | required"
    id="xml-id"
    url="cdata"
    urlPolicy="enrollmentServer | cdata"
    use="cdata"
    usehref="cdata"
>
    <issuers>  [0..1]
    <keyUsage>  [0..1]
    <oids>  [0..1]
    <signing>  [0..1]
    <subjectDNs>  [0..1]
</certificates>
```

The certificates element is used within the following other elements:
filter proto

The `certificates` element identifies certificates used for Public Key Infrastructure (PKI), including signing certificates, issuer certificates, and object IDs. Issuer certificates and object IDs are used to verify the signing certificate is valid. PKI information allows the document recipient to determine whether or not a specific public key really belongs to a specific individual. X.509v3 certificates are described in RFC 3280, Internet X.509 Public Key Infrastructure, Certificate and Certificate Revocation List (CRL) Profile [RFC3280]. The document "A primer on electronic security" [ElectronicSecurity] provides a more basic explanation of the roles of certificates in signer identification.

The credentialServerPolicy property

Determines Whether the signer must use the given URI or whether another may be used.

`optional`

Another URI may be used.

`required`

Only the supplied URI may be used.

The id property

A unique identifier that may be used to identify this element as a target.

The issuers property

A collection of issuer certificates that are acceptable for data signing an XML digital signature.

For more information see "The issuers element".

The keyUsage property

An element that specifies the key usage settings required in the signing certificate.

For more information see "The keyUsage element".

The oids property

A collection of Object Identifiers (OIDs) which apply to signing data with an XML digital signature.

For more information see "The oids element".

The signing property

A collection of signing certificates that are acceptable for use in affixing an XML digital signature.

For more information see "The signing element".

The subjectDNs property

An element that contains the collection of key-value pairs used to specify the Subject Distinguished Name (DN) that must be present within the certificate for it to be acceptable for signing.

For more information see "The subjectDNs element".

The url property

A URI that can be used to obtain a new credential if a matching credential is not found.

The urlPolicy property

The type of server which is at the associated URI.

`enrollmentServer`

A server where the user can enrol via browser for a new credential. The equivalent in PDF is an "urlType" of "Browser".

`roamingCredentialServer`

A signature web service for server-based signing. The equivalent in PDF is an "urlType" of "ASSP".

`userDefinedString`

Third parties can extend the meaning with custom values. The custom values must conform to the guidelines described in appendix E of the PDF Manual.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The checkButton element

A user interface element that describes either a checkbox or radio-button widget.

```
<checkButton

Properties:
    id="xml-id"
    mark="default | check | circle | cross |
          diamond | square | star"
    shape="square | round"
    size="10pt | measurement"
    use="cdata"
    usehref="cdata"
>
    <border>  [0..1]
    <extras>  [0..1]
    <margin>  [0..1]
</checkButton>
```

The checkButton element is used within the following other elements:
proto ui

The border property

A box model element that describes the border surrounding an object.

For more information see "The border element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The margin property

A box model element that specifies one or more insets for an object.

For more information see "The margin element".

The mark property

This property controls the appearance of the checkButton when its state is asserted.

`default`

The asserted appearance matches the unasserted appearance, as controlled by the `shape` attribute. If the unasserted appearance is a square outline then the asserted appearance is a corner to corner X. If the unasserted appearance is a circular outline then the asserted appearance is a filled circle.

check

The asserted appearance is a check mark.

circle

The asserted appearance is a filled circle.

cross

The asserted appearance is a corner-to-corner X.

square

The asserted appearance is a filled square.

star

The asserted appearance is a filled star.

These descriptions are deliberately vague to allow latitude for the application. For example when presenting on glass the filled circle might appear as a glowing light whereas when presenting on paper it would probably appear as a flat black circle taken from a font.

The shape property

This property controls the appearance of the checkButton when its state is unasserted.

square

The checkButton appears as a square outline. This is usually used to represent an unchecked box.

round

The CheckButton appears as a circular outline. This is usually used to represent an unpressed radio-button.

The size property

A measurement specifying the size of the checkbox or radio-button outline representing either the height/width for a square or the diameter for a circle. The default is `10pt`.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The choiceList element

A user interface element that describes a widget presenting a list of options. The list of options is specified by one or more sibling items elements.

```
<choiceList

Properties:
    commitOn="select | exit"
    id="xml-id"
    open="userControl | onEntry | always | multiSelect"
    textEntry="0 | 1"
    use="cdata"
    usehref="cdata"
>
    <border>   [0..1]
    <extras>   [0..1]
    <margin>   [0..1]
</choiceList>
```

The choiceList element is used within the following other elements:
proto ui

The border property

A box model element that describes the border surrounding an object.

For more information see "The border element".

The commitOn property

This property specifies when the user's selections are propagated to the XFA Data DOM.

select

When the user clicks choice-list data with the mouse, the selected data value is written to the XFA Data DOM. Alternatively, if the user presses the enter key after previously using other keyboard sequences to shift focus to a particular entry in the choice list, the selected data is written to the XFA Data DOM.

Note: Having a choice list commit data as soon as selections are made may be important in forms that contain non-XFA interactive features, such as Acrobat annotations or hypertext links. People filling out such forms may erroneously believe that selecting an item from a choice list followed by clicking a non-XFA interactive feature is the same as exiting the check list. In fact, the check list remains the field in focus.

exit

The selected data is not written to the XFA Data DOM until the field loses focus. This is the recommended setting for choice lists that support multiple selections (`open="multiSelect"`).

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The margin property

A box model element that specifies one or more insets for an object.

For more information see "The margin element".

The open property

This property determines when the drop-down choice list is presented by interactive applications.

userControl

The list drops down when the user clicks on a button or makes some other appropriate gesture. It disappears when the cursor moves outside the list or some other appropriate user-interface event occurs.

onEntry

The list drops down on entry into the field. It disappears upon exit from the field.

always

The list is displayed whenever the field is visible.

multiSelect

The user may select multiple entries from the list, by holding down the shift key while making selections. The list of choices is displayed whenever the field is visible.

The textEntry property

This property determines whether the user is allowed to enter the value by typing it.

0

The user is not allowed to type. The value must be chosen by selecting from the drop-down list.

1

The user is allowed to type or select from the drop-down list. This opens up the field value to be anything that the user might type; the list becomes more like a set of hints. If the `open` attribute is set to `multiSelect`, the user is not allowed to enter values in the widget.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The color element

An element that describes a color.

```
<color

Properties:
    cSpace="SRGB | cdata"
    id="xml-id"
    use="cdata"
    usehref="cdata"
    value="0,0,0 | cdata"
>
    <extras>  [0..1]
</color>
```

The color element is used within the following other elements:
corner edge fill linear pattern proto radial stipple

The cSpace property

This property specifies the color space. The default, and currently the only allowed value, is `SRGB`.

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The value property

This property specifies a comma separated list of values for each color component of the color space.

For the color-space of SRGB, the component values must be `r,g,b`, where *r* is the red component value, *g* is the green component value, and *b* is the blue component value. Each component value must be in the range 0 through 255, inclusive. 255 represents maximum display intensity. For example, `255,0,0` specifies the color red.

The default is dependent upon the context of where the color is used; the default color is determined by the object enclosing the color element.

The comb element

An element that causes a field to be presented with vertical lines between the character positions.

```
<comb

Properties:
    id="xml-id"
    numberOfCells="0 | integer"
    use="cdata"
    usehref="cdata"
>
</comb>
```

The comb element is used within the following other elements:
dateTimeEdit numericEdit proto textEdit

Comb fields must be single-line fields and do not support scrolling.

Each tine of the comb is the full height of the field. Each tine's color and width is that defined for the widget border's third (bottom) edge.

Rich text is not supported in comb fields.

The id property

A unique identifier that may be used to identify this element as a target.

The numberOfCells property

Specifies the number of character positions available for input or display.

The default for this attribute is 0. When the value is 0 the XFA processor falls back upon the `maxChars` property of the associated text element. If that is also 0 or unspecified then the XFA processor draws one cell.

Number of cells and number of characters are not the same thing. In languages using accents or other combining characters the combining characters count as characters but do not contribute to the cell count.

When merging with data from an XML data document the data may exceed the number of cells and/or characters allotted. In this case the data is preserved at the cost of a less than optimal appearance. For example if the form is being printed the excess data may overprint other text in the direction of text flow.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The connect element

An element that describes the relationship between its containing object and a connection to a web service, schema, or data description. Connections are defined outside the template in a separate packet with its own schema. See the XFA Connection Set Specification for more information.

```
<connect

Properties:
    connection="cdata"
    id="xml-id"
    ref="cdata"
    usage="exportAndImport | exportOnly | importOnly"
    use="cdata"
    usehref="cdata"
>
    <picture>  [0..1]
</connect>
```

The connect element is used within the following other elements:
exclGroup field proto subform

The connection property

The `name` of the associated `connection` element in the connection set.

The id property

A unique identifier that may be used to identify this element as a target.

The picture property

A rendering element that describes input parsing and output formatting information.

For more information see "The picture element".

The ref property

A modified XFA-SOM expression pointing to the node in the message or data document corresponding to the containing object.

When the connection is a web service, the message resides under `!connectionData.`*`connectionName`* where *connectionName* is the value of `name`.

The schema of the message is defined by the connection. The value of this property must match a node that is in the message. Furthermore, within a set of `connect` elements sharing the same `name` each connect element must point to a unique message node.

The rules for relative referencing are different in this context than in any other context using XFA-SOM expressions. Normally in XFA SOM expressions the current location ("$") is the container for the property the asserts the expression. Hence relative expressions are relative to the container. However in this context the value of "$" is inherited from the nearest ancestor that asserts a fully-qualified XFA SOM expression as its value of `ref` for the same connection. For example if a subform has a `ref` attribute with a value of `!connectionData.queryDatabase.body` then its child could use the relative expression `queryID` as a synonym for `!connectionData.queryDatabase.body.queryID`. In all other ways the value of

this property is a normal XFA SOM expression. See the *XFA-Scripting Object Model Expression Specification* [XFA SOM] for more information about XFA SOM expressions.

The usage property

The context(s) in which the connection is to be used. The value of this property must be one of the following:

`exportAndImport`

Used during both import and export. This value is allowed both for connections to web services and connections to XML data documents.

`exportOnly`

Used during export, ignored during import. This value is only allowed for connections to web services.

`importOnly`

Used during import, ignored during export. This value is only allowed for connections to web services.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The contentArea element

An element that describes a region within a page area eligible for receiving content.

```
<contentArea

Properties:
    h="0in | measurement"
    id="xml-id"
    name="xml-id"
    relevant="cdata"
    use="cdata"
    usehref="cdata"
    w="0in | measurement"
    x="0in | measurement"
    y="0in | measurement"
>
    <desc>  [0..1]
    <extras>  [0..1]
</contentArea>
```

The contentArea element is used within the following other elements:
pageArea proto

The desc property

An element to hold human-readable metadata.

For more information see "The desc element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The h property

Height for layout purposes. A measurement value for `h` overrides any growth range allowed by the `minH` and `maxH` attributes. The absolute omission of this attribute or a value specified as an empty string indicates that the `minH` and `maxH` must be respected.

This attribute has no default. Setting this attribute to "-1" is an error.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The relevant property

Specifies the views for which the enclosing object is relevant. The views themselves are specified in the config object.

Views are supplied as a space-separated list of viewnames: `relevant="[+|-]viewname [[+|-]viewname[...]]"`. A token of the form *viewname* or +*viewname* indicates the enclosing element should be included in that particular view. A token of the form `-viewname` indicates the element should be excluded from that particular view.

If a container is excluded, it is not considered in the data binding process.

See also Concealing Containers Depending on View and Config Specification.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The w property

Width for layout purposes. A measurement value for `w` overrides any growth range allowed by the `minW` and `maxW` attributes. The absolute omission of this attribute or a value specified as an empty string indicates that the `minW` and `maxW` must be respected.

This attribute has no default. Setting this attribute to "-1" is an error.

The x property

X coordinate of the container's anchor point relative to the top-left corner of the parent container's nominal content region when placed with positioned layout. Defaults to 0.

The y property

Y coordinate of the container's anchor point relative to the top-left corner of the parent container's nominal content region when placed with positioned layout. Defaults to 0.

The corner element

A formatting element that describes the appearance of a vertex between two edges

```
<corner

Properties:
    id="xml-id"
    inverted="0 | 1"
    join="square | round"
    presence="visible | invisible | hidden"
    radius="0in | measurement"
    stroke="solid | dashed | dotted | lowered |
            raised | etched | embossed | dashDot |
            dashDotDot"
    thickness="0.5pt | measurement"
    use="cdata"
    usehref="cdata"
>
    <color>  [0..1]
    <extras>  [0..1]
</corner>
```

The corner element is used within the following other elements:
border proto rectangle

In addition to properties of the corner element, the handedness specification of the enclosing element also influences the appearance of the corner. In turn, the corner exerts some influence over the appearance of the edges it draws, particularly through its radius property.

The default color for a corner if black.

The color property

An element that describes a color.

For more information see "The color element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The inverted property

Specifies whether the corner appears convex (it joins the edges tangentially) or is inverted and appears concave (it joins the edges at right angles).

0

The corner appears convex.

1

The corner appears concave.

The join property

Specifies the shape of the corner.

square

The corner has the shape of a right-angle between the adjoining edges.

round

The corner has the shape of a round curve between the adjoining edges.

The presence property

Visibility control.

visible

Make it visible.

invisible

Make it transparent. Although invisible it still takes up space.

hidden

Hide it. It is not displayed and does not take up space.

The radius property

Specifies the radius of the corner.

This property always influences the appearance of round corners, but will also determine the depth of an inverted square corner.

Each edge is trimmed from its end points by the corner radius, irrespective of the values of the inverted and join attributes. In general, this is of no consequence, as the corner will visibly join with the edges at their trim points. However, if the corner specifies a presence of `invisible`, the trimming of the edges will become apparent, even when the corner is square and not inverted.

The stroke property

Specifies the appearance of the line.

solid

Solid.

dashed

A series of rectangular dashes.

dotted

A series of round dots.

dashDot

Alternating rectangular dashes and dots.

dashDotDot

A series of a single rectangular dash followed by two round dots.

lowered

The line appears to enclose a lowered region.

raised

The line appears to enclose a raised region.

etched

The line appears to be a groove lowered into the drawing surface.

embossed

The line appears to be a ridge raised out of the drawing surface.

The thickness property

Thickness or weight of the displayed line. Defaults to `0.5pt`.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The date element

A content element that describes a single unit of data content representing a date.

```
<date

Properties:
    id="xml-id"
    name="xml-id"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</date>
```

The date element is used within the following other elements:
desc exObject extras items proto value variables

XFA dates conform to a subset of ISO-8601, as specified in Canonical Format Reference. This element is intended to hold a date only to the resolution of a single day and any date information beyond that resolution will be truncated. For instance, a date element enclosing the value `20010326T0630`, meaning 6:30am on March 26th 2001, will truncate the time and hold the value of `20010326`, resulting in a value of March 26th 2001.

Content

This element may enclose date data which is a subset of [ISO-8601] as specified in Canonical Format Reference.

When no content is present, the content shall be interpreted as representing a null value, irrespective of the value of the associtated `nullType` property in the data description.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The dateTime element

A content element that describes a single unit of data content representing a date and time value.

```
<dateTime

Properties:
     id="xml-id"
     name="xml-id"
     use="cdata"
     usehref="cdata"
>
     ...pcdata...
</dateTime>
```

The dateTime element is used within the following other elements:
desc exObject extras items proto value variables

Content

This element may enclose date/time data which is a subset of [ISO-8601] as specified in Canonical Format Reference.

When no content is present, the content shall be interpreted as representing a null value, irrespective of the value of the associated `nullType` property in the data description.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The dateTimeEdit element

A user interface element describing a widget intended to aid in the selection of date and/or time.

```
<dateTimeEdit

Properties:
    hScrollPolicy="auto | on | off"
    id="xml-id"
    use="cdata"
    usehref="cdata"
>
    <border>  [0..1]
    <comb>  [0..1]
    <extras>  [0..1]
    <margin>  [0..1]
</dateTimeEdit>
```

The dateTimeEdit element is used within the following other elements:
proto ui

The border property

A box model element that describes the border surrounding an object.

For more information see "The border element".

The comb property

An element that causes a field to be presented with vertical lines between the character positions.

For more information see "The comb element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The hScrollPolicy property

Controls the scrollability of the field in the horizontal direction.

auto

If the field is single-line it scrolls horizontally. Multi-line fields do not scroll horizontally.

on

A horizontal scroll bar is always displayed whether or not the input overflows the boundaries of the field. The field is scrollable regardless of whether it is a single-line or multi-line field.

off

The user is not allowed to enter characters beyond what can physically fit in the field width. This applies to typing and pasting from the clipboard. However data which is merged into the field

from the Data DOM is not restricted. If the data exceeds the field size the user may not be able to view all of it.

The id property

A unique identifier that may be used to identify this element as a target.

The margin property

A box model element that specifies one or more insets for an object.

For more information see "The margin element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The decimal element

A content type element that describes a single unit of data content representing a number with a fixed number of digits after the decimal.

```
<decimal

Properties:
    fracDigits="2 | integer"
    id="xml-id"
    leadDigits="-1 | integer"
    name="xml-id"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</decimal>
```

The decimal element is used within the following other elements:
desc exObject extras items proto value variables

Note that the `decimal` element and the `float` element differ only in the user interface. They are interchangeable in every other way.

Content

This element may enclose decimal-data which is an optional leading minus sign (Unicode character U+002D), followed by a sequence of decimal digits (Unicode characters U+0030 - U+0039) separated by a single period (Unicode character U+002E) as a decimal indicator.

To maximize the potential for data interchange, the decimal point is defined as '.' (Unicode character U+002E). No thousands/grouping separator, or other formatting characters, are permitted in the data. The template may specify a picture clause to provide a presentation more suitable for human consumption.

When no content is present, the content shall be interpreted as representing a null value, irrespective of the value of the associtated `nullType` property in the data description.

The fracDigits property

The maximum number of digits (inclusively) following the decimal point to capture and store. The default value (`-1`) causes the XFA processing application to record the number of fractional digits supplied in the original value (but not as the value of `fracDigits`). For example, if the data associated with the `decimal` content type is "12.000", the unformatted data is printed or displayed as "12.000".

If `fracDigits="-1"` and a new value is supplied for the decimal element, the number of fractional digits is reset accordingly. If the decimal value is changed as a result of a calculation, whether the number of fractional digits is nullified or retained is implementation defined.

It is an error if the number of fractional digits in the decimal value exceeds the value of fracDigits.

The id property

A unique identifier that may be used to identify this element as a target.

The leadDigits property

The maximum number of digits (inclusively) preceding the decimal point to capture and store. The default value (`-1`) indicates the number of preceding digits is retained. For example, if the data associated with the `decimal` content type is "00012.0", the unformatted data is printed or displayed as "00012.0". It is an error if the number of preceding digits in the value exceeds the value of leadDigits.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The defaultUi element

An element for widgets whose depiction is delegated to the XFA application.

```
<defaultUi

Properties:
    id="xml-id"
    use="cdata"
    usehref="cdata"
>
    <extras>   [0..1]
</defaultUi>
```

The defaultUi element is used within the following other elements:
proto ui

When the depiction of the widget is defaulted this element is used. In this mode the appearance and interaction of the widget is determined by examining the content of the field. For example, if the content is a number then a numeric editing widget is used. This element can also supply additional hints to a custom GUI via its `extras` child.

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The desc element

An element to hold human-readable metadata.

```
<desc

Properties:
    id="xml-id"
    use="cdata"
    usehref="cdata"
>

Children:
    <boolean>  [0..n]
    <date>  [0..n]
    <dateTime>  [0..n]
    <decimal>  [0..n]
    <exData>  [0..n]
    <float>  [0..n]
    <image>  [0..n]
    <integer>  [0..n]
    <text>  [0..n]
    <time>  [0..n]
</desc>
```

The desc element is used within the following other elements:
area contentArea draw exclGroup field pageArea proto subform subformSet

The boolean child

A content element describing single unit of data content representing a Boolean logical value.

For more information see "The boolean element".

The date child

A content element that describes a single unit of data content representing a date.

For more information see "The date element".

The dateTime child

A content element that describes a single unit of data content representing a date and time value.

For more information see "The dateTime element".

The decimal child

A content type element that describes a single unit of data content representing a number with a fixed number of digits after the decimal.

For more information see "The decimal element".

The exData child

A content element that describes a single unit of data of a foreign datatype.

For more information see "The exData element".

The float child

A content element that describes a single unit of data content representing a floating point value.

For more information see "The float element".

The id property

A unique identifier that may be used to identify this element as a target.

The image child

A content element that describes a single image.

For more information see "The image element".

The integer child

A content element that describes a single unit of data content representing an integer value.

For more information see "The integer element".

The text child

A content element that describes a single unit of data content representing a plain textual value.

For more information see "The text element".

The time child

A content element that describes a single unit of data content representing a time value.

For more information see "The time element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The digestMethod element

An element to hold the name of an acceptable digest method for a signature.

```
<digestMethod

Properties:
    id="xml-id"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</digestMethod>
```

The digestMethod element is used within the following other elements:
digestMethods proto

Content

One of the following values:

SHA1

May be used with credentials containing RSA or DSA public/private keys.

SHA256

May be used only with credentials containing RSA public/private keys.

SHA512

May be used only with credentials containing RSA public/private keys.

RIPEMD160

May be used only with credentials containing RSA public/private keys.

The id property

A unique identifier that may be used to identify this element as a target.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The digestMethods element

An element to hold a list of names of acceptable digest methods for a signature.

```
<digestMethods

Properties:
    id="xml-id"
    type="optional | required"
    use="cdata"
    usehref="cdata"
>

Children:
    <digestMethod>  [0..n]
</digestMethods>
```

The digestMethods element is used within the following other elements:
filter proto

The digestMethod child

An element to hold the name of an acceptable digest method for a signature.

For more information see "The digestMethod element".

The id property

A unique identifier that may be used to identify this element as a target.

The type property

Specifies whether the values provided in the element should be treated as a restrictive or non-restrictive set.

optional

The values provided in the element are optional seed values from which the XFA processing application may choose. The XFA processing application may also supply its own value.

required

The values provided in the element are seed values from which the XFA processing application must choose.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The draw element

A container element that contains non-interactive data content.

```
<draw

Properties:
    anchorType="topLeft | topCenter | topRight | middleLeft |
                middleCenter | middleRight | bottomLeft | bottomCenter |
                bottomRight"
    colSpan="1 | integer"
    h="0in | measurement"
    id="xml-id"
    locale="cdata"
    maxH="0in | measurement"
    maxW="0in | measurement"
    minH="0in | measurement"
    minW="0in | measurement"
    name="xml-id"
    presence="visible | invisible | hidden"
    relevant="cdata"
    rotate="0 | angle"
    use="cdata"
    usehref="cdata"
    w="0in | measurement"
    x="0in | measurement"
    y="0in | measurement"
>
    <assist>  [0..1]
    <border>  [0..1]
    <caption>  [0..1]
    <desc>  [0..1]
    <extras>  [0..1]
    <font>  [0..1]
    <margin>  [0..1]
    <para>  [0..1]
    <traversal>  [0..1]
    <ui>  [0..1]
    <value>  [0..1]

Children:
    <setProperty>  [0..n]
</draw>
```

The draw element is used within the following other elements:
area pageArea proto subform

Note that although all draw elements have `minH`, `maxH`,`minW` and `maxH` properties, not all draws are growable. Draw elements that are not growable ignore these properties. Draw elements with the following content types cannot grow:

- image

- arc
- rectangle
- line

The anchorType property

Location of the container's anchor point when placed with positioned layout strategy.

`topLeft`

Top left corner of the nominal extent.

`topCenter`

Center of the top edge of the nominal extent.

`topRight`

Top right corner of the nominal extent.

`middleLeft`

Middle of the left edge of the nominal extent.

`middleCenter`

Middle of the nominal extent.

`middleRight`

Middle of the right edge of the nominal extent.

`bottomLeft`

Bottom left corner of the nominal extent.

`bottomCenter`

Center of the bottom edge of the nominal extent.

`bottomRight`

Bottom right corner of the nominal extent.

The assist property

An element that supplies additional information about a container for users of interactive applications.

For more information see "The assist element".

The border property

A box model element that describes the border surrounding an object.

For more information see "The border element".

The caption property

A box model element that describes a descriptive label associated with an object.

For more information see "The caption element".

The colSpan property

Number of columns spanned by this object, when used inside a subform with a layout type of `row`. Defaults to 1.

The desc property

An element to hold human-readable metadata.

For more information see "The desc element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The font property

A formatting element that describes a font.

For more information see "The font element".

The h property

Height for layout purposes. A measurement value for `h` overrides any growth range allowed by the `minH` and `maxH` attributes. The absolute omission of this attribute or a value specified as an empty string indicates that the `minH` and `maxH` must be respected.

This attribute has no default. Setting this attribute to "-1" is an error.

The id property

A unique identifier that may be used to identify this element as a target.

The locale property

A designator that determines the ambient locale and hence default direction of text flow within this element. The text layout engine may override this within portions or all of such text as per the rules in the *Unicode Annex 9* [UAX-9] reference.

The value of this property must be one of the following:

`ambient`

Causes the ambient locale of the XFA application to be used.

`localeName`

A valid locale name that conforms to the syntax: language[_country]. Examples of valid locales are `zh` for Chinese and `en_CA` for English specific for Canada. For a complete list of valid locale values, refer to the *IETF RFC 1766* [RFC1766] and *ISO 639* [ISO-639-1] */ISO 3166* [ISO-3166-1] specifications. Note that this is the same set of locale names used by the `xml:lang` attribute defined in [XML1.0].

When this property is absent or empty the default behavior is to inherit the parent object's locale. If the outermost subform does not specify a locale it uses the ambient locale from the operating system. If the operating system does not supply a locale it falls back onto `en_US`.

The margin property

A box model element that specifies one or more insets for an object.

For more information see "The margin element".

The maxH property

Maximum height for layout purposes. This attribute is relevant only if the enclosing container element is growable and has an `h` attribute whose value is null. If an `h` attribute is supplied, the container is not vertically growable and this attribute is ignored.

If this attribute is not supplied, there is no limit. This attribute has no default. If `h` is omitted, a value must be supplied for this attribute. Setting this attribute to "-1" is an error.

The maxW property

Maximum width for layout purposes.This attribute is relevant only if the enclosing container element is growable and has a `w` attribute whose value is null. If a `w` attribute is supplied, the container is not horizontally growable and this attribute is ignored.

This attribute has no default. If `w` is omitted, a value must be supplied for this attribute. Setting this attribute to "-1" is an error.

The minH property

Minimum height for layout purposes. This attribute is relevant only if the enclosing container element is growable and has an `h` attribute whose value is null. If an `h` attribute is supplied, the container is not vertically growable and this attribute is ignored.

This attribute has no default. If `h` is omitted, a value must be supplied for this attribute. Setting this attribute to "-1" is an error.

The minW property

Minimum width for layout purposes. This attribute is relevant only if the enclosing container element is growable and has a `w` attribute whose value is null. If a `w` attribute is supplied, the container is not horizontally growable and this attribute is ignored.

This attribute has no default. If `w` is omitted, a value must be supplied for this attribute. Setting this attribute to "-1" is an error.

The name property

An identifier that may be used to identify this element in script expressions.

The para property

A formatting element that specifies default paragraph and alignment properties to be applied to the content of an enclosing container.

For more information see "The para element".

The presence property

Visibility control.

visible

Make it visible.

invisible

Make it transparent. Although invisible it still takes up space.

hidden

Hide it. It is not displayed and does not take up space.

The relevant property

Specifies the views for which the enclosing object is relevant. The views themselves are specified in the config object.

Views are supplied as a space-separated list of viewnames: `relevant="[+|-]viewname [[+|-]viewname[...]]"`. A token of the form *viewname* or *+viewname* indicates the enclosing element should be included in that particular view. A token of the form `-viewname` indicates the element should be excluded from that particular view.

If a container is excluded, it is not considered in the data binding process.

See also Concealing Containers Depending on View and Config Specification.

The rotate property

Causes the object to be rotated about its anchor point by the specified angle.

The angle is measured in degrees counter-clockwise with respect to the default position. The value must be a positive or negative multiple of 90. The default is `0`. Positive values cause the object to rotate counter-clockwise, and negative values, clockwise.

Note that the direction of rotation is the same as for positive angles in PostScript, PDF, and PCL but opposite to that in SVG.

The setProperty child

An element that causes a property of the container to be copied from a value in the XFA Data DOM or from data returned by a web service.

For more information see "The setProperty element".

The traversal property

An element that links its container to other objects in sequence.

For more information see "The traversal element".

The ui property

A user-interface element that encloses the actual user interface widget element.

For more information see "The ui element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The value property

A content element that encloses a single unit of data content.

For more information see "The value element".

The w property

Width for layout purposes. A measurement value for `w` overrides any growth range allowed by the `minW` and `maxW` attributes. The absolute omission of this attribute or a value specified as an empty string indicates that the `minW` and `maxW` must be respected.

This attribute has no default. Setting this attribute to "-1" is an error.

The x property

X coordinate of the container's anchor point relative to the top-left corner of the parent container's nominal content region when placed with positioned layout. Defaults to 0.

The y property

Y coordinate of the container's anchor point relative to the top-left corner of the parent container's nominal content region when placed with positioned layout. Defaults to 0.

The edge element

A formatting element that describes an arc, line, or one side of a border or rectangle.

```
<edge

Properties:
    cap="square | butt | round"
    id="xml-id"
    presence="visible | invisible | hidden"
    stroke="solid | dashed | dotted | lowered |
            raised | etched | embossed | dashDot |
            dashDotDot"
    thickness="0.5pt | measurement"
    use="cdata"
    usehref="cdata"
>
    <color>  [0..1]
    <extras>  [0..1]
</edge>
```

The edge element is used within the following other elements:
arc border line proto rectangle

The properties here influence the appearance of the edge. In addition, the handedness of the enclosing element influences the edge's appearance.

When an edge is part of a border or rectangle, the sibling corner elements exert some influence over the appearance of edges. In particular, each edge is trimmed back from its endpoints by the corner radius in effect at that endpoint, irrespective of whether the corner is round or square, inverted or not, and visible or invisible.

The default edge color is black.

The cap property

Specifies the rendered termination of the stroke. Strokes that form an enclosed area do not have such termination. In particular, *all* rectangle and border edges, as well as all 360-degree arc edges are not considered to have any termination, and this property does not apply. Arcs with sweep angles less than 360 degrees and lines *do* have terminations at both endpoints.

square

The stroke shall be terminated by rendering the end of the edge squarely beyond the edge's endpoint a distance equal to one-half the edge's thickness.

butt

The stroke shall be terminated by rendering the end of the edge squarely across the endpoint.

round

The stroke shall be terminated by rendering the end of the edge with a semi-circle at the edge's endpoint, having a radius equal to one-half the edge's thickness.

The color property

An element that describes a color.

For more information see "The color element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The presence property

Visibility control.

`visible`

Make it visible.

`invisible`

Make it transparent. Although invisible it still takes up space.

`hidden`

Hide it. It is not displayed and does not take up space.

The stroke property

Specifies the appearance of the line.

`solid`

Solid.

`dashed`

A series of rectangular dashes.

`dotted`

A series of round dots.

`dashDot`

Alternating rectangular dashes and dots.

`dashDotDot`

A series of a single rectangular dash followed by two round dots.

`lowered`

The line appears to enclose a lowered region.

`raised`

The line appears to enclose a raised region.

etched

The line appears to be a groove lowered into the drawing surface.

embossed

The line appears to be a ridge raised out of the drawing surface.

The thickness property

Thickness or weight of the displayed line. Defaults to `0.5pt`.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The encoding element

An element holding the name of an acceptable recipe for signature encoding.

```
<encoding

Properties:
    id="xml-id"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</encoding>
```

The encoding element is used within the following other elements:
encodings proto

This element corresponds to the subFilters element in PDFL.

Content

A signature encoding recipe name. Certain names have been assigned by Adobe but other security handlers may define their own. The names assigned by Adobe are:

adbe.x509.rsa_sha1

adbe.pkcs7.detached

adbe.pkcs7.sha1

The id property

A unique identifier that may be used to identify this element as a target.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The encodings element

An element holding a list of names of acceptable recipes for signature encoding.

```
<encodings

Properties:
    id="xml-id"
    type="optional | required"
    use="cdata"
    usehref="cdata"
>

Children:
    <encoding>  [0..n]
</encodings>
```

The encodings element is used within the following other elements:
filter proto

The encoding child

An element holding the name of an acceptable recipe for signature encoding.

For more information see "The encoding element".

The id property

A unique identifier that may be used to identify this element as a target.

The type property

Specifies whether the values provided in the element should be treated as a restrictive or non-restrictive set.

optional

The values provided in the element are optional seed values from which the XFA processing application may choose. The XFA processing application may also supply its own value.

required

The values provided in the element are seed values from which the XFA processing application must choose.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The encrypt element

An element that controls encryption of barcode or submit data.

```
<encrypt

Properties:
    id="xml-id"
    use="cdata"
    usehref="cdata"
>
    <certificate>  [0..1]
</encrypt>
```

The encrypt element is used within the following other elements:
barcode proto submit

The presence of this element with its content causes the data to be encrypted before writing it to the barcode or submitting it to the host. If this element is absent or empty no encryption is performed.

For a barcode the data is packaged by first writing out a four-byte encryption header, followed by a randomly generated RC4 session key that has been encrypted according to the enclosed certificate, then finally the RC4 encryption of the original data under the previously mentioned random RC4 session key. The four-byte encryption header consists of a byte with the decimal value 130 (0x82 hex), a byte with the decimal value 1, then finally two bytes which are the two least significant bytes of the serial number of the enclosed certificate. These last two bytes serve as a hint to barcode decoders as to which public key certificate was used in the original encryption, and can thus aid in the selection of private keys for decrypting.

For submission the encrypted data is added to a PDF as an encrypted attachment.

The certificate property

An element that holds a suitable Base64 DER-encoded X.509v3 certificate.

For more information see "The certificate element".

The id property

A unique identifier that may be used to identify this element as a target.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The event element

An automation element that causes a script to be executed or data to be submitted whenever a particular event occurs.

```
<event

Properties:
    activity="click | initialize | enter | exit |
              mouseEnter | mouseExit | change | preSave |
              postSave | prePrint | postPrint | ready |
              docReady | docClose | mouseUp | mouseDown |
              full | preSubmit | preExecute | postExecute |
              preOpen | indexChange"
    id="xml-id"
    name="xml-id"
    ref="$ | cdata"
    use="cdata"
    usehref="cdata"
>
    <extras>  [0..1]

One-of properties:
    <execute>  [0..1]
    <script>  [0..1]
    <signData>  [0..1]
    <submit>  [0..1]
</event>
```

The event element is used within the following other elements:
exclGroup field proto subform

Any given object can only generate certain types of events. For example, a `subform` can generate `initialize`, `enter`, and `exit` events but it cannot generate events associated with key strokes and mouse gestures because it cannot accept input focus. It is the responsibility of the template creator to ensure that events are bound to appropriate objects.

The activity property

The name of the event. The accompanying `ref` property must specify an object that can generate the named event.

`change`

Occurs when the user changes the field value. This will occur with each key-stroke, when text is pasted, when a new choice is selected, when a check button is clicked, and so on.

`click`

Occurs when the user clicks in the field. Most systems define click as pressing and releasing the mouse button while not moving the pointer beyond a very small threshold.

docClose

Occurs at the very end of processing if and only if all validations succeeded. Success in this case is defined as generating nothing worse than a warning (no errors). Note that this event comes too late to modify the saved document; it is intended to be used for generating an exit status or completion message.

docReady

Occurs before the document is rendered but after data binding. It comes after the ready event associated with the Form DOM.

enter

For a field, occurs when the field gains keyboard focus. For a subform or exclusion group, occurs when some field within the subform or exclusion group gains keyboard focus, that is, keyboard focus moves from outside the object to inside it.

exit

For a field, occurs when the field loses keyboard focus. For a subform or exclusion group, occurs when all fields within the subform or exclusion group lose keyboard focus, that is, focus moves from inside the object to outside it.

full

Occurs when the user has entered the maximum allowed amount of content into the field.

indexChange

Occurs whenever the instance manager for a variable-occurrence object initially adds an instance or changes the instance number of an existing instance. The event is received only by the affected instances.

initialize

Occurs after data binding is complete. A separate event is generated for each instance of the field in the Form DOM.

mouseDown

Occurs when the user presses the mouse button in the field, but before the button is released.

mouseEnter

Occurs when the user drags the mouse pointer over the field without necessarily pressing the button.

mouseExit

Occurs when the user drags the mouse pointer out of the field without necessarily pressing the button.

mouseUp

Occurs when the user releases the mouse button in the field.

postExecute

Occurs when data is sent to a web service via WSDL, just after the reply to the request has been received and the received data is marshalled in a connectionData element underneath `$datasets`. A script triggered by this event has the chance to examine and process the received data. After execution of this event the received data is deleted.

postPrint

Occurs just after the rendered form has been sent to the printer, spooler, or output destination.

postSave

Occurs just after the form has been written out in PDF or XDP format. Does not occur when the Data DOM or some other subset of the form is exported to XDP.

preExecute

Occurs when a request is sent to a web service via WSDL, just after the data has been marshalled in a connectionData element underneath `$datasets` but before the request has been sent. A script triggered by this event has the chance to examine and alter the data before the request is sent. If the script is marked to be run only at the server, the data is sent to the server with an indication that it should run the associated script before performing the rest of the processing.

preOpen

This event applies only to drop-down choice lists, or more specifically choice lists for which `open="userControl"` or open="onEntry". It is intended to provided the form an opportunity to populate complex choice lists with additional of values from which the user can choose.

prePrint

Occurs just prior to rendering for print.

preSave

Occurs just before the form data is written out in PDF or XDP format. Does not occur when the Data DOM or some other subset of the form is exported to XDP. XSLT postprocessing, if enabled, occurs after this event.

preSubmit

Occurs when data is submitted to the host via the HTTP protocol, just after the data has been marshalled in a connectionData element underneath `$datasets` but before the data is submitted to the host. A script triggered by this event has the chance to examine and alter the data before it is submitted. If the script is marked to be run only at the server, the data is sent to the server with an indication that it should run the associated script before performing the rest of the processing.

ready

Occurs when the DOM has finished loading.

The execute property

An element that causes an event to invoke a WSDL-based web service.

For more information see "The execute element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The ref property

A SOM expression identifying the object which generates the event. Defaults to the object containing this element.

This syntax requires explanation. The `ref` property points to the source of the event, not the destination of the event. (It is a "come-from", not a "go-to".) The advantage of this is that a component can be dropped into a template and plug itself into the events it needs to monitor.

Depending upon the value of the accompanying `activity` property, the `ref` property may point to a subform, field, or exclusion group, to `$host`, or to a DOM such as `$layout`. See Events for a discussion about what type of event each object can generate.

The script property

An automation element that contains a script.

For more information see "The script element".

The signData property

An element controlling an XML digital signature.

For more information see "The signData element".

The submit property

An element that describes how to submit data to a host, using an HTTP POST operation.

For more information see "The submit element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The exclGroup element

A container element that describes a mutual exclusion relationship between a set of containers.

```
<exclGroup

Properties:
    access="open | protected | readOnly | nonInteractive"
    accessKey="cdata"
    anchorType="topLeft | topCenter | topRight | middleLeft |
                middleCenter | middleRight | bottomLeft | bottomCenter |
                bottomRight"
    colSpan="1 | integer"
    h="0in | measurement"
    id="xml-id"
    layout="position | lr-tb | rl-tb | tb |
            table | row"
    maxH="0in | measurement"
    maxW="0in | measurement"
    minH="0in | measurement"
    minW="0in | measurement"
    name="xml-id"
    presence="visible | invisible | hidden"
    relevant="cdata"
    use="cdata"
    usehref="cdata"
    w="0in | measurement"
    x="0in | measurement"
    y="0in | measurement"
>
    <assist>  [0..1]
    <bind>  [0..1]
    <border>  [0..1]
    <calculate>  [0..1]
    <caption>  [0..1]
    <desc>  [0..1]
    <extras>  [0..1]
    <margin>  [0..1]
    <para>  [0..1]
    <traversal>  [0..1]
    <validate>  [0..1]

Children:
    <connect>  [0..n]
    <event>  [0..n]
    <field>  [0..n]
    <setProperty>  [0..n]
</exclGroup>
```

The exclGroup element is used within the following other elements:
area pageArea proto subform

An exclGroup is used to cause a set of radio buttons or check boxes to be mutually exclusive. This means that when the user activates one member of the set the other members are automatically deactivated. For example, if the set consists of radio buttons, clicking one button causes the other buttons to be released.

Each member of the exclusion group has an "on" value and "off" value associated with it. When the member is activated it assumes the "on" value and when it is deactivated it assumes the "off" value. The "on" value for each member of a particular exclusion group must be unique.

Selecting one of the members of the exclusion group in the user interface causes each member's value to be set to its "on" or "off" value as appropriate. Similarly assigning a value to a member of the exclusion group, if the value assigned is the "on" value, causes the other members to be deactivated.

Alternatively a value may be assigned to the exclusion group itself. In this case each member is activated if and only if the value matches the "on" value for that member.

The access property

Controls user access to the contents.

`nonInteractive`

Allow the content to be loaded from the data document, but not updated interactively. The effect is to behave (for this container) as though rendering to paper regardless of whether or not the context is interactive. Calculations are performed at load time but the content is not subsequently recalculated even if values upon which it depended change. Neither can the content be modified by scripts or web service invocations.

`open`

Allow update without restriction. The interactive user may modify the container's content, and tab or otherwise navigate into it. The container will produce events.

`protected`

The processing application must prevent the user from making any direct changes to the container's content. Indirect changes (e.g., via calculations) may occur. The container will not participate in the tabbing sequence, though an application may allow the selection of text for clipboard copying. A protected container will not generate any events.

`readOnly`

The processing application must not allow the user to make direct changes to the container's content. Indirect changes (e.g., via calculations) may occur. The container shall participate in the tabbing sequence and must allow the user to view its content, possibly scrolling through that content if required. The user must be able to select the container's content for clipboard copying. The container shall also generate a subset of events (those not associated with the user making direct changes to the content).

The accessKey property

An accelerator key used by an interactive application to move the input focus to a particular field element. The value of this attribute is a single character. When the user synchronously presses the platform-specific modifier key and the single character, the XFA processing application sets the focus to this field. On Windows systems, the modifier key is the Alt key; and on Mac systems, it is the Option key.

For example: The form author sets the accessKey of a field to "f". If a Windows user holds down Alt key while pressing "f", the focus shifts to that field.

When designing forms that include accelerator keys, form designers should instruct the user on the availability of the accelerator keys.

The anchorType property

Location of the container's anchor point when placed with positioned layout strategy.

`topLeft`

Top left corner of the nominal extent.

`topCenter`

Center of the top edge of the nominal extent.

`topRight`

Top right corner of the nominal extent.

`middleLeft`

Middle of the left edge of the nominal extent.

`middleCenter`

Middle of the nominal extent.

`middleRight`

Middle of the right edge of the nominal extent.

`bottomLeft`

Bottom left corner of the nominal extent.

`bottomCenter`

Center of the bottom edge of the nominal extent.

`bottomRight`

Bottom right corner of the nominal extent.

The assist property

An element that supplies additional information about a container for users of interactive applications.

For more information see "The assist element".

The bind property

An element that controls the behavior during merge operations of its enclosing element.

For more information see "The bind element".

The border property

A box model element that describes the border surrounding an object.

For more information see "The border element".

The calculate property

An automation element that controls the calculation of its container's value.

For more information see "The calculate element".

The caption property

A box model element that describes a descriptive label associated with an object.

For more information see "The caption element".

The colSpan property

Number of columns spanned by this object, when used inside a subform with a layout type of `row`. Defaults to 1.

The connect child

An element that describes the relationship between its containing object and a connection to a web service, schema, or data description.

Connections are defined outside the template in a separate packet with its own schema. See the XFA Connection Set Specification for more information.

For more information see "The connect element".

The desc property

An element to hold human-readable metadata.

For more information see "The desc element".

The event child

An automation element that causes a script to be executed or data to be submitted whenever a particular event occurs.

For more information see "The event element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The field child

A container element that describes a single interactive container capable of capturing and presenting data content.

For more information see "The field element".

The h property

Height for layout purposes. A measurement value for `h` overrides any growth range allowed by the `minH` and `maxH` attributes. The absolute omission of this attribute or a value specified as an empty string indicates that the `minH` and `maxH` must be respected.

This attribute has no default. Setting this attribute to "-1" is an error.

The id property

A unique identifier that may be used to identify this element as a target.

The layout property

Layout strategy to be used within this element.

position

The content of the element is positioned according to the to the location information expressed on the content elements.

lr-tb

The content of the element is flowed in a direction proceeding from left to right and top to bottom.

rl-tb

The content of the element is flowed in a direction proceeding from right to left and top to bottom.

row

This is an inner element of a table, representing one or more rows. The objects contained in this element are cells of the table and their height and width attributes, if any, are ignored. The cells are laid out from right to left and each one is adjusted to the height of the row and the width of one or more contiguous columns.

table

This is the outer element of a table. Each of its child subforms or exclusion groups must have its `layout` property set to `row`. The rows of the table are laid out from top to bottom.

tb

The content of the element is flowed in a direction proceeding from top to bottom.

The margin property

A box model element that specifies one or more insets for an object.

For more information see "The margin element".

The maxH property

Maximum height for layout purposes. This attribute is relevant only if the enclosing container element is growable and has an `h` attribute whose value is null. If an `h` attribute is supplied, the container is not vertically growable and this attribute is ignored.

If this attribute is not supplied, there is no limit. This attribute has no default. If `h` is omitted, a value must be supplied for this attribute. Setting this attribute to "-1" is an error.

The maxW property

Maximum width for layout purposes.This attribute is relevant only if the enclosing container element is growable and has a `w` attribute whose value is null. If a `w` attribute is supplied, the container is not horizontally growable and this attribute is ignored.

This attribute has no default. If `w` is omitted, a value must be supplied for this attribute. Setting this attribute to "-1" is an error.

The minH property

Minimum height for layout purposes. This attribute is relevant only if the enclosing container element is growable and has an `h` attribute whose value is null. If an `h` attribute is supplied, the container is not vertically growable and this attribute is ignored.

This attribute has no default. If `h` is omitted, a value must be supplied for this attribute. Setting this attribute to "-1" is an error.

The minW property

Minimum width for layout purposes. This attribute is relevant only if the enclosing container element is growable and has a `w` attribute whose value is null. If a `w` attribute is supplied, the container is not horizontally growable and this attribute is ignored.

This attribute has no default. If `w` is omitted, a value must be supplied for this attribute. Setting this attribute to "-1" is an error.

The name property

An identifier that may be used to identify this element in script expressions.

The para property

A formatting element that specifies default paragraph and alignment properties to be applied to the content of an enclosing container.

For more information see "The para element".

The presence property

Visibility control.

`visible`

Make it visible.

`invisible`

Make it transparent. Although invisible it still takes up space.

`hidden`

Hide it. It is not displayed and does not take up space.

The relevant property

Specifies the views for which the enclosing object is relevant. The views themselves are specified in the config object.

Views are supplied as a space-separated list of viewnames: `relevant="[+|-]viewname [[+|-]viewname[...]]"`. A token of the form *viewname* or *+viewname* indicates the enclosing element should be included in that particular view. A token of the form *-viewname* indicates the element should be excluded from that particular view.

If a container is excluded, it is not considered in the data binding process.

See also Concealing Containers Depending on View and Config Specification.

The setProperty child

An element that causes a property of the container to be copied from a value in the XFA Data DOM or from data returned by a web service.

For more information see "The setProperty element".

The traversal property

An element that links its container to other objects in sequence.

For more information see "The traversal element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The validate property

A automation element that controls validation of user-supplied data.

For more information see "The validate element".

The w property

Width for layout purposes. A measurement value for `w` overrides any growth range allowed by the `minW` and `maxW` attributes. The absolute omission of this attribute or a value specified as an empty string indicates that the `minW` and `maxW` must be respected.

This attribute has no default. Setting this attribute to "-1" is an error.

The x property

X coordinate of the container's anchor point relative to the top-left corner of the parent container's nominal content region when placed with positioned layout. Defaults to 0.

The y property

Y coordinate of the container's anchor point relative to the top-left corner of the parent container's nominal content region when placed with positioned layout. Defaults to 0.

The exData element

A content element that describes a single unit of data of a foreign datatype.

```
<exData

Properties:
    contentType="text/plain | cdata"
    href="cdata"
    id="xml-id"
    maxLength="-1 | integer"
    name="xml-id"
    transferEncoding="none | base64"
    use="cdata"
    usehref="cdata"
>

One-of properties:
    ...pcdata...
    ...xhtml...
    ...xml...
</exData>
```

The exData element is used within the following other elements:
desc exObject extras items proto value variables

Content

This element may enclose foreign data which is PCDATA that represents the actual data content of the specified content type, encoded in the specified transfer encoding.

When no data content is provided, the data content may be interpreted as representing a null value. This behavior is dependent upon the context of where the data content is used. For instance, a field may interpret empty data content as null based upon the associated `nullType` property in the data description.

The contentType property

The type of content in the referenced document, expressed as a MIME type. For more information, please see [RFC2046].

The following values are allowed for documents containing text:

`text/plain`

Unadorned text. The XFA application may accept content that does not conform strictly to the requirements of the MIME type.

`pcdata`

Support for other text types, such as `text/html` is implementation-defined.

When the referenced document is an image, a suitable MIME-type must be supplied for this property to tell the application that the content is an image. However, the application is free to override the supplied

value if upon examining the image data it determines that the image data is of a different type. Which image types are supported is implementation-defined.

The href property

Specifies a reference to an external entity.

The set of supported URI schemes (e.g., `http:`, `ftp:`) is implementation-defined.

The id property

A unique identifier that may be used to identify this element as a target.

The maxLength property

Specifies the maximum (inclusive) allowable length of the content, or `-1` to signify that there is no maximum length imposed on the content. The default is `-1`.

The interpretation of this property is affected by the content type. This specification only defines the interpretation of this property for content types that represent some form of textual content. In this case this property specifies the maximum (inclusive) allowable length of the content in characters. For instance, where the content type is `text/plain` this property represents the maximum (inclusive) number of characters of plain text content. In kind, where the content type is "text/html" this property represents the maximum (inclusive) number of characters of content excluding markup, insignificant whitespace, etc.

The name property

An identifier that may be used to identify this element in script expressions.

The transferEncoding property

The encoding of binary content in the referenced document.

`none`

The referenced document is not encoded. If the referenced document is specified via a URI then it will be transferred as a byte stream. If the referenced document is inline it must conform to the restrictions on PCDATA.

`base64`

The binary content is encoded in accordance with the base64 transfer encoding s specified in [RFC2045].

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The execute element

An element that causes an event to invoke a WSDL-based web service.

```
<execute

Properties:
    connection="cdata"
    executeType="import | remerge"
    id="xml-id"
    runAt="client | server | both"
    use="cdata"
    usehref="cdata"
>
</execute>
```

The execute element is used within the following other elements:
event proto

Events can cause transactions to occur with web services. This element associates its parent event with a particular connection to a web service as defined in the connectionSet packet of the XDP. The connection definition supplies the particulars of the transaction such as the URIs to be used and the operation to request. The fields and exclusion groups which exchange data with the web service are nominated by their `connect` properties.

The event can be processed by the client, by the server, or both. When an event is processed by both, the client does its part of the processing first, then sends the resulting data to the server for completion.

The connection property

The `name` of the associated `connection` element in the connection set.

Connections are defined outside the template in a separate packet with its own schema. See Connection Set Grammar for more information.

This property identifies the connection to the server. It is an error if this property is not supplied or is empty.

If the value of the `runAt` property is `client` and the XFA processor is acting as server this property is ignored. Likewise if the property is `server` and the XFA processor is acting as client. Otherwise it is an error for there to not be any connection with the given name.

Not all connections point to a web service. Some point to a data description or a schema. The connection named by this property must point to a web service.

The executeType property

Specifies how the XFA processor should process imported data, assuming the following conditions exist: (1) the name of the `connection` attribute matches the name of the currently executing connection; and (2) the `usage` attribute has a value of `importOnly` or `exportAndImport`.

`import`

The XFA processor updates the values of containers that are already bound to the output of the connection.

remerge

The XFA processor clears the Form DOM and then rebuilds it, using a merge (data binding) operation. When merging data with dynamic forms, the XFA processor may dynamically create subforms to accommodate data.

The id property

A unique identifier that may be used to identify this element as a target.

The runAt property

Specifies whence the script is to be invoked. The value must be one of the following:

client

The service is invoked only by the client.

server

The service is invoked only by the server.

both

The service is invoked by both client and server.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The exObject element

An element that describes a single program or implementation-dependent foreign object.

```
<exObject

Properties:
    archive="cdata"
    classId="cdata"
    codeBase="cdata"
    codeType="cdata"
    id="xml-id"
    name="xml-id"
    use="cdata"
    usehref="cdata"
>
    <extras>  [0..1]

Children:
    <boolean>  [0..n]
    <date>  [0..n]
    <dateTime>  [0..n]
    <decimal>  [0..n]
    <exData>  [0..n]
    <exObject>  [0..n]
    <float>  [0..n]
    <image>  [0..n]
    <integer>  [0..n]
    <text>  [0..n]
    <time>  [0..n]
</exObject>
```

The exObject element is used within the following other elements:
area exObject proto subform ui

The archive property

A URI specifying the location of an archive file that may contain program code related to the exObject.

The boolean child

A content element describing single unit of data content representing a Boolean logical value.

For more information see "The boolean element".

The classId property

A URI specifying a name or location of the program code represented by the exObject.

The codeBase property

A URI specifying a location that may be used to assist the resolution of a relative classID.

The codeType property

This property specifies an identifier corresponding to a MIME type that identifies the program code represented by the object, such as "application/java". For more information, please see [RFC2046] and [MIMETYPES].

The date child

A content element that describes a single unit of data content representing a date.

For more information see "The date element".

The dateTime child

A content element that describes a single unit of data content representing a date and time value.

For more information see "The dateTime element".

The decimal child

A content type element that describes a single unit of data content representing a number with a fixed number of digits after the decimal.

For more information see "The decimal element".

The exData child

A content element that describes a single unit of data of a foreign datatype.

For more information see "The exData element".

The exObject child

An element that describes a single program or implementation-dependent foreign object.

For more information see "The exObject element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The float child

A content element that describes a single unit of data content representing a floating point value.

For more information see "The float element".

The id property

A unique identifier that may be used to identify this element as a target.

The image child

A content element that describes a single image.

For more information see "The image element".

The integer child

A content element that describes a single unit of data content representing an integer value.

For more information see "The integer element".

The name property

An identifier that may be used to identify this element in script expressions.

The text child

A content element that describes a single unit of data content representing a plain textual value.

For more information see "The text element".

The time child

A content element that describes a single unit of data content representing a time value.

For more information see "The time element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The extras element

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

```
<extras

Properties:
    id="xml-id"
    name="xml-id"
    use="cdata"
    usehref="cdata"
>

Children:
    <boolean>  [0..n]
    <date>  [0..n]
    <dateTime>  [0..n]
    <decimal>  [0..n]
    <exData>  [0..n]
    <extras>  [0..n]
    <float>  [0..n]
    <image>  [0..n]
    <integer>  [0..n]
    <text>  [0..n]
    <time>  [0..n]
</extras>
```

The extras element is used within the following other elements:
area barcode border break button calculate caption checkButton choiceList color contentArea corner dateTimeEdit defaultUi draw edge event exclGroup exObject extras field fill font format imageEdit keep linear manifest margin numericEdit occur pageArea pageSet passwordEdit pattern proto radial signature solid stipple subform subformSet template textEdit traversal traverse ui validate

The boolean child

A content element describing single unit of data content representing a Boolean logical value.

For more information see "The boolean element".

The date child

A content element that describes a single unit of data content representing a date.

For more information see "The date element".

The dateTime child

A content element that describes a single unit of data content representing a date and time value.

For more information see "The dateTime element".

The decimal child

A content type element that describes a single unit of data content representing a number with a fixed number of digits after the decimal.

For more information see "The decimal element".

The exData child

A content element that describes a single unit of data of a foreign datatype.

For more information see "The exData element".

The extras child

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The float child

A content element that describes a single unit of data content representing a floating point value.

For more information see "The float element".

The id property

A unique identifier that may be used to identify this element as a target.

The image child

A content element that describes a single image.

For more information see "The image element".

The integer child

A content element that describes a single unit of data content representing an integer value.

For more information see "The integer element".

The name property

An identifier that may be used to identify this element in script expressions.

The text child

A content element that describes a single unit of data content representing a plain textual value.

For more information see "The text element".

The time child

A content element that describes a single unit of data content representing a time value.

For more information see "The time element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The field element

A container element that describes a single interactive container capable of capturing and presenting data content.

```
<field

Properties:
    access="open | protected | readOnly | nonInteractive"
    accessKey="cdata"
    anchorType="topLeft | topCenter | topRight | middleLeft |
                middleCenter | middleRight | bottomLeft | bottomCenter |
                bottomRight"
    colSpan="1 | integer"
    h="0in | measurement"
    id="xml-id"
    locale="cdata"
    maxH="0in | measurement"
    maxW="0in | measurement"
    minH="0in | measurement"
    minW="0in | measurement"
    name="xml-id"
    presence="visible | invisible | hidden"
    relevant="cdata"
    rotate="0 | angle"
    use="cdata"
    usehref="cdata"
    w="0in | measurement"
    x="0in | measurement"
    y="0in | measurement"
>
    <assist>  [0..1]
    <bind>  [0..1]
    <border>  [0..1]
    <calculate>  [0..1]
    <caption>  [0..1]
    <desc>  [0..1]
    <extras>  [0..1]
    <font>  [0..1]
    <format>  [0..1]
    <margin>  [0..1]
    <para>  [0..1]
    <traversal>  [0..1]
    <ui>  [0..1]
    <validate>  [0..1]
    <value>  [0..1]

Children:
    <bindItems>  [0..n]
    <connect>  [0..n]
    <event>  [0..n]
    <items>  [0..2]
    <setProperty>  [0..n]
```

```
</field>
```

The field element is used within the following other elements:
area exclGroup pageArea proto subform

Note that although all field elements have `minH`, `maxH`,`minW` and `maxH` properties, not all fields are growable. Fields that are not growable ignore these properties. Field elements with the following content or user interface types cannot grow:

- image
- choice list

The access property

Controls user access to the contents.

`nonInteractive`

Allow the content to be loaded from the data document, but not updated interactively. The effect is to behave (for this container) as though rendering to paper regardless of whether or not the context is interactive. Calculations are performed at load time but the content is not subsequently recalculated even if values upon which it depended change. Neither can the content be modified by scripts or web service invocations.

`open`

Allow update without restriction. The interactive user may modify the container's content, and tab or otherwise navigate into it. The container will produce events.

`protected`

The processing application must prevent the user from making any direct changes to the container's content. Indirect changes (e.g., via calculations) may occur. The container will not participate in the tabbing sequence, though an application may allow the selection of text for clipboard copying. A protected container will not generate any events.

`readOnly`

The processing application must not allow the user to make direct changes to the container's content. Indirect changes (e.g., via calculations) may occur. The container shall participate in the tabbing sequence and must allow the user to view its content, possibly scrolling through that content if required. The user must be able to select the container's content for clipboard copying. The container shall also generate a subset of events (those not associated with the user making direct changes to the content).

The accessKey property

An accelerator key used by an interactive application to move the input focus to a particular field element. The value of this attribute is a single character. When the user synchronously presses the platform-specific modifier key and the single character, the XFA processing application sets the focus to this field. On Windows systems, the modifier key is the Alt key; and on Mac systems, it is the Option key.

For example: The form author sets the accessKey of a field to "f". If a Windows user holds down Alt key while pressing "f", the focus shifts to that field.

When designing forms that include accelerator keys, form designers should instruct the user on the availability of the accelerator keys.

The anchorType property

Location of the container's anchor point when placed with positioned layout strategy.

`topLeft`

Top left corner of the nominal extent.

`topCenter`

Center of the top edge of the nominal extent.

`topRight`

Top right corner of the nominal extent.

`middleLeft`

Middle of the left edge of the nominal extent.

`middleCenter`

Middle of the nominal extent.

`middleRight`

Middle of the right edge of the nominal extent.

`bottomLeft`

Bottom left corner of the nominal extent.

`bottomCenter`

Center of the bottom edge of the nominal extent.

`bottomRight`

Bottom right corner of the nominal extent.

The assist property

An element that supplies additional information about a container for users of interactive applications.

For more information see "The assist element".

The bind property

An element that controls the behavior during merge operations of its enclosing element.

For more information see "The bind element".

The bindItems child

An element that extracts data into an item list.

For more information see "The bindItems element".

The border property

A box model element that describes the border surrounding an object.

For more information see "The border element".

The calculate property

An automation element that controls the calculation of its container's value.

For more information see "The calculate element".

The caption property

A box model element that describes a descriptive label associated with an object.

For more information see "The caption element".

The colSpan property

Number of columns spanned by this object, when used inside a subform with a layout type of `row`. Defaults to 1.

The connect child

An element that describes the relationship between its containing object and a connection to a web service, schema, or data description.

Connections are defined outside the template in a separate packet with its own schema. See the XFA Connection Set Specification for more information.

For more information see "The connect element".

The desc property

An element to hold human-readable metadata.

For more information see "The desc element".

The event child

An automation element that causes a script to be executed or data to be submitted whenever a particular event occurs.

For more information see "The event element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The font property

A formatting element that describes a font.

For more information see "The font element".

The format property

A rendering element that encloses output formatting information such as the picture clause.

For more information see "The format element".

The h property

Height for layout purposes. A measurement value for `h` overrides any growth range allowed by the `minH` and `maxH` attributes. The absolute omission of this attribute or a value specified as an empty string indicates that the `minH` and `maxH` must be respected.

This attribute has no default. Setting this attribute to "-1" is an error.

The id property

A unique identifier that may be used to identify this element as a target.

The items child

An element that supplies a set of values for a choice list or a check button.

For more information see "The items element".

The locale property

A designator that influences the locale used to format the localizable content of this element. Such localizable content includes currency and time/date. Locale affects the representation of data formatted, validated, or parsed by picture clauses. Locale is also considered by certain FormCalc functions.

This designator also influences the default direction of text flow within this element. The text layout engine may override this within portions or all of such text as per the rules in the *Unicode Annex 9* [UAX-9] reference.

The value of this property must be one of the following:

`ambient`

Causes the ambient locale of the XFA application to be used.

`localeName`

A valid locale name that conforms to the syntax: language[_country]. Examples of valid locales are `zh` for Chinese and `en_CA` for English specific for Canada. For a complete list of valid locale values, refer to the *IETF RFC 1766* [RFC1766] and *ISO 639* [ISO-639-1] */ISO 3166* [ISO-3166-1] specifications. Note that this is the same set of locale names used by the `xml:lang` attribute defined in [XML1.0].

When this property is absent or empty the default behavior is to inherit the parent object's locale. If the outermost subform does not specify a locale it uses the ambient locale from the operating system. If the operating system does not supply a locale it falls back onto `en_US`.

The margin property

A box model element that specifies one or more insets for an object.

For more information see "The margin element".

The maxH property

Maximum height for layout purposes. This attribute is relevant only if the enclosing container element is growable and has an `h` attribute whose value is null. If an `h` attribute is supplied, the container is not vertically growable and this attribute is ignored.

If this attribute is not supplied, there is no limit. This attribute has no default. If `h` is omitted, a value must be supplied for this attribute. Setting this attribute to "-1" is an error.

The maxW property

Maximum width for layout purposes.This attribute is relevant only if the enclosing container element is growable and has a `w` attribute whose value is null. If a `w` attribute is supplied, the container is not horizontally growable and this attribute is ignored.

This attribute has no default. If `w` is omitted, a value must be supplied for this attribute. Setting this attribute to "-1" is an error.

The minH property

Minimum height for layout purposes. This attribute is relevant only if the enclosing container element is growable and has an `h` attribute whose value is null. If an `h` attribute is supplied, the container is not vertically growable and this attribute is ignored.

This attribute has no default. If `h` is omitted, a value must be supplied for this attribute. Setting this attribute to "-1" is an error.

The minW property

Minimum width for layout purposes. This attribute is relevant only if the enclosing container element is growable and has a `w` attribute whose value is null. If a `w` attribute is supplied, the container is not horizontally growable and this attribute is ignored.

This attribute has no default. If `w` is omitted, a value must be supplied for this attribute. Setting this attribute to "-1" is an error.

The name property

An identifier that may be used to identify this element in script expressions.

The para property

A formatting element that specifies default paragraph and alignment properties to be applied to the content of an enclosing container.

For more information see "The para element".

The presence property

Visibility control.

`visible`

Make it visible.

`invisible`

Make it transparent. Although invisible it still takes up space.

`hidden`

Hide it. It is not displayed and does not take up space.

The relevant property

Specifies the views for which the enclosing object is relevant. The views themselves are specified in the config object.

Views are supplied as a space-separated list of viewnames: `relevant="[+|-]viewname [[+|-]viewname[...]]"`. A token of the form *viewname* or +*viewname* indicates the enclosing element should be included in that particular view. A token of the form `-viewname` indicates the element should be excluded from that particular view.

If a container is excluded, it is not considered in the data binding process.

See also Concealing Containers Depending on View and Config Specification.

The rotate property

Causes the object to be rotated about its anchor point by the specified angle.

The angle is measured in degrees counter-clockwise with respect to the default position. The value must be a positive or negative multiple of 90. The default is `0`. Positive values cause the object to rotate counter-clockwise, and negative values, clockwise.

Note that the direction of rotation is the same as for positive angles in PostScript, PDF, and PCL but opposite to that in SVG.

The setProperty child

An element that causes a property of the container to be copied from a value in the XFA Data DOM or from data returned by a web service.

For more information see "The setProperty element".

The traversal property

An element that links its container to other objects in sequence.

For more information see "The traversal element".

The ui property

A user-interface element that encloses the actual user interface widget element.

For more information see "The ui element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The validate property

A automation element that controls validation of user-supplied data.

For more information see "The validate element".

The value property

A content element that encloses a single unit of data content.

For more information see "The value element".

The w property

Width for layout purposes. A measurement value for `w` overrides any growth range allowed by the `minW` and `maxW` attributes. The absolute omission of this attribute or a value specified as an empty string indicates that the `minW` and `maxW` must be respected.

This attribute has no default. Setting this attribute to "-1" is an error.

The x property

X coordinate of the container's anchor point relative to the top-left corner of the parent container's nominal content region when placed with positioned layout. Defaults to 0.

The y property

Y coordinate of the container's anchor point relative to the top-left corner of the parent container's nominal content region when placed with positioned layout. Defaults to 0.

The fill element

A formatting element that applies a color and optional rendered designs to the region enclosed by an object.

```
<fill

Properties:
    id="xml-id"
    presence="visible | invisible | hidden"
    use="cdata"
    usehref="cdata"
>
    <color>  [0..1]
    <extras>  [0..1]

One-of properties:
    <linear>  [0..1]
    <pattern>  [0..1]
    <radial>  [0..1]
    <solid>  [0..1]
    <stipple>  [0..1]
</fill>
```

The fill element is used within the following other elements:
arc border font proto rectangle

In the absence of a fill element the object is drawn without any fill, except for text which is drawn with a solid black fill.

The fill element has a child color element that specifies the *background*or *starting* color. If a fill element is provided but it has no child color element the color defaults to white.

The fill element also has a child fill type element (linear, pattern, radial, solid, stipple) that specifies the type of fill operation to perform. This uses the color established with the fill's color element, possibly along with its own color, to achieve the desired effect. If a fill element is provided but it has no child type element the type defaults to solid.

The color property

An element that describes a color.

For more information see "The color element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The linear property

A fill type element that describes a linear gradient fill.

For more information see "The linear element".

The pattern property

A fill type element that describes a hatching pattern.

For more information see "The pattern element".

The presence property

Visibility control.

This property is ignored when the `fill` element is the child of a `font` element. In such cases the effective value is always `visible`.

`visible`

Make it visible.

`invisible`

Make it transparent. Although invisible it still takes up space.

`hidden`

Hide it. It is not displayed and does not take up space.

The radial property

A fill type element that describes a radial gradient fill.

For more information see "The radial element".

The solid property

A fill type element that describes a solid fill.

For more information see "The solid element".

The stipple property

A fill type element that describes a stippling effect.

For more information see "The stipple element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The filter element

An element that contains the criteria for filtering signing certificates used to generate XML digital signatures.

```
<filter

Properties:
    addRevocationInfo="cdata"
    id="xml-id"
    name="xml-id"
    use="cdata"
    usehref="cdata"
    version="cdata"
>
    <certificates>  [0..1]
    <digestMethods>  [0..1]
    <encodings>  [0..1]
    <handler>  [0..1]
    <mdp>  [0..1]
    <reasons>  [0..1]
    <timeStamp>  [0..1]
</filter>
```

The filter element is used within the following other elements:
proto signature signData

The `mdp`, `reasons` and `timestamp` children are only meaningful if the parent of this element is a `signature` element. If they are grandchild of `signData` they are ignored.

The addRevocationInfo property

Controls whether certificate revocation information is included in the signature manifest.

Note that despite the appearance of the syntax summary above there is no mandated default value for this attribute. If the attribute is omitted or empty the XFA application may default to whatever value it considers appropriate.

`required`

Revocation information is required.

`optional`

Revocation information is optional.

`none`

Revocation information is prohibited.

The certificates property

An element that holds a collection of certificate filters used to identify the signer.

For more information see "The certificates element".

The digestMethods property

An element to hold a list of names of acceptable digest methods for a signature.

For more information see "The digestMethods element".

The encodings property

An element holding a list of names of acceptable recipes for signature encoding.

For more information see "The encodings element".

The handler property

An element controlling what signature handler is used for a data-signing operation for an XML digital signature.

For more information see "The handler element".

The id property

A unique identifier that may be used to identify this element as a target.

The mdp property

An element that controls an MDP+ signature.

For more information see "The mdp element".

The name property

An identifier that may be used to identify this element in script expressions.

The reasons property

An element containing a choice of reason strings for including with an XML Digital Signature.

For more information see "The reasons element".

The timeStamp property

An element that controls the time-stamping of a signature.

For more information see "The timeStamp element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The version property

Specifies compatibility with a particular revision of the PDF signature verification algorithm. See the PDF manual for the history of changes to this algorithm.

There is no default for this value. It must always be supplied.

1

Compatible with Acrobat 6 or later.

2

Compatible with Acrobat 8 or later.

3

Compatible with Acrobat 9 or later.

Additional values will be defined as needed.

The float element

A content element that describes a single unit of data content representing a floating point value.

```
<float

Properties:
     id="xml-id"
     name="xml-id"
     use="cdata"
     usehref="cdata"
>
     ...pcdata...
</float>
```

The float element is used within the following other elements:
desc exObject extras items proto value variables

Note that the `decimal` element and the `float` element differ only in the user interface. They are interchangeable in every other way.

Content

This element may enclose float-data which is an optional leading minus sign (Unicode character U+002D), followed by a sequence of decimal digits (Unicode characters U+0030 - U+0039) separated by a single period (Unicode character U+002E) as a decimal indicator.

To maximize the potential for data interchange, the decimal point is defined as '.' (Unicode character U+002E). No thousands/grouping separator, or other formatting characters, are permitted in the data. However, the template may employ a picture clause to generate a more suitable human-readable presentation of the value.

When no content is present, the content shall be interpreted as representing a null value, irrespective of the value of the associated `nullType` property in the data description.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The font element

A formatting element that describes a font.

```
<font

Properties:
    baselineShift="0in | measurement"
    id="xml-id"
    lineThrough="0 | 1 | 2"
    lineThroughPeriod="all | word"
    overline="0 | 1 | 2"
    overlinePeriod="all | word"
    posture="normal | italic"
    size="10pt | measurement"
    typeface="Courier Std | cdata"
    underline="0 | 1 | 2"
    underlinePeriod="all | word"
    use="cdata"
    usehref="cdata"
    weight="normal | bold"
>
    <extras>  [0..1]
    <fill>  [0..1]
</font>
```

The font element is used within the following other elements:
caption draw field proto

The baselineShift property

Specifies a positive or negative measurement value to express that the font should shift up from the baseline (a positive measurement) or shift down from the baseline (a negative measurement).The default is 0.

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The fill property

A formatting element that applies a color and optional rendered designs to the region enclosed by an object.

For more information see "The fill element".

The id property

A unique identifier that may be used to identify this element as a target.

The lineThrough property

This property specifies the activation of a single or double line extending through the text (also known as strikethrough).

0

The font shall be rendered without a line through the text.

1

The font shall be rendered with a single line through the text.

2

The font shall be rendered with a double line through the text.

The lineThroughPeriod property

This property controls the appearance of the line extending through the text (also known as strikethrough).

all

The rendered line shall extend across word breaks.

word

The rendered line shall be interrupted at word breaks.

The overline property

This property specifies the activation and type of overlining.

0

The font shall be rendered without overlining.

1

The font shall be rendered with a single overline.

2

The font shall be rendered with a double overline.

The overlinePeriod property

This property controls the appearance of overlining.

all

The rendered line shall extend across word breaks.

word

The rendered line shall be interrupted at word breaks.

The posture property

This property specifies the posture of the font. (Currently, the set of choices has been kept small. It is likely that the list will grow in future versions of this specification.)

normal

The font shall have a normal posture.

italic

The font shall be italicized.

The size property

Specifies the height of the font as a measurement value. The default is `10pt`.

The typeface property

This property specifies the name of the typeface. The default is `Courier`.

The underline property

This property specifies the activation and type of underlining.

0

The font shall be rendered without underlining.

1

The font shall be rendered with a single underline.

2

The font shall be rendered with a double underline.

The underlinePeriod property

This property controls the appearance of underlining.

all

The rendered line shall extend across word breaks.

word

The rendered line shall be interrupted at word breaks.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The weight property

This property controls the appearance of typeface weight.

`bold`

The typeface will be rendered with a bold weight.

`normal`

The typeface will be rendered at its default weight.

The format element

A rendering element that encloses output formatting information such as the picture clause.

```
<format

Properties:
    id="xml-id"
    use="cdata"
    usehref="cdata"
>
    <extras>   [0..1]
    <picture>   [0..1]
</format>
```

The format element is used within the following other elements:
field proto

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The picture property

A rendering element that describes input parsing and output formatting information.

For more information see "The picture element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The handler element

An element controlling what signature handler is used for a data-signing operation for an XML digital signature.

```
<handler

Properties:
    id="xml-id"
    type="optional | required"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</handler>
```

The handler element is used within the following other elements:
filter proto

A signature handler is software that creates or authenticates a signature. The signature creation and authentication may be purely mathematical, such as a public/private-key encrypted document digest, or it may be a biometric form of identification, such as a handwritten signature, fingerprint, or retinal scan. The signature handler may be a plug-in or server provided by an agency that specializes in signature authentication.

Content

The name of the preferred signature handler to use when validating this signature. Example signature handlers are `Adobe.PPKLite`,`Entrust.PPKEF`, `CICI.SignIt`, and `VeriSign.PPKVS`.

The id property

A unique identifier that may be used to identify this element as a target.

The type property

Indicates whether the XFA processing application is required to use the specified handler filter.

`optional`

The handler is optional. That is, the XFA processing application may chose to use another algorithm rather than the one specified.

`required`

The handler is required. If it is not available, XFA processing application does not produce the requested signature. It is suggested the application also provide an error response to inform the person filling out the form that the signature has not been produced.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The image element

A content element that describes a single image.

```
<image

Properties:
    aspect="fit | none | actual | width |
            height"
    contentType="cdata"
    href="cdata"
    id="xml-id"
    name="xml-id"
    transferEncoding="base64 | none"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</image>
```

The image element is used within the following other elements:
desc exObject extras items proto value variables

Content

This element may enclose image-data which is PCDATA representing the actual image data content of the image, encoded in base64 encoding (see [RFC2045] for more information). If the image element also specifies external image content via the href property, the external content shall take priority.

When no data content is provided, the data content may be interpreted as representing a null value. This behavior is dependent upon the context of where the data content is used. For instance, a field may interpret empty data content as null based upon the associated `nullType` property in the data description.

The aspect property

This property specifies how the image is to map to the nominal content region of the image's container.

`fit`

The processing application must scale the image proportionally to the maximum possible size such that it fits within the nominal content region of the container.

`none`

The image shall be scaled such that it occupies the entire nominal content region of the container. This may result in different scale values being applied to the image's X and Y coordinates.

`actual`

The image shall be rendered using the dimensions stored in the image content. The extent of the container's nominal content region plays no role in the sizing of the image.

width

The image shall be scaled proportionally such that its width maps to the width of the container's nominal content region. The rendered image may not occupy the entire height of the nominal content region, or it may overflow the height.

height

The image shall be scaled proportionally such that its height maps to the height of the container's nominal content region. The rendered image may not occupy the entire width of the nominal content region, or it may overflow the width.

The contentType property

The MIME-type of content in the referenced document. Please see [RFC2046] for more information.

A suitable value must be supplied for this property. However, the application is free to override the supplied value if upon examining the image data it determines that the image data is of a different type.

The href property

Specifies a reference to an external image.

The transferEncoding property does not apply to external images.

The set of supported URI schemes (e.g., `http:`, `ftp:`) is implementation-defined.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The transferEncoding property

The encoding of binary content in the referenced document.

none

The referenced document is not encoded. If the referenced document is specified via a URI then it will be transferred as a byte stream. If the referenced document is inline it must conform to the restrictions on PCDATA.

base64

The binary content is encoded in accordance with the base64 transfer encoding s specified in [RFC2045].

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The imageEdit element

A user interface element that encloses a widget intended to aid in the manipulation of image content.

```
<imageEdit

Properties:
    data="link | embed"
    id="xml-id"
    use="cdata"
    usehref="cdata"
>
    <border>   [0..1]
    <extras>   [0..1]
    <margin>   [0..1]
</imageEdit>
```

The imageEdit element is used within the following other elements:
proto ui

The border property

A box model element that describes the border surrounding an object.

For more information see "The border element".

The data property

Indicates whether the image provided to the widget should be represented as a reference or should be embedded. This attribute affects the widget behavior during form fill-in.

`link`

The image is represented as a URI reference. If the user provides the widget with a URI, the `href` attribute of the container's `image` object is updated to reflect the new URI; and if the image object was previously loaded with an embedded image, that image is removed from the object.

`embed`

The image is embedded in the container's `image` object. If the user provides the widget with a URI, the image referenced by the URI is embedded as the content of the `image` object.

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The margin property

A box model element that specifies one or more insets for an object.

For more information see "The margin element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The integer element

A content element that describes a single unit of data content representing an integer value.

```
<integer

Properties:
    id="xml-id"
    name="xml-id"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</integer>
```

The integer element is used within the following other elements:
desc exObject extras items proto value variables

Content

This element may enclose integer-data which is an optional leading minus sign (Unicode character U+002D), followed by a sequence of decimal digits (Unicode characters U+0030 - U+0039).

When no content is present, the content shall be interpreted as representing a null value, irrespective of the value of the associtated `nullType` property in the data description.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The issuers element

A collection of issuer certificates that are acceptable for data signing an XML digital signature.

```
<issuers

Properties:
    id="xml-id"
    type="optional | required"
    use="cdata"
    usehref="cdata"
>

Children:
    <certificate>  [0..n]
</issuers>
```

The issuers element is used within the following other elements:
certificates proto

If the certificate being used to sign the manifest can be authenticated by any of the issuers (either directly or indirectly), that certificate is considered acceptable for signing. The issuer certificates specified by this element may be used in conjunction with the object identifiers specified in the oids element. X.509v3 certificates are described in RFC 3280, Internet X.509 Public Key Infrastructure, Certificate and Certificate Revocation List (CRL) Profile [RFC3280].

The certificate child

An element that holds a suitable Base64 DER-encoded X.509v3 certificate.

For more information see "The certificate element".

The id property

A unique identifier that may be used to identify this element as a target.

The type property

Specifies whether the values provided in the element should be treated as a restrictive or non-restrictive set.

optional

The values provided in the element are optional seed values from which the XFA processing application may choose. The XFA processing application may also supply its own value.

required

The values provided in the element are seed values from which the XFA processing application must choose.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The items element

An element that supplies a set of values for a choice list or a check button.

```
<items

Properties:
    id="xml-id"
    name="xml-id"
    presence="visible | invisible | hidden"
    ref="cdata"
    save="0 | 1"
    use="cdata"
    usehref="cdata"
>

Children:
    <boolean>  [0..n]
    <date>  [0..n]
    <dateTime>  [0..n]
    <decimal>  [0..n]
    <exData>  [0..n]
    <float>  [0..n]
    <image>  [0..n]
    <integer>  [0..n]
    <text>  [0..n]
    <time>  [0..n]
</items>
```

The items element is used within the following other elements:
field proto

This element has two different meanings depending upon whether is interpreted by a choice list user interface or a check box / radio button user interface.

The choice list user interface, its containing field element and the set of items elements all participate together to provide a set of choices and control the value that gets stored in the field.

The choice list presents the user with a set of choices. The object displayed for each choice (for example a text string) is generated from one content element which is a child of an items element. If there is only one items element that is a child of the field then the displayed object is copied into the field when the end-user selects that object.

However there can be two items element within a choice list field. If there are two items elements one contains the set of objects to be displayed and the other contains the corresponding set of values to be saved into the field. The items element containing the set of values to be saved must be flagged as such.

The checkButton user interface, its containing field element and the set of items elements all participate together to provide a single radio button or check box and control the value that gets stored in the field. (Mutually exclusive sets of check boxes or radio buttons are created by grouping these fields inside an exclGroup element.)

Usually a check box is presented as a rectangle that contains a check mark when it is selected and is empty when deselected. However a check box can have three states, often represented by a check mark, a cross, and emptiness. A radio button can only have two states, which are often presented as a circle that is filled (or "illuminated") when the button is selected and empty when deselected.

A field with a checkBox user interface can have at most one items child. The items list can have at most three values. The first value in the list is the "on" value, that is the value taken when the button or box is selected. If there is a second value, it is the "off" value, that is the value taken when the button or box is deselected. If there is a third value, it is the "neutral" value, that is the value taken when the check box is empty. If a third value is provided for a radio button it is ignored. When the second or third value is not provided it defaults to the null string.

The boolean child

A content element describing single unit of data content representing a Boolean logical value.

For more information see "The boolean element".

The date child

A content element that describes a single unit of data content representing a date.

For more information see "The date element".

The dateTime child

A content element that describes a single unit of data content representing a date and time value.

For more information see "The dateTime element".

The decimal child

A content type element that describes a single unit of data content representing a number with a fixed number of digits after the decimal.

For more information see "The decimal element".

The exData child

A content element that describes a single unit of data of a foreign datatype.

For more information see "The exData element".

The float child

A content element that describes a single unit of data content representing a floating point value.

For more information see "The float element".

The id property

A unique identifier that may be used to identify this element as a target.

The image child

A content element that describes a single image.

For more information see "The image element".

The integer child

A content element that describes a single unit of data content representing an integer value.

For more information see "The integer element".

The name property

An identifier that may be used to identify this element in script expressions.

The presence property

Visibility control.

visible

Make it visible.

invisible

Make it transparent. Although invisible it still takes up space.

hidden

Hide it. It is not displayed and does not take up space.

The ref property

A SOM expression pointing to a DOM node. All of the child nodes of the referenced node are incorporated into the list of items, regardless of the name of the child node.

This is commonly used to present a list held in the dataset.

The save property

This property determines whether this particular column contains values that may be entered into the corresponding field.

0

The values supplied by this element are for display only.

1

The values supplied by this element may be entered into the field.

At least one column must have `save` set to `1`. If more than one column have this property set, the value in the first column with it set is saved.

The text child

A content element that describes a single unit of data content representing a plain textual value.

For more information see "The text element".

The time child

A content element that describes a single unit of data content representing a time value.

For more information see "The time element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The keep element

An element that describes the constraints on keeping subforms together within a page or content area.

```
<keep

Properties:
    id="xml-id"
    intact="none | contentArea | pageArea"
    next="none | contentArea | pageArea"
    previous="none | contentArea | pageArea"
    use="cdata"
    usehref="cdata"
>
    <extras>  [0..1]
</keep>
```

The keep element is used within the following other elements:
proto subform

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The intact property

This property specifies the constraints on keeping a subform intact within a content area or page.

Note that the default value for this property is not fully depicted in the syntax summary above. There is no single default, instead it is context-sensitive. When the parent container's layout is "tb", "lr-tb", or "table" the default value is "none". When the parent container's layout is "position" or "row" the default value is "contentArea". Note that the default is (re-)computed at the moment the API call to get the value is made or at the moment the layout operation is invoked.

`none`

The determination of whether a subform will be rendered intact within a content area or page will be delegated to the processing application. It is possible that the subform will be split across a content area or page. This is the default when the parent container's layout is "tb", "lr-tb", or "table".

`contentArea`

The subform is requested to be rendered intact within a content area. This is the default when the parent container's layout is "position" or "row".

`pageArea`

The subform is requested to be rendered intact within a page.

The next property

This property specifies the constraints on keeping a subform together with the next subform within a content area or page.

none

The determination of whether a subform will be rendered in the same content area or page together with the next subform will be delegated to the processing application. No special keep constraints will be forced.

contentArea

The subform is requested to be rendered in the same content area with the next subform.

pageArea

The subform is requested to be rendered in the same page with the next subform.

The previous property

This property specifies the constraints on keeping a subform together with the previous subform within a content area or page.

none

The determination of whether a subform will be rendered in the same content area or page together with the previous subform will be delegated to the processing application. No special keep constraints will be forced.

contentArea

The subform is requested to be rendered in the same content area with the previous subform.

pageArea

The subform is requested to be rendered in the same page with the previous subform.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The keyUsage element

An element that specifies the key usage settings required in the signing certificate.

```
<keyUsage

Properties:
    crlSign="cdata"
    dataEncipherment="cdata"
    decipherOnly="cdata"
    digitalSignature="cdata"
    encipherOnly="cdata"
    id="xml-id"
    keyAgreement="cdata"
    keyCertSign="cdata"
    keyEncipherment="cdata"
    nonRepudiation="cdata"
    type="optional | required"
    use="cdata"
    usehref="cdata"
>
</keyUsage>
```

The keyUsage element is used within the following other elements:
certificates proto

The crlSign property

Controls whether and how the value of the same name in the certificate makes it acceptable or unacceptable.

yes

The value must be set in the certificate for it to be acceptable.

no

The value must **not** be set in the certificate for it to be acceptable.

There is no default for this attribute. If it is omitted or empty the corresponding attribute of the certificate is disregarded.

The dataEncipherment property

Controls whether and how the value of the same name in the certificate makes it acceptable or unacceptable.

yes

The value must be set in the certificate for it to be acceptable.

no

The value must **not** be set in the certificate for it to be acceptable.

There is no default for this attribute. If it is omitted or empty the corresponding attribute of the certificate is disregarded.

The decipherOnly property

Controls whether and how the value of the same name in the certificate makes it acceptable or unacceptable.

yes

The value must be set in the certificate for it to be acceptable.

no

The value must **not** be set in the certificate for it to be acceptable.

There is no default for this attribute. If it is omitted or empty the corresponding attribute of the certificate is disregarded.

The digitalSignature property

Controls whether and how the value of the same name in the certificate makes it acceptable or unacceptable.

yes

The value must be set in the certificate for it to be acceptable.

no

The value must **not** be set in the certificate for it to be acceptable.

There is no default for this attribute. If it is omitted or empty the corresponding attribute of the certificate is disregarded.

The encipherOnly property

Controls whether and how the value of the same name in the certificate makes it acceptable or unacceptable.

yes

The value must be set in the certificate for it to be acceptable.

no

The value must **not** be set in the certificate for it to be acceptable.

There is no default for this attribute. If it is omitted or empty the corresponding attribute of the certificate is disregarded.

The id property

A unique identifier that may be used to identify this element as a target.

The keyAgreement property

Controls whether and how the value of the same name in the certificate makes it acceptable or unacceptable.

yes

The value must be set in the certificate for it to be acceptable.

no

The value must **not** be set in the certificate for it to be acceptable.

There is no default for this attribute. If it is omitted or empty the corresponding attribute of the certificate is disregarded.

The keyCertSign property

Controls whether and how the value of the same name in the certificate makes it acceptable or unacceptable.

yes

The value must be set in the certificate for it to be acceptable.

no

The value must **not** be set in the certificate for it to be acceptable.

There is no default for this attribute. If it is omitted or empty the corresponding attribute of the certificate is disregarded.

The keyEncipherment property

Controls whether and how the value of the same name in the certificate makes it acceptable or unacceptable.

yes

The value must be set in the certificate for it to be acceptable.

no

The value must **not** be set in the certificate for it to be acceptable.

There is no default for this attribute. If it is omitted or empty the corresponding attribute of the certificate is disregarded.

The nonRepudiation property

Controls whether and how the value of the same name in the certificate makes it acceptable or unacceptable.

yes

The value must be set in the certificate for it to be acceptable.

no

The value must **not** be set in the certificate for it to be acceptable.

There is no default for this attribute. If it is omitted or empty the corresponding attribute of the certificate is disregarded.

The type property

Specifies whether the values provided in the element should be treated as a restrictive or non-restrictive set.

optional

The values provided in the element are optional seed values from which the XFA processing application may choose. The XFA processing application may also supply its own value. The application typically allows a person filling out the form to choose from the values provided or to specify his own value.

required

The values provided in the element are seed values from which the XFA processing application must choose. The application typically allows a person filling out the form to choose from only those values provided in the element.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The line element

A content element that describes a single rendered line.

```
<line

Properties:
    hand="even | left | right"
    id="xml-id"
    slope="\ | /"
    use="cdata"
    usehref="cdata"
>
    <edge>  [0..1]
</line>
```

The line element is used within the following other elements:
proto value

The edge property

A formatting element that describes an arc, line, or one side of a border or rectangle.

For more information see "The edge element".

The hand property

Description of the handedness of a line or edge.

even

Center the displayed line on the underlying vector or arc.

left

Position the displayed line immediately to the left of the underlying vector or arc, when following that line from its start point to its end point.

right

Position the displayed line immediately to the right of the underlying vector or arc, when following that line from its start point to its end point.

The id property

A unique identifier that may be used to identify this element as a target.

The slope property

This property specifies the orientation of the line.

\

The line extends from the top-left to the bottom-right.

/

The line extends from the bottom-left to the top-right.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The linear element

A fill type element that describes a linear gradient fill.

```
<linear

Properties:
    id="xml-id"
    type="toRight | toBottom | toLeft | toTop"
    use="cdata"
    usehref="cdata"
>
    <color>  [0..1]
    <extras>  [0..1]
</linear>
```

The linear element is used within the following other elements:
fill proto

A linear gradient fill appears as the *start color* at one "side" of the object and the *end color* at the opposite side. Between those two sides, the color gradually changes from start color to end color.

The color element enclosed by the linear element determines the end color. The color element enclosed by the parent fill element determines the start color.

The color property

An element that describes a color.

For more information see "The color element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The type property

Specifies the direction of the color transition.

`toRight`

The start color appears at the left side of the object and transitions into the end color at the right side.

`toLeft`

The start color appears at the right side of the object and transitions into the end color at the left side.

toTop

The start color appears at the bottom side of the object and transitions into the end color at the top side.

toBottom

The start color appears at the top side of the object and transitions into the end color at the bottom side.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The manifest element

An element that contains a list of references to all the nodes that are included in a node collection.

```
<manifest

Properties:
    action="include | exclude | all"
    id="xml-id"
    name="xml-id"
    use="cdata"
    usehref="cdata"
>
    <extras>  [0..1]

Children:
    <ref>  [0..n]
</manifest>
```

The manifest element is used within the following other elements:
proto signature signData variables

Node collections are commonly used for signatures, however they may also be employed in scripts.

The references may include non-unique SOM expressions, such as foo[*]. All nodes referenced by the expression are included in the list.

The references may overlap or duplicate node references. Multiply-referenced nodes are included in multiple places in the list.

When this element is the child of a `signature` element (hence being used to generate a PDF signature) only nodes which are fields are processed. Other nodes in the node list are ignored. PDF signatures automatically include all template and other nodes necessary to establish a document of record for the indicated fields.

The action property

Controls the definition of the node set.

`include`

The node set consists of those nodes and only those nodes listed.

`exclude`

The node set consists of all candidate nodes except those nodes listed. This value may only be used when this element is the child of a `signature` element.

`all`

The node set consists of all candidate nodes. No `ref` children are needed or expected. This value may only be used when this element is the child of a `signature` element.

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The ref child

An element holding an XFA-SOM expression that identifies a node to be included in an XML digital signature.

For more information see "The ref element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The margin element

A box model element that specifies one or more insets for an object.

```
<margin

Properties:
    bottomInset="0in | measurement"
    id="xml-id"
    leftInset="0in | measurement"
    rightInset="0in | measurement"
    topInset="0in | measurement"
    use="cdata"
    usehref="cdata"
>
    <extras>  [0..1]
</margin>
```

The margin element is used within the following other elements:
border caption checkButton choiceList dateTimeEdit draw exclGroup field imageEdit numericEdit passwordEdit proto signature subform textEdit

The bottomInset property

A measurement specifying the size of the bottom inset. The default is `0`.

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The leftInset property

A measurement specifying the size of the left inset. The default is `0`.

The rightInset property

A measurement specifying the size of the right inset. The default is `0`.

The topInset property

A measurement specifying the size of the top inset. The default is `0`.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The mdp element

An element that controls an MDP+ signature.

```
<mdp

Properties:
    id="xml-id"
    permissions="2 | 1 | 3"
    signatureType="filler | author"
    use="cdata"
    usehref="cdata"
>
</mdp>
```

The mdp element is used within the following other elements:
filter proto

This element is only meaningful when it is the grandchild of a `signature` element. Otherwise it is ignored.

The id property

A unique identifier that may be used to identify this element as a target.

The permissions property

An author signature attests to the validity of the whole form or parts of it. This attribute governs what operations may be performed on the certified form without invalidating the signature. Downstream XFA applications may enforce these permissions at run time but they don't have to be trusted to do so, because the signature is a hash of all attested parts of the form and hence is invalidated by any change to them. This attribute is ignored for filler signatures, which only attest to the data.

1

No changes to the document are permitted; any change to the document invalidates the signature.

2

Permitted changes are filling in forms, instantiating page templates, and signing; other changes invalidate the signature.

3

Permitted changes are those allowed by 2, as well as annotation creation, deletion, and modification; other changes invalidate the signature.

The signatureType property

The role of the person or program that has signed or will sign the form.

filler

A person or program that supplies data to an existing form.

author

A person or program that makes up a new form. Documents with this kind of signature are often referred to as *certified*.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The medium element

An element that describes a physical medium upon which to render. Some hybrid paper/glass media, such as PDF, may require both paper and glass properties.

```
<medium

Properties:
    id="xml-id"
    imagingBBox="none | cdata"
    long="0in | measurement"
    orientation="portrait | landscape"
    short="0in | measurement"
    stock="cdata"
    trayIn="auto | delegate | pageFront"
    trayOut="auto | delegate"
    use="cdata"
    usehref="cdata"
>
</medium>
```

The medium element is used within the following other elements:
pageArea proto

The id property

A unique identifier that may be used to identify this element as a target.

The imagingBBox property

Region within the paper that is available for rendering with four comma separated measurements representing the measurements for x, y, width, and height.

none

The entire area of the paper is available for rendering.

x*, *y*, *width*,*height

The rendering area is limited to a rectangle of the given `width` and `height`, at a distance of `x` from the left edge and `y` from the top edge. Note that the comma separators are required.

The long property

A measurement specifying the length of the long edge of the medium. The default is `0`.The length specified by `long` must be greater than the length specified by `short` .

The orientation property

The orientation of the medium as follows:

portrait

The orientation of the medium places the short edge at the top.

landscape

The orientation of the medium places the long edge at the top.

The short property

A measurement specifying the length of the short edge of the medium. The default is `0`. The length specified by `short` must be smaller than the length specified by `long`.

The stock property

The name of a standard paper size. The default is `letter`.

This name is the key used to find the appropriate section in the XDC file.

The trayIn property

Reserved for future use.

The trayOut property

Reserved for future use.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The message element

A automation element that holds one or more sub-elements containing messages used with validations and calculations.

```
<message

Properties:
    id="xml-id"
    use="cdata"
    usehref="cdata"
>

Children:
    <text>  [0..n]
</message>
```

The message element is used within the following other elements:
calculate proto validate

If the `message` element is a child of validate, it may contain multiple `text` elements, each of which corresponds with a different type of validation. The `name` attribute of the text element associates the message with the type of validation. Specifically, the child `text` element named `scriptTest` is used for script validation, the one named `nullTest` is used for null validation, and the one `formatTest` is used for format validation. It is erroneous to have more than one child element with the same name or with no name. If the `message` element contains a single un-named `text` element, the message it contains is used for all messages issued by the enclosing validate element.

If the `message` element is a child of calculate, it contains a single `text` element, which is displayed as specified in the calculate element's `override` attribute.

The id property

A unique identifier that may be used to identify this element as a target.

The text child

A content element that describes a single unit of data content representing a plain textual value.

For more information see "The text element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The numericEdit element

A user interface element that describes a widget intended to aid in the manipulation of numeric content.

```
<numericEdit

Properties:
    hScrollPolicy="auto | on | off"
    id="xml-id"
    use="cdata"
    usehref="cdata"
>
    <border>  [0..1]
    <comb>  [0..1]
    <extras>  [0..1]
    <margin>  [0..1]
</numericEdit>
```

The numericEdit element is used within the following other elements:
proto ui

The border property

A box model element that describes the border surrounding an object.

For more information see "The border element".

The comb property

An element that causes a field to be presented with vertical lines between the character positions.

For more information see "The comb element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The hScrollPolicy property

Controls the scrollability of the field in the horizontal direction.

auto

If the field is single-line it scrolls horizontally. Multi-line fields do not scroll horizontally.

on

A horizontal scroll bar is always displayed whether or not the input overflows the boundaries of the field. The field is scrollable regardless of whether it is a single-line or multi-line field.

off

The user is not allowed to enter characters beyond what can physically fit in the field width. This applies to typing and pasting from the clipboard. However data which is merged into the field

from the Data DOM is not restricted. If the data exceeds the field size the user may not be able to view all of it.

The id property

A unique identifier that may be used to identify this element as a target.

The margin property

A box model element that specifies one or more insets for an object.

For more information see "The margin element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The occur element

An element that describes the constraints over the number of allowable instances for its enclosing container.

```
<occur

Properties:
    id="xml-id"
    initial="1 | integer"
    max="1 | integer"
    min="1 | integer"
    use="cdata"
    usehref="cdata"
>
    <extras>  [0..1]
</occur>
```

The occur element is used within the following other elements:
pageArea pageSet proto subform subformSet

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The initial property

Specifies the initial number of occurrences for the enclosing subform element or subformSet element. The default is `1`. This property has no meaning when the container is a pageArea element or a pageSet element.

The max property

Specifies the maximum number of occurrences for the enclosing container, or `-1` to set no upper boundary for occurrences. This value defaults to the value of the `min` property. In the absence of a `min` property the default value varies depending upon the type of the enclosing container. If the enclosing container is a subform element or subformSet element the default is `1`. However if the enclosing container is a pageArea element or a pageSet element the default is `-1`.

The min property

Specifies the minimum number of occurrences for the enclosing container. If the enclosing container is a subform element or subformSet element the default is `1`. However if the enclosing container is a pageArea element or a pageSet element the default is `0`.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The oid element

An Object Identifier (OID) of the certificate policies that must be present in the signing certificate.

```
<oid

Properties:
    id="xml-id"
    name="xml-id"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</oid>
```

The oid element is used within the following other elements:
oids proto

Content

The Object Identifier string.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The oids element

A collection of Object Identifiers (OIDs) which apply to signing data with an XML digital signature.

```
<oids

Properties:
    id="xml-id"
    type="optional | required"
    use="cdata"
    usehref="cdata"
>

Children:
    <oid>  [0..n]
</oids>
```

The oids element is used within the following other elements:
certificates proto

Values that uniquely identify the issuer certificate. This element is only applicable if it has a sibling issuers element which is non-empty. The certificate policies extension is described in RFC 3280, Internet X.509 Public Key Infrastructure, Certificate and Certificate Revocation List (CRL) Profile [RFC3280].

The id property

A unique identifier that may be used to identify this element as a target.

The oid child

An Object Identifier (OID) of the certificate policies that must be present in the signing certificate.

For more information see "The oid element".

The type property

Specifies whether the values provided in the element should be treated as a restrictive or non-restrictive set.

optional

The values provided in the element are optional seed values from which the XFA processing application may choose. The XFA processing application may also supply its own value.

required

The values provided in the element are seed values from which the XFA processing application must choose.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The overflow element

An element that controls what happens when the parent subform or subform set overflows the current layout container.

```
<overflow

Properties:
     id="xml-id"
     leader="cdata"
     target="cdata"
     trailer="cdata"
     use="cdata"
     usehref="cdata"
>
</overflow>
```

The overflow element is used within the following other elements:
proto subform subformSet

The id property

A unique identifier that may be used to identify this element as a target.

The leader property

The value of this property is either a SOM expression (which can not start with '#') or a '#' followed by an XML ID. The SOM expression or XML ID points to a subform or subform set to be laid down at the top of the next layout container. When this property is empty or blank no special action is taken on overflow.

Note that this replaces the `overflowLeader` attribute on the deprecated `break` element.

The target property

The value of this property is either a SOM expression (which can not start with '#') or a '#' followed by an XML ID. The SOM expression or XML ID points to a `contentArea` or `pageArea` which becomes the next layout container. When this property is empty or blank the next layout container is determined by the properties of the current (overflowing) layout container.

Note that this replaces the `overflowTarget` attribute on the deprecated `break` element.

The trailer property

The value of this property is either a SOM expression (which can not start with '#') or a '#' followed by an XML ID. The SOM expression or XML ID points to a subform or subform set to be laid down at the bottom of the current (overflowing) layout container. When this property is empty or blank no special action is taken on overflow.

Note that this replaces the `overflowTrailer` attribute on the deprecated `break` element.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The pageArea element

An element that describes a rendering surface.

```
<pageArea

Properties:
    blankOrNotBlank="any | blank | notBlank"
    id="xml-id"
    initialNumber="1 | integer"
    name="xml-id"
    numbered="1 | integer"
    oddOrEven="any | odd | even"
    pagePosition="any | first | last | rest |
                  only"
    relevant="cdata"
    use="cdata"
    usehref="cdata"
>
    <desc>  [0..1]
    <extras>  [0..1]
    <medium>  [0..1]
    <occur>  [0..1]

Children:
    <area>  [0..n]
    <contentArea>  [0..n]
    <draw>  [0..n]
    <exclGroup>  [0..n]
    <field>  [0..n]
    <subform>  [0..n]
</pageArea>
```

The pageArea element is used within the following other elements:
pageSet proto

The area child

A container representing a geographical grouping of other containers.

For more information see "The area element".

The blankOrNotBlank property

Controls whether the page may appear in contexts where it is explicitly blank.

any

The page may be used in any context.

blank

The page may only be inserted in response to a break-to-even-page while on an even page, or a break-to-odd-page while on an odd page.

nonBlank

The page may only be inserted to hold content or to meet minimum occurrence rules.

This property is ignored within an orderedOccurrence pageSet.

The contentArea child

An element that describes a region within a page area eligible for receiving content.

For more information see "The contentArea element".

The desc property

An element to hold human-readable metadata.

For more information see "The desc element".

The draw child

A container element that contains non-interactive data content.

For more information see "The draw element".

The exclGroup child

A container element that describes a mutual exclusion relationship between a set of containers.

For more information see "The exclGroup element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The field child

A container element that describes a single interactive container capable of capturing and presenting data content.

For more information see "The field element".

The id property

A unique identifier that may be used to identify this element as a target.

The initialNumber property

This property supplies the page number if and only if the page is the first page in the document. Otherwise numbering is sequential.

The medium property

An element that describes a physical medium upon which to render. Some hybrid paper/glass media, such as PDF, may require both paper and glass properties.

For more information see "The medium element".

The name property

An identifier that may be used to identify this element in script expressions.

The numbered property

This property specifies whether the page is considered a numbered page.

Numbered pages contribute to the normal incrementing of page numbers, whereas un-numbered pages occur without incrementing page numbering.

1

The page area represents a numbered page. Therefore the instantiation of the page area contributes to the incrementing of the current page number.

0

The page area represents a un-numbered page. Therefore the instantiation of the page area does not contribute to the incrementing of the current page number.

The occur property

An element that describes the constraints over the number of allowable instances for its enclosing container.

For more information see "The occur element".

The oddOrEven property

controls whether the page may be in odd or even positions. Odd or even is determined by physical surface count, not by the page number. The first page in a document and every second page thereafter is odd, the other pages are even. When printing in duplex odd pages are on the front surface of a sheet and even on the back surface of a sheet.

any

This page can be in any position.

even

This page can only be placed in an even position (when printing in duplex, on the back of a sheet).

odd

This page can only be placed in an odd position (when printing in duplex, on the front of a sheet).

This property is ignored within an orderedOccurrence pageSet.

The pagePosition property

Controls in what context the page may be used within a contiguous sequence of pages from the same pageSet.

any

This page can be used in any context.

first

This page can only be used as the first page in a contiguous sequence.

last

This page can only be used as the last page in a contiguous sequence.

only

This page can only be used as the sole page in a sequence.

rest

This page can be used in any context except first or last in a sequence.

This property is ignored within an orderedOccurrence pageSet.

The relevant property

Specifies the views for which the enclosing object is relevant. The views themselves are specified in the config object.

Views are supplied as a space-separated list of viewnames: `relevant="[+|-]viewname [[+|-]viewname[...]]"`. A token of the form *viewname* or *+viewname* indicates the enclosing element should be included in that particular view. A token of the form `-viewname` indicates the element should be excluded from that particular view.

The viewnames `simplex`, `duplex`, and `preprinted` are particularly interesting here. By convention these are used for single-sided printing, double-sided printing, and printing onto preprinted stock, respectively.

The subform child

A container element that describes a single subform capable of enclosing other containers.

For more information see "The subform element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The pageSet element

An element that describes a set of related page area objects.

```
<pageSet

Properties:
    id="xml-id"
    name="xml-id"
    relation="orderedOccurrence | simplexPaginated | duplexPaginated"
    relevant="cdata"
    use="cdata"
    usehref="cdata"
>
    <extras>  [0..1]
    <occur>  [0..1]

Children:
    <pageArea>  [0..n]
    <pageSet>  [0..n]
</pageSet>
```

The pageSet element is used within the following other elements:
pageSet proto subform

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The occur property

An element that describes the constraints over the number of allowable instances for its enclosing container.

For more information see "The occur element".

The pageArea child

An element that describes a rendering surface.

For more information see "The pageArea element".

The pageSet child

An element that describes a set of related page area objects.

For more information see "The pageSet element".

The relation property

Selects the method used to choose what pageArea to use next.

`orderedOccurrence`

The pageArea objects are consumed in document order based on their occurrence indicators, ignoring their `oddOrEven`, `blankOrNotBlank` and `pagePosition` properties. This was the only method available prior to XFA 2.5.

`simplexPaginated`

pageArea objects are chosen according to need, ignoring `oddOrEven` and `blankOrNotBlank` properties but taking into account `pagePosition`.

`duplexPaginated`

pageArea objects are chosen according to need, taking into account `oddOrEven`, `blankOrNotBlank` and `pagePosition` properties.

The relevant property

Specifies the views for which the enclosing object is relevant. The views themselves are specified in the config object.

Views are supplied as a space-separated list of viewnames: `relevant="[+|-]`*`viewname`* `[[+|-]`*`viewname`*`[...]]"`. A token of the form *viewname* or *+viewname* indicates the enclosing element should be included in that particular view. A token of the form `-`*`viewname`* indicates the element should be excluded from that particular view.

If a container is excluded, it is not considered in the data binding process.

See also Concealing Containers Depending on View and Config Specification.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The para element

A formatting element that specifies default paragraph and alignment properties to be applied to the content of an enclosing container.

```
<para

Properties:
    hAlign="left | center | right | justify |
            justifyAll | radix"
    id="xml-id"
    lineHeight="0pt | measurement"
    marginLeft="0in | measurement"
    marginRight="0in | measurement"
    preserve="0 | cdata"
    radixOffset="0in | measurement"
    spaceAbove="0in | measurement"
    spaceBelow="0in | measurement"
    tabDefault="cdata"
    tabStops="cdata"
    textIndent="0in | measurement"
    use="cdata"
    usehref="cdata"
    vAlign="top | middle | bottom"
>
</para>
```

The para element is used within the following other elements:
caption draw exclGroup field proto subform

The hAlign property

Horizontal text alignment control.

center

Center horizontally within the available region.

justify

Left-align the last line and spread-justify the rest.

justifyAll

Spread-justify all lines to fill the available region.

left

Align with left edge of the available region.

radix

Align the radix indicator (decimal point or comma, depending upon locale) at the location specified by the `radixOffset` property of the para element. If there is no radix indicator, the last character is assumed to represent the units column.

`right`

Align with right edge of the available region.

The id property

A unique identifier that may be used to identify this element as a target.

The lineHeight property

A measurement specifying the line height to be applied to the paragraph content. Absolute omission or an empty specified value indicates that the font is to be used to determine the line height.

The marginLeft property

A measurement representing the left indentation of the paragraph. The default is zero.

The marginRight property

A measurement representing the right indentation of the paragraph. The default is zero.

The preserve property

This property specifies widow/orphan-style constraints on the overflow behavior of the content within the enclosing container.

This property has a lower precedence than any keep properties specified on the content within the enclosing container.

`0`

The content can be broken across an overflow boundary in an implementation-defined manner.

`integer`

An integer value greater than zero specifies the minimum quantity of content that must transition across the overflow boundary. For instance, specifying an integer value of `2` would prevent a single line of content from being widowed across the overflow boundary; it would result in a minimum of two lines of content transitioning across the overflow boundary.

`all`

Each paragraph of content must be kept intact and therefore cannot be broken across an overflow boundary.

The radixOffset property

A measurement representing the separation between the right margin and the radix point. If omitted, the value is assumed to be zero. This attribute is meaningful only if `hAlign` is `radix`.

The spaceAbove property

A measurement representing the vertical spacing in addition to the maximum font leading of the first line of the paragraph. The default is zero.

The spaceBelow property

A measurement representing the vertical spacing that appears after a paragraph. The default is zero.

The tabDefault property

A measurement representing the distance between default tab stops. The default is not to set default tab stops.

For more information see Tab Stops.

The tabStops property

A space-separated list of tab stop locations. Within the region from the left margin to the rightmost tab stop in the list, these replace the default tab stops specified by the `tabDefault` attribute. To the right of that point the default tab stops apply.

Each list entry consists of a keyword specifying the alignment at the tab stop, followed by a space, followed by the distance of the tab stop from the left margin. The tab stop alignment is one of the following:

`center`

Center-aligned tab stop

`left`

Left-aligned tab stop

`right`

Right-aligned tab stop

`decimal`

Tab-stop that aligns content around a radix point

For more information see Tab Stops.

The textIndent property

A measurement representing the horizontal positioning of the first line relative to the remaining lines in the paragraph. A negative value indicates a hanging indent whereas a positive value indicates first line indent. The default is zero.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The vAlign property

Vertical text alignment control.

`top`

Align with top of the available region.

`middle`

Center vertically within the available region.

`bottom`

Align with bottom of the available region.

`tabDefault`

Reserved for future use.

`tabStops`

Reserved for future use.

The passwordEdit element

A user interface element that describes a widget intended to aid in the manipulation of password content. Typically the user-interface will obscure any visual representation of the content.

```
<passwordEdit

Properties:
    hScrollPolicy="auto | on | off"
    id="xml-id"
    passwordChar="* | cdata"
    use="cdata"
    usehref="cdata"
>
    <border>   [0..1]
    <extras>   [0..1]
    <margin>   [0..1]
</passwordEdit>
```

The passwordEdit element is used within the following other elements:
proto ui

The border property

A box model element that describes the border surrounding an object.

For more information see "The border element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The hScrollPolicy property

Controls the scrollability of the field in the horizontal direction.

auto

If the field is single-line it scrolls horizontally. Multi-line fields do not scroll horizontally.

on

A horizontal scroll bar is always displayed whether or not the input overflows the boundaries of the field. The field is scrollable regardless of whether it is a single-line or multi-line field.

off

The user is not allowed to enter characters beyond what can physically fit in the field width. This applies to typing and pasting from the clipboard. However data which is merged into the field from the Data DOM is not restricted. If the data exceeds the field size the user may not be able to view all of it.

The id property

A unique identifier that may be used to identify this element as a target.

The margin property

A box model element that specifies one or more insets for an object.

For more information see "The margin element".

The passwordChar property

A single character to be echoed in place of each entered password character. The default is "*" (asterisk).

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The pattern element

A fill type element that describes a hatching pattern.

```
<pattern

Properties:
    id="xml-id"
    type="crossHatch | horizontal | vertical | diagonalLeft |
          diagonalRight | crossDiagonal"
    use="cdata"
    usehref="cdata"
>
    <color>  [0..1]
    <extras>  [0..1]
</pattern>
```

The pattern element is used within the following other elements:
fill proto

The pattern is rendered as a series of parallel strokes, drawn at an application-defined interval across the fill area. Some pattern variations draw a second set of strokes at right angles to the first set.

The strokes are drawn in the *foreground color* on top of a background that is pre-filled with the *background color*. The color element enclosed by the linear element determines the foreground color. The color element enclosed by the parent fill element determines the background color.

The color property

An element that describes a color.

For more information see "The color element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The type property

Specifies the appearance of the pattern.

`crossHatch`

The pattern appears as a series of intersecting horizontal and vertical lines.

`horizontal`

The pattern appears as a series of horizontal lines.

vertical

The pattern appears as a series of vertical lines.

diagonalLeft

The pattern appears as a series of diagonal lines proceeding from the top-left to the bottom-right.

diagonalRight

The pattern appears as a series of diagonal lines proceeding from the bottom-left to the top-right.

crossDiagonal

The pattern appears as a series of intersecting diagonal lines.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The picture element

A rendering element that describes input parsing and output formatting information.

```
<picture

Properties:
    id="xml-id"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</picture>
```

The picture element is used within the following other elements:
bind connect format proto ui validate

Content

This element encloses picture-data which is a special text format described in Picture Clause Specification.

The id property

A unique identifier that may be used to identify this element as a target.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The proto element

An element that describes a set of reusable element definitions, as described in the section Prototypes.

```
<proto

Properties:
>

Children:
    <arc>  [0..n]
    <area>  [0..n]
    <assist>  [0..n]
    <barcode>  [0..n]
    <bindItems>  [0..n]
    <bookend>  [0..n]
    <boolean>  [0..n]
    <border>  [0..n]
    <break>  [0..n]
    <breakAfter>  [0..n]
    <breakBefore>  [0..n]
    <button>  [0..n]
    <calculate>  [0..n]
    <caption>  [0..n]
    <certificate>  [0..n]
    <certificates>  [0..n]
    <checkButton>  [0..n]
    <choiceList>  [0..n]
    <color>  [0..n]
    <comb>  [0..n]
    <connect>  [0..n]
    <contentArea>  [0..n]
    <corner>  [0..n]
    <date>  [0..n]
    <dateTime>  [0..n]
    <dateTimeEdit>  [0..n]
    <decimal>  [0..n]
    <defaultUi>  [0..n]
    <desc>  [0..n]
    <digestMethod>  [0..n]
    <digestMethods>  [0..n]
    <draw>  [0..n]
    <edge>  [0..n]
    <encoding>  [0..n]
    <encodings>  [0..n]
    <encrypt>  [0..n]
    <event>  [0..n]
    <exclGroup>  [0..n]
    <exData>  [0..n]
    <execute>  [0..n]
    <exObject>  [0..n]
    <extras>  [0..n]
    <field>  [0..n]
```

<fill> [0..n]
<filter> [0..n]
<float> [0..n]
<font> [0..n]
<format> [0..n]
<handler> [0..n]
<image> [0..n]
<imageEdit> [0..n]
<integer> [0..n]
<issuers> [0..n]
<items> [0..n]
<keep> [0..n]
<keyUsage> [0..n]
<line> [0..n]
<linear> [0..n]
<manifest> [0..n]
<margin> [0..n]
<mdp> [0..n]
<medium> [0..n]
<message> [0..n]
<numericEdit> [0..n]
<occur> [0..n]
<oid> [0..n]
<oids> [0..n]
<overflow> [0..n]
<pageArea> [0..n]
<pageSet> [0..n]
<para> [0..n]
<passwordEdit> [0..n]
<pattern> [0..n]
<picture> [0..n]
<radial> [0..n]
<reason> [0..n]
<reasons> [0..n]
<rectangle> [0..n]
<ref> [0..n]
<script> [0..n]
<setProperty> [0..n]
<signature> [0..n]
<signData> [0..n]
<signing> [0..n]
<solid> [0..n]
<speak> [0..n]
<stipple> [0..n]
<subform> [0..n]
<subformSet> [0..n]
<subjectDN> [0..n]
<subjectDNs> [0..n]
<submit> [0..n]
<text> [0..n]
<textEdit> [0..n]
<time> [0..n]
<timeStamp> [0..n]

```
    <toolTip>  [0..n]
    <traversal>  [0..n]
    <traverse>  [0..n]
    <ui>  [0..n]
    <validate>  [0..n]
    <value>  [0..n]
    <variables>  [0..n]
</proto>
```

The proto element is used within the following other elements:
subform

The arc child

A curve that can be used for describing either an arc or an ellipse.

For more information see "The arc element".

The area child

A container representing a geographical grouping of other containers.

For more information see "The area element".

The assist child

An element that supplies additional information about a container for users of interactive applications.

For more information see "The assist element".

The barcode child

An element that represents a barcode.

For more information see "The barcode element".

The bindItems child

An element that extracts data into an item list.

For more information see "The bindItems element".

The bookend child

An element controlling content that is inserted to "bookend" the contents of the parent object.

For more information see "The bookend element".

The boolean child

A content element describing single unit of data content representing a Boolean logical value.

For more information see "The boolean element".

The border child

A box model element that describes the border surrounding an object.

For more information see "The border element".

The break child

(DEPRECATED) An element that describes the constraints on moving to a new page or content area before or after rendering an object.

For more information see "The break element".

The breakAfter child

An element that controls actions to be taken after laying down the contents of the parent object.

For more information see "The breakAfter element".

The breakBefore child

An element that controls actions to be taken before laying down the contents of the parent object.

For more information see "The breakBefore element".

The button child

A user interface element that describes a push-button widget.

For more information see "The button element".

The calculate child

An automation element that controls the calculation of its container's value.

For more information see "The calculate element".

The caption child

A box model element that describes a descriptive label associated with an object.

For more information see "The caption element".

The certificate child

An element that holds a suitable Base64 DER-encoded X.509v3 certificate.

For more information see "The certificate element".

The certificates child

An element that holds a collection of certificate filters used to identify the signer.

For more information see "The certificates element".

The checkButton child

A user interface element that describes either a checkbox or radio-button widget.

For more information see "The checkButton element".

The choiceList child

A user interface element that describes a widget presenting a list of options. The list of options is specified by one or more sibling `items` elements.

For more information see "The choiceList element".

The color child

An element that describes a color.

For more information see "The color element".

The comb child

An element that causes a field to be presented with vertical lines between the character positions.

For more information see "The comb element".

The connect child

An element that describes the relationship between its containing object and a connection to a web service, schema, or data description.

Connections are defined outside the template in a separate packet with its own schema. See the XFA Connection Set Specification for more information.

For more information see "The connect element".

The contentArea child

An element that describes a region within a page area eligible for receiving content.

For more information see "The contentArea element".

The corner child

A formatting element that describes the appearance of a vertex between two edges

For more information see "The corner element".

The date child

A content element that describes a single unit of data content representing a date.

For more information see "The date element".

The dateTime child

A content element that describes a single unit of data content representing a date and time value.

For more information see "The dateTime element".

The dateTimeEdit child

A user interface element describing a widget intended to aid in the selection of date and/or time.

For more information see "The dateTimeEdit element".

The decimal child

A content type element that describes a single unit of data content representing a number with a fixed number of digits after the decimal.

For more information see "The decimal element".

The defaultUi child

An element for widgets whose depiction is delegated to the XFA application.

For more information see "The defaultUi element".

The desc child

An element to hold human-readable metadata.

For more information see "The desc element".

The digestMethod child

An element to hold the name of an acceptable digest method for a signature.

For more information see "The digestMethod element".

The digestMethods child

An element to hold a list of names of acceptable digest methods for a signature.

For more information see "The digestMethods element".

The draw child

A container element that contains non-interactive data content.

For more information see "The draw element".

The edge child

A formatting element that describes an arc, line, or one side of a border or rectangle.

For more information see "The edge element".

The encoding child

An element holding the name of an acceptable recipe for signature encoding.

For more information see "The encoding element".

The encodings child

An element holding a list of names of acceptable recipes for signature encoding.

For more information see "The encodings element".

The encrypt child

An element that controls encryption of barcode or submit data.

For more information see "The encrypt element".

The event child

An automation element that causes a script to be executed or data to be submitted whenever a particular event occurs.

For more information see "The event element".

The exclGroup child

A container element that describes a mutual exclusion relationship between a set of containers.

For more information see "The exclGroup element".

The exData child

A content element that describes a single unit of data of a foreign datatype.

For more information see "The exData element".

The execute child

An element that causes an event to invoke a WSDL-based web service.

For more information see "The execute element".

The exObject child

An element that describes a single program or implementation-dependent foreign object.

For more information see "The exObject element".

The extras child

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The field child

A container element that describes a single interactive container capable of capturing and presenting data content.

For more information see "The field element".

The fill child

A formatting element that applies a color and optional rendered designs to the region enclosed by an object.

For more information see "The fill element".

The filter child

An element that contains the criteria for filtering signing certificates used to generate XML digital signatures.

For more information see "The filter element".

The float child

A content element that describes a single unit of data content representing a floating point value.

For more information see "The float element".

The font child

A formatting element that describes a font.

For more information see "The font element".

The format child

A rendering element that encloses output formatting information such as the picture clause.

For more information see "The format element".

The handler child

An element controlling what signature handler is used for a data-signing operation for an XML digital signature.

For more information see "The handler element".

The image child

A content element that describes a single image.

For more information see "The image element".

The imageEdit child

A user interface element that encloses a widget intended to aid in the manipulation of image content.

For more information see "The imageEdit element".

The integer child

A content element that describes a single unit of data content representing an integer value.

For more information see "The integer element".

The issuers child

A collection of issuer certificates that are acceptable for data signing an XML digital signature.

For more information see "The issuers element".

The items child

An element that supplies a set of values for a choice list or a check button.

For more information see "The items element".

The keep child

An element that describes the constraints on keeping subforms together within a page or content area.

For more information see "The keep element".

The keyUsage child

An element that specifies the key usage settings required in the signing certificate.

For more information see "The keyUsage element".

The line child

A content element that describes a single rendered line.

For more information see "The line element".

The linear child

A fill type element that describes a linear gradient fill.

For more information see "The linear element".

The manifest child

An element that contains a list of references to all the nodes that are included in a node collection.

For more information see "The manifest element".

The margin child

A box model element that specifies one or more insets for an object.

For more information see "The margin element".

The mdp child

An element that controls an MDP+ signature.

For more information see "The mdp element".

The medium child

An element that describes a physical medium upon which to render. Some hybrid paper/glass media, such as PDF, may require both paper and glass properties.

For more information see "The medium element".

The message child

A automation element that holds one or more sub-elements containing messages used with validations and calculations.

For more information see "The message element".

The numericEdit child

A user interface element that describes a widget intended to aid in the manipulation of numeric content.

For more information see "The numericEdit element".

The occur child

An element that describes the constraints over the number of allowable instances for its enclosing container.

For more information see "The occur element".

The oid child

An Object Identifier (OID) of the certificate policies that must be present in the signing certificate.

For more information see "The oid element".

The oids child

A collection of Object Identifiers (OIDs) which apply to signing data with an XML digital signature.

For more information see "The oids element".

The overflow child

An element that controls what happens when the parent subform or subform set overflows the current layout container.

For more information see "The overflow element".

The pageArea child

An element that describes a rendering surface.

For more information see "The pageArea element".

The pageSet child

An element that describes a set of related page area objects.

For more information see "The pageSet element".

The para child

A formatting element that specifies default paragraph and alignment properties to be applied to the content of an enclosing container.

For more information see "The para element".

The passwordEdit child

A user interface element that describes a widget intended to aid in the manipulation of password content. Typically the user-interface will obscure any visual representation of the content.

For more information see "The passwordEdit element".

The pattern child

A fill type element that describes a hatching pattern.

For more information see "The pattern element".

The picture child

A rendering element that describes input parsing and output formatting information.

For more information see "The picture element".

The radial child

A fill type element that describes a radial gradient fill.

For more information see "The radial element".

The reason child

An element containing a candidate reason string for inclusion in an XML digital signature.

For more information see "The reason element".

The reasons child

An element containing a choice of reason strings for including with an XML Digital Signature.

For more information see "The reasons element".

The rectangle child

A content element that describes a single rendered rectangle.

For more information see "The rectangle element".

The ref child

An element holding an XFA-SOM expression that identifies a node to be included in an XML digital signature.

For more information see "The ref element".

The script child

An automation element that contains a script.

For more information see "The script element".

The setProperty child

An element that causes a property of the container to be copied from a value in the XFA Data DOM or from data returned by a web service.

For more information see "The setProperty element".

The signature child

A user interface element that describes a widget intended to allow a user to sign a completed form, making it a document of record.

For more information see "The signature element".

The signData child

An element controlling an XML digital signature.

For more information see "The signData element".

The signing child

A collection of signing certificates that are acceptable for use in affixing an XML digital signature.

For more information see "The signing element".

The solid child

A fill type element that describes a solid fill.

For more information see "The solid element".

The speak child

An audible prompt describing the contents of a container. This element is ignored by non-interactive applications.

For more information see "The speak element".

The stipple child

A fill type element that describes a stippling effect.

For more information see "The stipple element".

The subform child

A container element that describes a single subform capable of enclosing other containers.

For more information see "The subform element".

The subformSet child

An element that describes a set of related subform objects.

For more information see "The subformSet element".

The subjectDN child

An element that contains a key-value pair used to specify the Subject Distinguished Name (DN) that must be present within the certificate for it to be acceptable for signing.

For more information see "The subjectDN element".

The subjectDNs child

An element that contains the collection of key-value pairs used to specify the Subject Distinguished Name (DN) that must be present within the certificate for it to be acceptable for signing.

For more information see "The subjectDNs element".

The submit child

An element that describes how to submit data to a host, using an HTTP POST operation.

For more information see "The submit element".

The text child

A content element that describes a single unit of data content representing a plain textual value.

For more information see "The text element".

The textEdit child

A user interface element that encloses a widget intended to aid in the manipulation of textual content.

For more information see "The textEdit element".

The time child

A content element that describes a single unit of data content representing a time value.

For more information see "The time element".

The timeStamp child

An element that controls the time-stamping of a signature.

For more information see "The timeStamp element".

The toolTip child

An element that supplies text for a tool tip. This element is ignored by non-interactive applications.

For more information see "The toolTip element".

The traversal child

An element that links its container to other objects in sequence.

For more information see "The traversal element".

The traverse child

An element that declares a single link from its container to another object in a unidirectional chain of links.

For more information see "The traverse element".

The ui child

A user-interface element that encloses the actual user interface widget element.

For more information see "The ui element".

The validate child

A automation element that controls validation of user-supplied data.

For more information see "The validate element".

The value child

A content element that encloses a single unit of data content.

For more information see "The value element".

The variables child

An element to hold document variables.

For more information see "The variables element".

The radial element

A fill type element that describes a radial gradient fill.

```
<radial

Properties:
    id="xml-id"
    type="toEdge | toCenter"
    use="cdata"
    usehref="cdata"
>
    <color>  [0..1]
    <extras>  [0..1]
</radial>
```

The radial element is used within the following other elements:
fill proto

A radial gradient fill appears as the *start color* at the center of the fill area, and the *end color* at the outer edges. Between those two extremes, the color gradually changes from start color to end color. Alternately, the roles of the start and end colors may be reversed.

The color element enclosed by the radial element determines the end color. The color element enclosed by the parent fill element determines the start color.

The color property

An element that describes a color.

For more information see "The color element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The type property

Specifies the direction of the color transition.

`toEdge`

The start color appears at the center of the object and transitions into the end color at the outer edge.

`toCenter`

The start color appears at the outer edge of the object and transitions into the end color at the center.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The reason element

An element containing a candidate reason string for inclusion in an XML digital signature.

```
<reason

Properties:
    id="xml-id"
    name="xml-id"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</reason>
```

The reason element is used within the following other elements:
proto reasons

Content

An acceptable reason.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The reasons element

An element containing a choice of reason strings for including with an XML Digital Signature.

```
<reasons

Properties:
    id="xml-id"
    type="optional | required"
    use="cdata"
    usehref="cdata"
>

Children:
    <reason>  [0..n]
</reasons>
```

The reasons element is used within the following other elements:
filter proto

The id property

A unique identifier that may be used to identify this element as a target.

The reason child

An element containing a candidate reason string for inclusion in an XML digital signature.

For more information see "The reason element".

The type property

Specifies whether it is mandatory to include a reason string or not.

optional

A reason is not required.

required

A reason is required.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The rectangle element

A content element that describes a single rendered rectangle.

```
<rectangle

Properties:
    hand="even | left | right"
    id="xml-id"
    use="cdata"
    usehref="cdata"
>
    <corner>  [0..4]
    <edge>  [0..4]
    <fill>  [0..1]
</rectangle>
```

The rectangle element is used within the following other elements:
proto value

The edges of a rectangle are rendered in a clockwise fashion, starting from the top left corner. This has implications for the rectangle's handedness. In particular, a left-handed stroke will appear immediately outside the rectangle's edge, while a right-handed edge will appear immediately inside. Such behavior is consistent with borders, but not arcs.

The corner property

A formatting element that describes the appearance of a vertex between two edges

For more information see "The corner element".

The edge property

A formatting element that describes an arc, line, or one side of a border or rectangle.

For more information see "The edge element".

The fill property

A formatting element that applies a color and optional rendered designs to the region enclosed by an object.

For more information see "The fill element".

The hand property

Description of the handedness of a line or edge.

even

Center the displayed line on the underlying vector or arc.

left

Position the displayed line immediately to the left of the underlying vector or arc, when following that line from its start point to its end point.

right

Position the displayed line immediately to the right of the underlying vector or arc, when following that line from its start point to its end point.

The id property

A unique identifier that may be used to identify this element as a target.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The ref element

An element holding an XFA-SOM expression that identifies a node to be included in an XML digital signature.

```
<ref

Properties:
     id="xml-id"
     use="cdata"
     usehref="cdata"
>
     ...pcdata...
</ref>
```

The ref element is used within the following other elements:
manifest proto

The reference must be to a node that is written out to XML when the form is saved. The reference is stored in the XML digital signature manifest as an XPath expression pointing to the corresponding element in the XML document. The computed signature includes that element and all of its children.

If the value of this element includes the destination of the signature, the signature handler automatically excludes the signature from the signature value it calculates.

Examples of SOM expressions used as the value of this element are foo[*] and mySubform..myField

Content

The XFA-SOM expression.

The id property

A unique identifier that may be used to identify this element as a target.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The script element

An automation element that contains a script.

```
<script

Properties:
    binding="XFA | cdata"
    contentType="application/x-formcalc | cdata"
    id="xml-id"
    name="xml-id"
    runAt="client | server | both"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</script>
```

The script element is used within the following other elements:
breakAfter breakBefore calculate event proto traverse validate variables

Content

This element contains a script in the scripting language specified by the contentType property.

The binding property

Identifies the type of application to which the script is directed.

XFA

The script is to be applied by standard XFA applications.

cdata

Any value other than XFA signifies that the script may be ignored by standard XFA applications.

The contentType property

The type of content in the enclosed script.

The following values are allowed:

application/x-formcalc

A FormCalc script, as defined in FormCalc Specification.

cdata

Support for other script types, such as `application/javascript` is implementation-defined.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The runAt property

Specifies where the script is allowed to run.

This restrictions also applies when this script is called by another script. Hence a script marked to run only on one side can only be called on that side.

The value must be one of the following:

`client`

The script runs only on the client.

`server`

The script runs only on the server.

`both`

The script runs on both client and server.

There are important security considerations when using scripts that may run on the server. See "Discarding Unexpected Submitted Packets" for a full discussion of security issues.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The setProperty element

An element that causes a property of the container to be copied from a value in the XFA Data DOM or from data returned by a web service.

```
<setProperty

Properties:
     connection="cdata"
     ref="cdata"
     target="cdata"
>
</setProperty>
```

The setProperty element is used within the following other elements:
draw exclGroup field proto subform

The connection property

Optionally supplies the name of an associated connection to a web service. When this is supplied and non-empty the `ref` property is interpreted differently.

The ref property

Supplies a pointer to the data to be copied. This is a SOM expression with the restriction that it cannot contain the string "..". It may be a relative SOM expression. If there is no associated web service then the expression is interpreted relative to the enclosing container. If there is an associated web service then it is interpreted relative to the nearest ancestor that asserts a fully-qualified SOM expression as its value of `ref` for the same connection.

The target property

A SOM expression identifying the property to be set. The expression is evaluated relative to the container. The target must be a property or subproperty of the container.

Almost any property of the container can be the target of this element. However the copying is done near the end of the data merge process. For some properties it is too late because the property has already had its effect. Also it is not recommended to use this element to set the `value` property of a field or exclusion group; use an explicit data reference instead.

The signature element

A user interface element that describes a widget intended to allow a user to sign a completed form, making it a document of record.

```
<signature

Properties:
    id="xml-id"
    type="PDF1.3 | PDF1.6"
    use="cdata"
    usehref="cdata"
>
    <border>  [0..1]
    <extras>  [0..1]
    <filter>  [0..1]
    <manifest>  [0..1]
    <margin>  [0..1]
</signature>
```

The signature element is used within the following other elements:
proto ui

Note that this element is *not* used for an XML digital signature. This is used for PDF signatures only. The presence of a `manifest` child further indicates that this is an MDP+ signature.

The border property

A box model element that describes the border surrounding an object.

For more information see "The border element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The filter property

An element that contains the criteria for filtering signing certificates used to generate XML digital signatures.

For more information see "The filter element".

The id property

A unique identifier that may be used to identify this element as a target.

The manifest property

An element that contains a list of references to all the nodes that are included in a node collection.

For more information see "The manifest element".

The margin property

A box model element that specifies one or more insets for an object.

For more information see "The margin element".

The type property

Controls the signature algorithm used. The default is `PDF1.3` which is the signature algorithm used in Acrobat 4, 5, and 6. The value `PDF1.6` signifies the algorithm used in Acrobat 7 and 8. These algorithms are described in the PDF manual.

The `manifest` and `filter` properties are only valid if the signature type is `PDF1.6`.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The signData element

An element controlling an XML digital signature.

```
<signData

Properties:
    id="xml-id"
    operation="sign | verify | clear"
    ref="cdata"
    target="cdata"
    use="cdata"
    usehref="cdata"
>
    <filter>  [0..1]
    <manifest>  [0..1]
</signData>
```

The signData element is used within the following other elements:
event proto submit

The filter property

An element that contains the criteria for filtering signing certificates used to generate XML digital signatures.

For more information see "The filter element".

The id property

A unique identifier that may be used to identify this element as a target.

The manifest property

An element that contains a list of references to all the nodes that are included in a node collection.

For more information see "The manifest element".

The operation property

The operation to be performed.

sign

Create a digital signature.

verify

Verify that the portion of the document included in the signature manifest matches the signature.

clear

Remove the signature, if any.

The ref property

A SOM expression controlling where the signature is placed during a `sign` operation. During signature validate and remove operations, ref specifies the location of the signature that should be validated or removed.

The target property

The XML ID for the XML digital signature.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The signing element

A collection of signing certificates that are acceptable for use in affixing an XML digital signature.

```
<signing

Properties:
    id="xml-id"
    type="optional | required"
    use="cdata"
    usehref="cdata"
>

Children:
    <certificate>  [0..n]
</signing>
```

The signing element is used within the following other elements:
certificates proto

The handler uses the certificates in this element to populate the default list certificates from which the signor can choose.

The certificate child

An element that holds a suitable Base64 DER-encoded X.509v3 certificate.

For more information see "The certificate element".

The id property

A unique identifier that may be used to identify this element as a target.

The type property

Specifies whether the values provided in the element should be treated as a restrictive or non-restrictive set.

optional

The values provided in the element are optional seed values from which the XFA processing application may choose. The XFA processing application may also supply its own value. The application typically allows a person filling out the form to choose from the values provided or to specify his own value.

required

The values provided in the element are seed values from which the XFA processing application must choose. The application typically allows a person filling out the form to choose from only those values provided in the element.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The solid element

A fill type element that describes a solid fill.

```
<solid

Properties:
    id="xml-id"
    use="cdata"
    usehref="cdata"
>
    <extras>   [0..1]
</solid>
```

The solid element is used within the following other elements:
fill proto

The color element enclosed by the parent fill element determines the solid fill color.

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The speak element

An audible prompt describing the contents of a container. This element is ignored by non-interactive applications.

```
<speak

Properties:
    disable="0 | 1"
    id="xml-id"
    priority="custom | toolTip | caption | name"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</speak>
```

The speak element is used within the following other elements:
assist proto

Content

This property may supply text to be enunciated as an audible prompt.

This property may be empty or not supplied. When an interactive application prepares to issue an audible prompt, it searches for text in a search path that includes the speak element, the associated toolTip element, the associated caption element, and the container's name. The order of the search path is determined by the `priority` property.

The disable property

Inhibits the audible prompt.

1

An audible prompt will be produced if the field is not hidden or invisible.

0

There will not be an audible prompt.

The default value of this property is `1`.

The id property

A unique identifier that may be used to identify this element as a target.

The priority property

Alters the search path for text to speak. Whichever element is named in this attribute moves to the front of the search path. The other elements retain their relative order. The default order is the order in which the values are shown below.

The value must be one of:

custom

The search order is speak,tooltip, caption, the container's name.

caption

The search order is caption,speak, tooltip, the container's name.

name

The search order is the container's name, speak, tooltip, caption.

tooltip

The search order is tooltip,speak, caption, the container's name.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of proto. Any object of the appropriate class can be used as a prototype.

If both use and usehref are non-empty usehref takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both use and usehref are non-empty usehref takes precedence.

The stipple element

A fill type element that describes a stippling effect.

```
<stipple

Properties:
    id="xml-id"
    rate="50 | integer"
    use="cdata"
    usehref="cdata"
>
    <color>  [0..1]
    <extras>  [0..1]
</stipple>
```

The stipple element is used within the following other elements:
fill proto

A stipple fill appears as the stippling of a *stipple color* on top of a solid *background color*

The color element enclosed by the stipple element determines the stipple color. The color element enclosed by the parent fill element determines the background color.

The color property

An element that describes a color.

For more information see "The color element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The rate property

This property specifies the percentage of stipple color that is stippled over the background color. The background color is not specified by this element.

The stipple-rate is an integer between 0 and 100 inclusive where `0` results in no visible stippling drawn over the background color and `100` results in a complete obscuring of the background color by filling the area completely with stipple color. Any stipple rate between 0 and 100 results in a varying blend of background color and an overlaid stipple color. For instance, a stipple rate of `50` results in an equal blend of background color and stipple color.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The subform element

A container element that describes a single subform capable of enclosing other containers.

```
<subform

Properties:
    allowMacro="0 | 1"
    anchorType="topLeft | topCenter | topRight | middleLeft |
                middleCenter | middleRight | bottomLeft | bottomCenter |
                bottomRight"
    colSpan="1 | integer"
    columnWidths="cdata"
    h="0in | measurement"
    id="xml-id"
    layout="position | lr-tb | rl-tb | tb |
            table | row"
    locale="cdata"
    maxH="0in | measurement"
    maxW="0in | measurement"
    minH="0in | measurement"
    minW="0in | measurement"
    name="xml-id"
    presence="visible | invisible | hidden"
    relevant="cdata"
    restoreState="manual | auto"
    scope="name | none"
    use="cdata"
    usehref="cdata"
    w="0in | measurement"
    x="0in | measurement"
    y="0in | measurement"
>
    <assist>  [0..1]
    <bind>  [0..1]
    <bookend>  [0..1]
    <border>  [0..1]
    <break>  [0..1]
    <calculate>  [0..1]
    <desc>  [0..1]
    <extras>  [0..1]
    <keep>  [0..1]
    <margin>  [0..1]
    <occur>  [0..1]
    <overflow>  [0..1]
    <pageSet>  [0..1]
    <para>  [0..1]
    <traversal>  [0..1]
    <validate>  [0..1]
    <variables>  [0..1]

Children:
    <area>  [0..n]
```

```
    <breakAfter>  [0..n]
    <breakBefore>  [0..n]
    <connect>  [0..n]
    <draw>  [0..n]
    <event>  [0..n]
    <exclGroup>  [0..n]
    <exObject>  [0..n]
    <field>  [0..n]
    <proto>  [0..n]
    <setProperty>  [0..n]
    <subform>  [0..n]
    <subformSet>  [0..n]
</subform>
```

The subform element is used within the following other elements:
area pageArea proto subform subformSet template

The allowMacro property

This property specifies whether to permit the processing application to optimize output by generating a printer macro for all of the subform's draw content. The use of macros may have an impact on the z-order of objects.

1

The processing application is permitted to utilize a printer macro for this subform.

0

The processing application is forbidden from utilizing a printer macro for this subform.

The anchorType property

Location of the container's anchor point when placed with positioned layout strategy.

topLeft

Top left corner of the nominal extent.

topCenter

Center of the top edge of the nominal extent.

topRight

Top right corner of the nominal extent.

middleLeft

Middle of the left edge of the nominal extent.

middleCenter

Middle of the nominal extent.

middleRight

Middle of the right edge of the nominal extent.

bottomLeft

Bottom left corner of the nominal extent.

bottomCenter

Center of the bottom edge of the nominal extent.

bottomRight

Bottom right corner of the nominal extent.

The area child

A container representing a geographical grouping of other containers.

For more information see "The area element".

The assist property

An element that supplies additional information about a container for users of interactive applications.

For more information see "The assist element".

The bind property

An element that controls the behavior during merge operations of its enclosing element.

For more information see "The bind element".

The bookend property

An element controlling content that is inserted to "bookend" the contents of the parent object.

For more information see "The bookend element".

The border property

A box model element that describes the border surrounding an object.

For more information see "The border element".

The break property

(DEPRECATED) An element that describes the constraints on moving to a new page or content area before or after rendering an object.

For more information see "The break element".

The breakAfter child

An element that controls actions to be taken after laying down the contents of the parent object.

For more information see "The breakAfter element".

The breakBefore child

An element that controls actions to be taken before laying down the contents of the parent object.

For more information see "The breakBefore element".

The calculate property

An automation element that controls the calculation of its container's value.

For more information see "The calculate element".

The colSpan property

Number of columns spanned by this object, when used inside a subform with a layout type of `row`. Defaults to 1.

The columnWidths property

Widths for columns of a table. Ignored unless the `layout` property is set to `table`.

The value of this property is a set of space-separated tokens. Each token must be a measurement or "-1". The presence of a measurement causes the corresponding column to be set to that width. The presence of "-1" causes the corresponding column to grow to the width of the widest content for that column across all rows of the table.

The connect child

An element that describes the relationship between its containing object and a connection to a web service, schema, or data description.

Connections are defined outside the template in a separate packet with its own schema. See the XFA Connection Set Specification for more information.

For more information see "The connect element".

The desc property

An element to hold human-readable metadata.

For more information see "The desc element".

The draw child

A container element that contains non-interactive data content.

For more information see "The draw element".

The event child

An automation element that causes a script to be executed or data to be submitted whenever a particular event occurs.

For more information see "The event element".

The exclGroup child

A container element that describes a mutual exclusion relationship between a set of containers.

For more information see "The exclGroup element".

The exObject child

An element that describes a single program or implementation-dependent foreign object.

For more information see "The exObject element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The field child

A container element that describes a single interactive container capable of capturing and presenting data content.

For more information see "The field element".

The h property

Height for layout purposes. A measurement value for `h` overrides any growth range allowed by the `minH` and `maxH` attributes. The absolute omission of this attribute or a value specified as an empty string indicates that the `minH` and `maxH` must be respected.

This attribute has no default. Setting this attribute to "-1" is an error.

The id property

A unique identifier that may be used to identify this element as a target.

The keep property

An element that describes the constraints on keeping subforms together within a page or content area.

For more information see "The keep element".

The layout property

Layout strategy to be used within this element.

`position`

The content of the element is positioned according to the to the location information expressed on the content elements.

`lr-tb`

The content of the element is flowed in a direction proceeding from left to right and top to bottom.

`rl-tb`

The content of the element is flowed in a direction proceeding from right to left and top to bottom.

`row`

This is an inner element of a table, representing one or more rows. The objects contained in this element are cells of the table and their height and width attributes, if any, are ignored. The cells are laid out from right to left and each one is adjusted to the height of the row and the width of one or more contiguous columns.

table

This is the outer element of a table. Each of its child subforms or exclusion groups must have its `layout` property set to `row`. The rows of the table are laid out from top to bottom.

tb

The content of the element is flowed in a direction proceeding from top to bottom.

The locale property

A designator that influences the locale used to format the localizable content of this element. Such localizable content includes currency and time/date. Locale affects the representation of data formatted, validated, or parsed by picture clauses. Locale is also considered by certain FormCalc functions.

This designator also influences the default direction of text flow within this element. The text layout engine may override this within portions or all of such text as per the rules in the *Unicode Annex 9* [UAX-9] reference.

The value of this property must be one of the following:

ambient

Causes the ambient locale of the XFA application to be used.

localeName

A valid locale name that conforms to the syntax: language[_country]. Examples of valid locales are `zh` for Chinese and `en_CA` for English specific for Canada. For a complete list of valid locale values, refer to the *IETF RFC 1766* [RFC1766] and *ISO 639* [ISO-639-1] */ISO 3166* [ISO-3166-1] specifications. Note that this is the same set of locale names used by the `xml:lang` attribute defined in [XML1.0].

When this property is absent or empty the default behavior is to inherit the parent object's locale. If the outermost subform does not specify a locale it uses the ambient locale from the operating system. If the operating system does not supply a locale it falls back onto `en_US`.

The margin property

A box model element that specifies one or more insets for an object.

For more information see "The margin element".

The maxH property

Maximum height for layout purposes. This attribute is relevant only if the enclosing container element is growable and has an `h` attribute whose value is null. If an `h` attribute is supplied, the container is not vertically growable and this attribute is ignored.

If this attribute is not supplied, there is no limit. This attribute has no default. If `h` is omitted, a value must be supplied for this attribute. Setting this attribute to "-1" is an error.

The maxW property

Maximum width for layout purposes.This attribute is relevant only if the enclosing container element is growable and has a `w` attribute whose value is null. If a `w` attribute is supplied, the container is not horizontally growable and this attribute is ignored.

This attribute has no default. If `w` is omitted, a value must be supplied for this attribute. Setting this attribute to "-1" is an error.

The minH property

Minimum height for layout purposes. This attribute is relevant only if the enclosing container element is growable and has an `h` attribute whose value is null. If an `h` attribute is supplied, the container is not vertically growable and this attribute is ignored.

This attribute has no default. If `h` is omitted, a value must be supplied for this attribute. Setting this attribute to "-1" is an error.

The minW property

Minimum width for layout purposes. This attribute is relevant only if the enclosing container element is growable and has a `w` attribute whose value is null. If a `w` attribute is supplied, the container is not horizontally growable and this attribute is ignored.

This attribute has no default. If `w` is omitted, a value must be supplied for this attribute. Setting this attribute to "-1" is an error.

The name property

An identifier that may be used to identify this element in script expressions.

The occur property

An element that describes the constraints over the number of allowable instances for its enclosing container.

For more information see "The occur element".

The overflow property

An element that controls what happens when the parent subform or subform set overflows the current layout container.

For more information see "The overflow element".

The pageSet property

An element that describes a set of related page area objects.

For more information see "The pageSet element".

The para property

A formatting element that specifies default paragraph and alignment properties to be applied to the content of an enclosing container.

For more information see "The para element".

The presence property

Visibility control.

`visible`

Make it visible.

invisible

Make it transparent. Although invisible it still takes up space.

hidden

Hide it. It is not displayed and does not take up space.

The proto child

An element that describes a set of reusable element definitions, as described in the section Prototypes.

For more information see "The proto element".

The relevant property

Specifies the views for which the enclosing object is relevant. The views themselves are specified in the config object.

Views are supplied as a space-separated list of viewnames: `relevant="[+|-]viewname [[+|-]viewname[...]]"`. A token of the form *viewname* or *+viewname* indicates the enclosing element should be included in that particular view. A token of the form `-viewname` indicates the element should be excluded from that particular view.

If a container is excluded, it is not considered in the data binding process.

See also Concealing Containers Depending on View and Config Specification.

The restoreState property

Controls whether the form state is automatically saved when closed and restored on reopening.

manual

Script may restore specific properties but nothing happens automatically. This setting is required for certified documents.

auto

The whole state of the document is saved at closing and restored automatically upon re-opening. This includes data which was manually entered to override calculations, and data which was retained despite generating a validation warning. As far as possible it is as though the original session was never interrupted. This setting can not be used for certified documents.

This property is only meaningful on the root subform.

The scope property

Controls participation of the subform in data binding and SOM expressions.

By default a named subform takes part in data binding and can be referenced using a SOM expression. This property allows a subform to be given a name but remain transparent to data binding and SOM expressions. The value of this property must be one of:

name

If the subform has a name it takes part in data binding and SOM expressions. Otherwise it does not.

none

The subform does not take part in data binding and SOM expressions, even if it has a name.

The setProperty child

An element that causes a property of the container to be copied from a value in the XFA Data DOM or from data returned by a web service.

For more information see "The setProperty element".

The subform child

A container element that describes a single subform capable of enclosing other containers.

For more information see "The subform element".

The subformSet child

An element that describes a set of related subform objects.

For more information see "The subformSet element".

The traversal property

An element that links its container to other objects in sequence.

For more information see "The traversal element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The validate property

A automation element that controls validation of user-supplied data.

For more information see "The validate element".

The variables property

An element to hold document variables.

For more information see "The variables element".

The w property

Width for layout purposes. A measurement value for `w` overrides any growth range allowed by the `minW` and `maxW` attributes. The absolute omission of this attribute or a value specified as an empty string indicates that the `minW` and `maxW` must be respected.

This attribute has no default. Setting this attribute to "-1" is an error.

The x property

X coordinate of the container's anchor point relative to the top-left corner of the parent container's nominal content region when placed with positioned layout. Defaults to 0.

The y property

Y coordinate of the container's anchor point relative to the top-left corner of the parent container's nominal content region when placed with positioned layout. Defaults to 0.

The subformSet element

An element that describes a set of related subform objects.

```
<subformSet

Properties:
    id="xml-id"
    name="xml-id"
    relation="ordered | unordered | choice"
    relevant="cdata"
    use="cdata"
    usehref="cdata"
>
    <bookend>  [0..1]
    <break>  [0..1]
    <desc>  [0..1]
    <extras>  [0..1]
    <occur>  [0..1]
    <overflow>  [0..1]

Children:
    <breakAfter>  [0..n]
    <breakBefore>  [0..n]
    <subform>  [0..n]
    <subformSet>  [0..n]
</subformSet>
```

The subformSet element is used within the following other elements:
area proto subform subformSet

The bookend property

An element controlling content that is inserted to "bookend" the contents of the parent object.

For more information see "The bookend element".

The break property

(DEPRECATED) An element that describes the constraints on moving to a new page or content area before or after rendering an object.

For more information see "The break element".

The breakAfter child

An element that controls actions to be taken after laying down the contents of the parent object.

For more information see "The breakAfter element".

The breakBefore child

An element that controls actions to be taken before laying down the contents of the parent object.

For more information see "The breakBefore element".

The desc property

An element to hold human-readable metadata.

For more information see "The desc element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The occur property

An element that describes the constraints over the number of allowable instances for its enclosing container.

For more information see "The occur element".

The overflow property

An element that controls what happens when the parent subform or subform set overflows the current layout container.

For more information see "The overflow element".

The relation property

This property specifies the relationship among the members of the set.

`ordered`

The members are to be instantiated in the order in which they are declared in the template. This has the effect of potentially re-ordering the content to satisfy the document order of the template.

`unordered`

The members are to be instantiated in data order regardless of the order in which they are declared. This has the effect of potentially re-ordering the set to satisfy the ordering of the content.

`choice`

The members are exclusive of each other, and only one member may be instantiated.The determination of which member to instantiate is based upon the data.

The relevant property

Specifies the views for which the enclosing object is relevant. The views themselves are specified in the config object.

Views are supplied as a space-separated list of viewnames: `relevant="[+|-]viewname [[+|-]viewname[...]]"`. A token of the form *viewname* or +*viewname* indicates the enclosing element should be included in that particular view. A token of the form `-viewname` indicates the element should be excluded from that particular view.

If a container is excluded, it is not considered in the data binding process.

See also Concealing Containers Depending on View and Config Specification.

The subform child

A container element that describes a single subform capable of enclosing other containers.

For more information see "The subform element".

The subformSet child

An element that describes a set of related subform objects.

For more information see "The subformSet element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The subjectDN element

An element that contains a key-value pair used to specify the Subject Distinguished Name (DN) that must be present within the certificate for it to be acceptable for signing.

```
<subjectDN

Properties:
    delimiter=", | cdata"
    id="xml-id"
    name="xml-id"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</subjectDN>
```

The subjectDN element is used within the following other elements:
proto subjectDNs

Content

Sets of key-value pairs separated by the delimiter character. Each key-value pair consists of optional whitespace, followed by a key string, followed by an equals sign (=), followed by the value (which may include whitespace). All but the last key-value pair must be delimited by the specified delimiter character.

The order of key-value pairs is not significant.

The delimiter property

The delimiter character used to separate key-value pairs. If the attribute is omitted or empty the delimiter defaults to comma (,).

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The subjectDNs element

An element that contains the collection of key-value pairs used to specify the Subject Distinguished Name (DN) that must be present within the certificate for it to be acceptable for signing.

```
<subjectDNs

Properties:
    id="xml-id"
    type="optional | required"
    use="cdata"
    usehref="cdata"
>

Children:
    <subjectDN>  [0..n]
</subjectDNs>
```

The subjectDNs element is used within the following other elements:
certificates proto

The certificate must contain all the attributes specified in the dictionary. It may also contain additional attributes.

The id property

A unique identifier that may be used to identify this element as a target.

The subjectDN child

An element that contains a key-value pair used to specify the Subject Distinguished Name (DN) that must be present within the certificate for it to be acceptable for signing.

For more information see "The subjectDN element".

The type property

Specifies whether the values provided in the element should be treated as a restrictive or non-restrictive set.

optional

The values provided in the element are optional seed values from which the XFA processing application may choose. The XFA processing application may also supply its own value.

required

The values provided in the element are seed values from which the XFA processing application must choose.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The submit element

An element that describes how to submit data to a host, using an HTTP POST operation.

```
<submit

Properties:
    embedPDF="0 | 1"
    format="xdp | xfd | formdata | xml |
            pdf | urlencoded"
    id="xml-id"
    target="cdata"
    textEncoding="cdata"
    use="cdata"
    usehref="cdata"
    xdpContent="cdata"
>
    <encrypt>  [0..1]

Children:
    <signData>  [0..n]
</submit>
```

The submit element is used within the following other elements:
event proto

When an event containing a submit element is triggered, several factors influence whether the submission occurs, as described in "Submitting Data and Other Form Content to a Server".

The embedPDF property

`embedPDF` specifies whether PDF is embedded in the submitted content or is represented as an external reference. This property is relevant only in following circumstances:

- Submitting event is part of a form included in or containing a PDF file.
- Format used to organize the data is XDP, as determined by the `format` property.
- XDP content being submitted includes PDF and/or XFDF, as determined by the `xdpContent` property.

0

The associated PDF document is not embedded in the XDP PDF packet; rather, a URI is optionally provided. The URI must resolve to a PDF resource of MIME type pdf. The URI is the value of the `href` attribute in the XDP PDF packet. The URI may be obtained from the XFDF F-key path, which is relative to the system on which the original PDF file was created. If the URI is unavailable, neither the PDF itself nor a URI is included in the PDF packet in the submitted XDP.

1

A copy of the associated PDF document is embedded in the submitted XDP. If the XFA application is capable of updating the PDF (for example, by adding annotations), the updated PDF is included in the PDF packet in the submitted XDP.

The encrypt property

An element that controls encryption of barcode or submit data.

For more information see "The encrypt element".

The format property

Determines the format in which the data will be submitted.

xdp

The data is packaged in XDP format, as described in XDP Specification.

formdata

The data is translated and packaged into an URL-encoded format which emulates certain legacy software. The use of this format is deprecated; use `urlencoded` for new applications.

pdf

The data is packaged in PDF format as described in the *PDF Reference* [PDF].

urlencoded

The data is packaged in URL-encoded format as described in *Uniform Resource Locators (URL)* [RFC1738]. However contrary to the recommendation of that specification, the `textEncoding` property is used to determine how the text is expressed before it is URL-encoded.

xfd

The data is packaged in XFD format, as described in [XFDF].

xml

The data is packaged in XML format as described in the *XML Specification version 1.0* [XML1.0]. The schema is determined according to the same rules used for data unloading, as described in Unload Processing.

The id property

A unique identifier that may be used to identify this element as a target.

The signData child

An element controlling an XML digital signature.

For more information see "The signData element".

The target property

The URL to which the data will be submitted. Omission of this attribute implies the XFA processing application obtains the URI using a product specific technique, such as accessing product-specific information in the config object.

The textEncoding property

The encoding of text content in the referenced document.

Note that the value of this property is case-insensitive. For that reason it is defined in the schema as cdata rather than as a list of XML keywords. However the value must match one of the following keywords in a case-insensitive manner.

none

No special encoding is specified. The characters are encoded using the ambient encoding for the operating system.

ISO-8859-1

The characters are encoded using ISO-8859-1 [ISO-8859-1], also known as Latin-1.

ISO-8859-2

The characters are encoded using ISO-8859-2 [ISO-8859-2].

ISO-8859-7

The characters are encoded using ISO-8859-7 [ISO-8859-7].

Shift-JIS

The characters are encoded using JIS X 0208, more commonly known as Shift-JIS [Shift-JIS].

KSC-5601

The characters are encoded using the *Code for Information Interchange (Hangul and Hanja)* [KSC5601].

Big-Five

The characters are encoded using Traditional Chinese (Big-Five). **Note:** there is no official standard for Big-Five and several variants are in use. XFA uses the variant implemented by Microsoft as code page 950 [Code-Page-950].

GB-2312

The characters are encoded using Simplified Chinese [GB2312].

UTF-8

The characters are encoded using Unicode code points as defined by [Unicode-3.2], and UTF-8 serialization as defined by *ISO/IEC 10646* [ISO-10646].

UTF-16

The characters are encoded using Unicode code points as defined by [Unicode-3.2], and UTF-16 serialization as defined by *ISO/IEC 10646* [ISO-10646].

UCS-2

The characters are encoded using Unicode code points as defined by [Unicode-3.2], and UCS-2 serialization as defined by *ISO/IEC 10646* [ISO-10646].

fontSpecific

The characters are encoded in a font-specific way. Each character is represented by one 8-bit byte.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The xdpContent property

Controls what subset of the data is submitted. This property is used only when the `format` property is `xdp`.

`datasets pdf xfdf`

Elements with the tags datasets, pdf, and xfdf are submitted to the host.

`tag1 tag2 ... tagN`

Elements with tags matching any of the specified tags are submitted to the host.

`*`

All data elements are submitted to the host.

The template element

An element that describes a template. One such element exists for each template and all other elements described in this specification are descendants of the template element.

```
<template

Properties:
    baseProfile="full | interactiveForms"
>
    <extras>  [0..1]

Children:
    <subform>  [0..n]
</template>
```

This element may contain an `originalXFAVersion` processing instruction. This processing instruction exists to allow a template to be upgraded from an earlier (pre-XFA 2.4) schema to XFA 2.4 or later without having to manually rewrite the scripts to accomodate changes in the event model. The parameters of the processing instruction are interpreted as follows:

`First parameter (mandatory):`

An XFA namespace URI identifying the XFA version for which the template was originally generated. If this corresponds to XFA 2.4 or later then the 2.4 event model is used regardless of any additional parameter.

`Additional parameter (optional):`

Provided the first parameter identified an XFA version of 2.3 or earlier, if there is an additional parameter with the value `LegacyEventModel:1` then the old pre-2.4 event model is used. Otherwise the XFA 2.4 event model is used.

The baseProfile property

Starting with XFA 2.5 subsets of the XFA grammar may be defined for particular special purposes. This attribute, if present, identifies the subset of the XFA template grammar for which the template was created. Programs that edit templates use this attribute to tell them what portion of the template grammar they are allowed to use.

`full`

The full XFA template grammar is allowed.

`interactiveForms`

The template grammar is restricted to the XFAF (XFA Foreground) subset.

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The subform child

A container element that describes a single subform capable of enclosing other containers.

For more information see "The subform element".

The text element

A content element that describes a single unit of data content representing a plain textual value.

```
<text

Properties:
    id="xml-id"
    maxChars="0 | integer"
    name="xml-id"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</text>
```

The text element is used within the following other elements:
desc exObject extras items message proto value variables

Content

This element may contain text data which is simple XML PCDATA or it may contain rich text. It may also be empty.

If the content is rich text it must be contained in an aggregating element such as body. The aggregating element, as well as its content, must belong to the XHTML namespace. Only a subset of XHTML markup is supported. The mechanism and its limitations are fully described in Rich Text Reference.

When no data content is provided, the data content may be interpreted as representing a null value. This behavior is dependent upon the context of where the data content is used. For instance, a field may interpret empty data content as null based upon the associated `nullType` property in the data description.

The id property

A unique identifier that may be used to identify this element as a target.

The maxChars property

This property specifies the maximum (inclusive) number of characters that this text value is permitted to enclose. The absolute omission of this property, or a value specified as an empty string indicates that there is no maximum.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The textEdit element

A user interface element that encloses a widget intended to aid in the manipulation of textual content.

```
<textEdit

Properties:
    allowRichText="0 | 1"
    hScrollPolicy="auto | on | off"
    id="xml-id"
    multiLine="1 | 0"
    use="cdata"
    usehref="cdata"
    vScrollPolicy="auto | on | off"
>
    <border>  [0..1]
    <comb>  [0..1]
    <extras>  [0..1]
    <margin>  [0..1]
</textEdit>
```

The textEdit element is used within the following other elements:
proto ui

The allowRichText property

Specifies whether the text may include styling (also known as rich text). The supported types of styling are described in the narrative section RichText.

Note: the allowRichText attribute informs the XFA application whether or not to present styling controls in the UI; it does not limit the user's ability to type plain text which might be interpreted by some down-stream application as styling. For instance, the user could type `<b>hello</b>` regardless of the setting of the property.

The value of this property must be one of the following:

0

Text styling is not allowed. This is the default when the `textEdit` element does not contain an `exData` element.

1

Text styling is allowed. This is the default when the `textEdit` element does contain an `exData` element.

The border property

A box model element that describes the border surrounding an object.

For more information see "The border element".

The comb property

An element that causes a field to be presented with vertical lines between the character positions.

For more information see "The comb element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The hScrollPolicy property

Controls the scrollability of the field in the horizontal direction.

auto

If the field is single-line it scrolls horizontally. Multi-line fields do not scroll horizontally.

on

A horizontal scroll bar is always displayed whether or not the input overflows the boundaries of the field. The field is scrollable regardless of whether it is a single-line or multi-line field.

off

The user is not allowed to enter characters beyond what can physically fit in the field width. This applies to typing and pasting from the clipboard. However data which is merged into the field from the Data DOM is not restricted. If the data exceeds the field size the user may not be able to view all of it.

The id property

A unique identifier that may be used to identify this element as a target.

The margin property

A box model element that specifies one or more insets for an object.

For more information see "The margin element".

The multiLine property

Specifies whether the text may span multiple lines.

1

The text may span multiple lines. This is the default when the textEdit element is contained within a draw element.

0

The text is limited to a single line. This is the default when the textEdit element is contained within a field element.

This property is provided for the benefit of clients (such as HTML browsers) that have two types of text edit widgets.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The vScrollPolicy property

Controls the scrollability of the field in the vertical direction.

`auto`

If the field is multi-line it scrolls vertically, displaying a vertical scroll bar when necessary. Single-line fields do not scroll vertically.

`on`

A vertical scroll bar is always displayed whether or not the input overflows the boundaries of the field. The field is scrollable regardless of whether it is a single-line or multi-line field.

`off`

The user is not allowed to enter characters beyond what can physically fit in the field height. This applies to typing and pasting from the clipboard. However data which is merged into the field from the Data DOM is not restricted. If the data exceeds the field size the user may not be able to view all of it.

Controls the scrollability of the field in the vertical direction.

`auto`

If the field is multi-line it scrolls vertically, displaying a vertical scroll bar when necessary, up to the limit set by maxChars. Single-line fields do not scroll vertically.

`on`

A vertical scroll bar is always displayed whether or not the input overflows the boundaries of the field. The field is scrollable to the limit set by maxChars regardless of whether it is a single-line or multi-line field.

`off`

The user is not allowed to enter characters beyond what can physically fit in the field height and is within maxChars. This applies to typing and pasting from the clipboard. However data which is merged into the field from the Data DOM is not restricted. If the data exceeds the field size the user may not be able to view all of it.

The time element

A content element that describes a single unit of data content representing a time value.

```
<time

Properties:
     id="xml-id"
     name="xml-id"
     use="cdata"
     usehref="cdata"
>
     ...pcdata...
</time>
```

The time element is used within the following other elements:
desc exObject extras items proto value variables

XFA time values conform to a subset of [ISO-8601]. This element is intended to hold only the time portion of an ISO-8601 date/time value, and any date information will be truncated. For instance, a time element enclosing the value `20010326T0630`, meaning 6:30am on March 26th 2001, will truncate the date and hold the value of `0630`, resulting in a value of 6:30am.

Content

This element may enclose time data which is a subset of [ISO-8601] as specified in Canonical Format Reference.

When no content is present, the content shall be interpreted as representing a null value, irrespective of the value of the associtated `nullType` property in the data description.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The timeStamp element

An element that controls the time-stamping of a signature.

```
<timeStamp

Properties:
    id="xml-id"
    server="cdata"
    type="optional | required"
    use="cdata"
    usehref="cdata"
>
</timeStamp>
```

The timeStamp element is used within the following other elements:
filter proto

This element is only meaningful when it is the child of a `signature` element. Otherwise it is ignored.

The id property

A unique identifier that may be used to identify this element as a target.

The server property

The URI of a server providing a time stamp that is compliant with [RFC 3161].

The type property

Indicates whether the time stamp is required or not.

`optional`

The time stamp is optional.

`required`

The signature must have a time stamp.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The toolTip element

An element that supplies text for a tool tip. This element is ignored by non-interactive applications.

```
<toolTip

Properties:
    id="xml-id"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</toolTip>
```

The toolTip element is used within the following other elements:
assist proto

Content

This property supplies text that is intended to be displayed by an interactive application when the cursor hovers over the associated field.

The id property

A unique identifier that may be used to identify this element as a target.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The traversal element

An element that links its container to other objects in sequence.

```
<traversal

Properties:
    id="xml-id"
    use="cdata"
    usehref="cdata"
>
    <extras>   [0..1]

Children:
    <traverse>   [0..n]
</traversal>
```

The traversal element is used within the following other elements:
draw exclGroup field proto subform

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The traverse child

An element that declares a single link from its container to another object in a unidirectional chain of links.

For more information see "The traverse element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The traverse element

An element that declares a single link from its container to another object in a unidirectional chain of links.

```
<traverse

Properties:
    id="xml-id"
    operation="next | up | down | left |
               right | back | first"
    ref="cdata"
    use="cdata"
    usehref="cdata"
>
    <extras>  [0..1]
    <script>  [0..1]
</traverse>
```

The traverse element is used within the following other elements:
proto traversal

The chain of links is not constrained to contain only one-to-one links. There may be many-to-one links, that is, traverse elements in multiple containers may point to the same destination. For this reason traversal chains may not be reversible, unless specifically designed to be so.

When any traversal is not specified, it defaults to geographical order, where the forward direction is defined as left-to-right top-to-bottom. This definition of forward direction is used regardless of the language component of the locale.

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The operation property

This property specifies the key-strokes or other situations that cause focus to switch to the container indicated by this element's `script` element or `ref` attribute.

`next`

Destination in any of the following circumstances:

- User presses the Tab key.
- User enters the final character in a fixed-width field.
- Speech tool finishes enunciating text for the container.

Defaults to left-to-right top-to-bottom order.

In order to serve the speech tool, the chain of `next` links may include boilerplate objects. Such objects cannot accept input focus. Therefore, when advancing focus to the next input widget, the XFA application continues traversing the chain until it reaches an object that does accept input focus. It is up to the template creator to ensure that the template does not present the XFA application with a non-terminating loop.

`back`

Destination when the user presses Shift-Tab on a PC, or the corresponding key on other platforms. Defaults to right-to-left bottom-to-top order.

`down`

Destination when the user presses the down-arrow key. Defaults to top-to-bottom order.

`first`

This property is applicable only when the container is a subform or subform set. The link points to the child container that gains focus when the container is entered. In effect, the container delegates focus via this link. If the container does not specify a "first" child container, the top left child container becomes by default the first to be traversed.

`left`

Destination when the user presses the left-arrow key. Defaults to right-to-left order.

`right`

Destination when the user presses the right-arrow key. Defaults to left-to-right order.

`up`

Destination when the user presses the up-arrow key. Defaults to bottom-to-top order.

The ref property

A SOM expression identifying the destination object. The expression must resolve to a valid layout node. If the `script` property is provided, this property is ignored.

The script property

An automation element that contains a script that provides a destination location. The script must resolve to a valid layout node. If the `script` element is provided, it takes precedence over the `ref` attribute. See also Sequencing: Tab Order and Speech Order.

For more information see "The script element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The ui element

A user-interface element that encloses the actual user interface widget element.

```
<ui

Properties:
    id="xml-id"
    use="cdata"
    usehref="cdata"
>
    <extras>  [0..1]
    <picture>  [0..1]

One-of properties:
    <barcode>  [0..1]
    <button>  [0..1]
    <checkButton>  [0..1]
    <choiceList>  [0..1]
    <dateTimeEdit>  [0..1]
    <defaultUi>  [0..1]
    <exObject>  [0..1]
    <imageEdit>  [0..1]
    <numericEdit>  [0..1]
    <passwordEdit>  [0..1]
    <signature>  [0..1]
    <textEdit>  [0..1]
</ui>
```

The ui element is used within the following other elements:
draw field proto

This element has a set of one-of properties. The choice of one-of property determines the type of widget displayed. For example, if the `button` property is included the content will be displayed as a button widget. This determines both the appearance of the content and the interaction with it. Including the `defaultUi` property delegates the decision about what widget to use to the XFA application.

Note that the presence of this element does not imply that its container accepts input from the user. The container could be a draw element, or it could be a field element with its `access` property set to `nonInteractive`. In either of these cases the ui element merely controls the manner in which the content is presented.

The barcode property

An element that represents a barcode.

For more information see "The barcode element".

The button property

A user interface element that describes a push-button widget.

For more information see "The button element".

The checkButton property

A user interface element that describes either a checkbox or radio-button widget.

For more information see "The checkButton element".

The choiceList property

A user interface element that describes a widget presenting a list of options. The list of options is specified by one or more sibling `items` elements.

For more information see "The choiceList element".

The dateTimeEdit property

A user interface element describing a widget intended to aid in the selection of date and/or time.

For more information see "The dateTimeEdit element".

The defaultUi property

An element for widgets whose depiction is delegated to the XFA application.

For more information see "The defaultUi element".

The exObject property

An element that describes a single program or implementation-dependent foreign object.

For more information see "The exObject element".

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The imageEdit property

A user interface element that encloses a widget intended to aid in the manipulation of image content.

For more information see "The imageEdit element".

The numericEdit property

A user interface element that describes a widget intended to aid in the manipulation of numeric content.

For more information see "The numericEdit element".

The passwordEdit property

A user interface element that describes a widget intended to aid in the manipulation of password content. Typically the user-interface will obscure any visual representation of the content.

For more information see "The passwordEdit element".

The picture property

A rendering element that describes input parsing and output formatting information.

For more information see "The picture element".

The signature property

A user interface element that describes a widget intended to allow a user to sign a completed form, making it a document of record.

For more information see "The signature element".

The textEdit property

A user interface element that encloses a widget intended to aid in the manipulation of textual content.

For more information see "The textEdit element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The validate element

A automation element that controls validation of user-supplied data.

```
<validate

Properties:
    formatTest="warning | disabled | error"
    id="xml-id"
    nullTest="disabled | warning | error"
    scriptTest="error | disabled | warning"
    use="cdata"
    usehref="cdata"
>
    <extras>   [0..1]
    <message>   [0..1]
    <picture>   [0..1]
    <script>   [0..1]
</validate>
```

The validate element is used within the following other elements:
exclGroup field proto subform

The extras property

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The formatTest property

Controls validation against the display picture clause.

warning

Emit a message if the data cannot be coerced to fit the picture clause, but allow the user to proceed to the next field (default).

disabled

Do not perform this test.

error

Emit a message and refuse to accept data that cannot be coerced to fit the picture clause.

The id property

A unique identifier that may be used to identify this element as a target.

The message property

A automation element that holds one or more sub-elements containing messages used with validations and calculations.

For more information see "The message element".

The nullTest property

Controls whether the field can be left empty.

disabled

Do not perform this test (default). An empty field is perfectly acceptable.

error

Emit a message and refuse to accept an empty field.

warning

Emit a message if the field is empty, but allow the user to proceed to the next field.

The picture property

A rendering element that describes input parsing and output formatting information.

For more information see "The picture element".

The script property

An automation element that contains a script.

For more information see "The script element".

The scriptTest property

Controls validation by the enclosed script.

error

Emit a message and refuse to accept data that the script reports is erroneous (default).

disabled

Do not perform this test.

warning

Emit a message if the script reports the data is erroneous, but allow the user to proceed to the next field.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The value element

A content element that encloses a single unit of data content.

```
<value

Properties:
    id="xml-id"
    override="0 | 1"
    relevant="cdata"
    use="cdata"
    usehref="cdata"
>

One-of properties:
    <arc>  [0..1]
    <boolean>  [0..1]
    <date>  [0..1]
    <dateTime>  [0..1]
    <decimal>  [0..1]
    <exData>  [0..1]
    <float>  [0..1]
    <image>  [0..1]
    <integer>  [0..1]
    <line>  [0..1]
    <rectangle>  [0..1]
    <text>  [0..1]
    <time>  [0..1]
</value>
```

The value element is used within the following other elements:
caption draw field proto

The arc property

A curve that can be used for describing either an arc or an ellipse.

For more information see "The arc element".

The boolean property

A content element describing single unit of data content representing a Boolean logical value.

For more information see "The boolean element".

The date property

A content element that describes a single unit of data content representing a date.

For more information see "The date element".

The dateTime property

A content element that describes a single unit of data content representing a date and time value.

For more information see "The dateTime element".

The decimal property

A content type element that describes a single unit of data content representing a number with a fixed number of digits after the decimal.

For more information see "The decimal element".

The exData property

A content element that describes a single unit of data of a foreign datatype.

For more information see "The exData element".

The float property

A content element that describes a single unit of data content representing a floating point value.

For more information see "The float element".

The id property

A unique identifier that may be used to identify this element as a target.

The image property

A content element that describes a single image.

For more information see "The image element".

The integer property

A content element that describes a single unit of data content representing an integer value.

For more information see "The integer element".

The line property

A content element that describes a single rendered line.

For more information see "The line element".

The override property

This property specifies whether the value resulted from an override to a calculation or validation.

`0`

The value does not represent a value supplied as an override to a calculation or validation constraint on the value.

`1`

The value does represent a value supplied as an override to a calculation or validation constraint on the value.

The rectangle property

A content element that describes a single rendered rectangle.

For more information see "The rectangle element".

The relevant property

Specifies the views for which the enclosing object is relevant. The views themselves are specified in the config object.

Views are supplied as a space-separated list of viewnames: `relevant="[+|-]viewname [[+|-]viewname[...]]"`. A token of the form *viewname* or *+viewname* indicates the enclosing element should be included in that particular view. A token of the form *-viewname* indicates the element should be excluded from that particular view.

If a container is excluded, it is not considered in the data binding process.

See also Concealing Containers Depending on View and Config Specification.

The text property

A content element that describes a single unit of data content representing a plain textual value.

For more information see "The text element".

The time property

A content element that describes a single unit of data content representing a time value.

For more information see "The time element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The variables element

An element to hold document variables.

```
<variables

Properties:
    id="xml-id"
    use="cdata"
    usehref="cdata"
>

Children:
    <boolean>  [0..n]
    <date>  [0..n]
    <dateTime>  [0..n]
    <decimal>  [0..n]
    <exData>  [0..n]
    <float>  [0..n]
    <image>  [0..n]
    <integer>  [0..n]
    <manifest>  [0..n]
    <script>  [0..n]
    <text>  [0..n]
    <time>  [0..n]
</variables>
```

The variables element is used within the following other elements:
proto subform

Document variables are used to hold boilerplate which may be inserted conditionally under control of a script, for example terms and conditions of a purchase agreement. Placing the boilerplate content into a `variables` element makes it accessible to scripts via the usual mechanism of SOM expressions.

The `variables` element can hold any number of separate data items. The data items can be any kind of data. Each data item bears its own `name` attribute so they are individually addressable by scripts. In SOM expressions, data items are directly under the subform. For example, if a subform is declared as:

```
<subform name="w">
    <subform name="x">
    <variables>
        <integer name="foo">1234</integer>
        <float name="bar">1.234</float>
    </variables>
    <field name="y">...</field>
    </subform>
</subform>
```

then in the context of the subform named `w`, the variables are addressed by the SOM expressions `x.foo` and `x.bar`, while the field is addressed as `x.y`.

It is conventional to place a single `variables` element in the root subform to hold all document variables, but this is only a convention. Any subform can hold a `variable` element.

The boolean child

A content element describing single unit of data content representing a Boolean logical value.

For more information see "The boolean element".

The date child

A content element that describes a single unit of data content representing a date.

For more information see "The date element".

The dateTime child

A content element that describes a single unit of data content representing a date and time value.

For more information see "The dateTime element".

The decimal child

A content type element that describes a single unit of data content representing a number with a fixed number of digits after the decimal.

For more information see "The decimal element".

The exData child

A content element that describes a single unit of data of a foreign datatype.

For more information see "The exData element".

The float child

A content element that describes a single unit of data content representing a floating point value.

For more information see "The float element".

The id property

A unique identifier that may be used to identify this element as a target.

The image child

A content element that describes a single image.

For more information see "The image element".

The integer child

A content element that describes a single unit of data content representing an integer value.

For more information see "The integer element".

The manifest child

An element that contains a list of references to all the nodes that are included in a node collection.

For more information see "The manifest element".

The script child

An automation element that contains a script.

For more information see "The script element".

The text child

A content element that describes a single unit of data content representing a plain textual value.

For more information see "The text element".

The time child

A content element that describes a single unit of data content representing a time value.

For more information see "The time element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

The object used as a prototype does not need to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

18 Config Specification

The Configuration Data Object Model (DOM) provides a centralized mechanism for specifying configuration options for XFA applications. The configuration information may be specified in XML format and passively loaded at the start of processing. However the Configuration DOM also supports a scripting interface which allows user-supplied scripts to examine and modify the configuration settings.

An XML document containing XFA configuration options is referred to in this specification as an "XFA configuration document". A file containing such a document is often referred to as an "XCI file". "XCI" stands for XFA Configuration Information. Often the filename suffix ".xci" is used for XCI files, but this is merely a convention.

Background

In order to use the configuration options it is necessary to understand where and when each option has its effect. Typically XFA applications pass through a number of stages. For example, an interactive form-filling application passes through the following stages:

1. Load the configuration options into the Configuration DOM.
2. Load the template into the Template DOM.
3. Obtain existing user data from the host in the form of an XML document.
4. Preprocess existing user data via an XSLT interpreter.
5. Load the preprocessed data into the XML Data DOM.
6. Load the data from the XML Data DOM into the XFA Data DOM.
7. Merge the existing data with the template.
8. Layout the merged template plus data onto pages.
9. Present the laid-out pages to the user.
10. Accept and validate input from the user, updating the XFA Data DOM.
11. Unload the updated user data from the XFA Data DOM to the XML Data DOM.
12. Create a new XML document containing the updated user data.
13. Postprocess the new XML document via an XSLT interpreter.
14. Send the postprocessed XML document to the host.

Notionally, a separate processor handles each phase, however all of the processors rely on the Configuration DOM to supply them with configuration information. This centralized mechanism makes it possible to keep all of the configuration information in a single XML document for easy maintenance, and to supply a uniform scripting interface for all configuration options.

The actual location of the configuration document (or documents) is variable. XFA provides a convenient mechanism for packaging XML data ("XDP Specification" on page 873), which might be used to bundle configuration options with the template and other relevant material. However, XFA applications are free to use other mechanisms, such as environment variables and command-line parameters, instead of or in addition to XDP. This specification is limited to the setting of configuration options via XML documents and scripts.

The Configuration Data Object Model

The Configuration Data Object Model (Configuration DOM) encapsulates the XFA configuration information and provides standard interfaces to it. The Configuration DOM contains data objects organized in a tree structure. The configuration tree is itself a branch within a larger XFA tree.

The "config" element and the standard contents shown above must belong to the namespace "http://www.xfa.org/schema/xci/1.0/" which is known as the XCI namespace. Custom elements may belong to the XCI namespace but are not required to.

Defaults

Conceptually the Configuration DOM exists before any configuration document is loaded. The Configuration DOM must initialize all option values to their default values at startup. Options that require a keyword value must be initialized to the values shown in the corresponding elements of the above skeleton. Options that do not require keyword values must be initialized to the empty string.

When the XFA application loads the configuration DOM from a configuration document, if a particular element is not present in the configuration document, the associated option must retain its preexisting value.

Most of the default values cause the associated option to adopt safe behavior. The only exceptions are the "uri" elements which supply the locations for the data and template documents; the XFA application may declare a fatal error if either or both of these options is set to the empty string when the time comes to load the associated document. As an alternative it may fall back on some other mechanism to locate the required document.

Scripting Interface

The Configuration DOM is part of a larger tree that holds all exposed XFA objects. The single large tree makes it possible to refer to XFA objects using a unified format known as a Scripting Object Model (SOM) expression. The grammar of SOM expressions is described in "Scripting Object Model" on page 73. Briefly, an expression consists of a sequence of node names separated by periods ("." characters). Starting from some point in the XFA tree, each successive name identifies which child of the current node to descend to. The root of the Configuration DOM must be a child of the root `xfa` node. Hence, the `config` node itself is `xfa.config`. Assuming the application Name is "myapp", the node representing the `attributes` element would be referenced by the SOM expression `xfa.config.myapp.data.attributes`.

In addition, SOM expressions must recognize the short-form "$config" as equivalent to `xfa.config`. Thus for example the "attributes" element mentioned in the preceding paragraph could be referenced as `$config.myapp.data.attributes`.

The scripting interface must make it possible for user-supplied scripts to inspect and modify the contents of the Configuration DOM. It is not responsible for guaranteeing that modifying the value of a particular option will have any effect. In particular, many options have an effect only during a particular phase of

processing. If a script changes the value of an option after the option has already had its effect the change will, at best, accomplish nothing. The script writer must ensure that each assignment is done before the phase(s) to which the option applies. It is recommended that XFA applications and scripts set any required non-default option values in the Configuration DOM as early in the processing cycle as possible.

Some options have effects during more than one phase of processing. This specification does not guarantee that it is safe to alter a value in the Configuration DOM during or after the processing phase in which the value is first used. When and under what circumstances it is safe to alter the value of an option is implementation-defined. It is recommended that XFA applications and scripts refrain from altering the value of each option in the Configuration DOM once it has been set.

Config Element Reference

The adjustData element

This option controls whether the XFA application re-normalizes the data after merging.

```
<adjustData

Properties:
    desc="cdata"
>
    ...pcdata...
</adjustData>
```

The adjustData element is used within the following other elements:
data

Re-normalizing is coercing the data in the XML Data DOM to fit the structure of the template. The coercion is carried out near the end of the data binding phase.

Content

The content must be one of the following:

0

Coercion of the data is forbidden.

1

Coercion of the data is required.

When the value of `adjustData` is 1, the XFA application rearranges the XML Data DOM to fit the hierarchical structure of the template. When the value is `0`, the XFA application does not alter the XML Data DOM except to introduce nodes that the template explicitly references. This option takes effect during data binding. See Re-Normalization.

The desc property

An attribute to hold human-readable metadata.

The agent element

This element is a container for all options for one particular XFA application.

```
<agent

Properties:
    desc="cdata"
    name="xml-id"
>
    <common>   [0..1]
</agent>
```

The agent element is used within the following other elements:
config

The common property

This element is a container for options used by many or all XFA applications.

For more information see "The common element".

The desc property

An attribute to hold human-readable metadata.

The name property

The value for this attribute, hence the name of the node in the Configuration DOM, is **implementation-defined**. This allows for multiple applications to share a configuration document without interfering with each other. To ensure that there are no name collisions the value used should be a unique URI based on a registered domain. Such URIs may contain characters that are not supported in names within XFA-SOM expressions, hence the application may have to carry out additional processing, such as searching through all sibling agent nodes, to find its own configuration information.

The attributes element

This option controls whether the data-loader loads attributes from the XML Data DOM into the XFA Data DOM.

```
<attributes

Properties:
    desc="cdata"
>
    ...pcdata...
</attributes>
```

The attributes element is used within the following other elements:
data

This option takes effect during the data-load phase. If this option prevents attributes from being loaded into the XFA Data DOM, then during the data-unload phase the XFA application obtains the attributes and their values from the XML Data DOM and inserts them into the output XML document. See the Basic Data Binding to Produce the XFA Form DOM for more detailed information about the algorithms used.

For more information about this element, see Extended Mapping Rules.

Content

The content must be one of the following:

preserve

The XFA application loads attributes from the XML Data DOM into the XFA Data DOM as described in Basic Data Binding to Produce the XFA Form DOM.

delegate

The XFA application determines whether or not to load attributes.

ignore

The XFA application does not load attributes into the XFA Data DOM.

The desc property

An attribute to hold human-readable metadata.

The common element

This element is a container for options used by many or all XFA applications.

```
<common

Properties:
    desc="cdata"
>
    <data>  [0..1]
    <locale>  [0..1]
    <template>  [0..1]
</common>
```

The common element is used within the following other elements:
agent

The data property

This element is a container for those options which control the handling of user data (as opposed to templates or other XFA documents).

For more information see "The data element".

The desc property

An attribute to hold human-readable metadata.

The locale property

This option specifies a default locale.

For more information see "The locale element".

The template property

This option controls the location of the template file.

For more information see "The template element".

The config element

Outermost element for the configuration information.

```
<config

Properties:
    desc="cdata"
>

Children:
    <agent>  [0..n]
</config>
```

The agent child

This element is a container for all options for one particular XFA application.

For more information see "The agent element".

The desc property

An attribute to hold human-readable metadata.

The data element

This element is a container for those options which control the handling of user data (as opposed to templates or other XFA documents).

```
<data

Properties:
    desc="cdata"
>
    <adjustData>  [0..1]
    <attributes>  [0..1]
    <outputXSL>  [0..1]
    <range>  [0..1]
    <record>  [0..1]
    <startNode>  [0..1]
    <uri>  [0..1]
    <window>  [0..1]
    <xsl>  [0..1]

Children:
    <excludeNS>  [0..n]
    <transform>  [0..n]
</data>
```

The data element is used within the following other elements:
common

The adjustData property

This option controls whether the XFA application re-normalizes the data after merging.

For more information see "The adjustData element".

The attributes property

This option controls whether the data-loader loads attributes from the XML Data DOM into the XFA Data DOM.

For more information see "The attributes element".

The desc property

An attribute to hold human-readable metadata.

The excludeNS child

This option controls the exclusion of data in particular namespaces from the XFA Data DOM.

For more information see "The excludeNS element".

The outputXSL property

This element contains the elements that govern the postprocessing of the output XML document by an XSLT interpreter.

For more information see "The outputXSL element".

The range property

This option controls which records are processed.

For more information see "The range element".

The record property

This option controls the division of the document into records.

For more information see "The record element".

The startNode property

This option controls the subtree of the input document which is processed by the XFA application.

For more information see "The startNode element".

The transform child

A `transform` element nominates a set of input data elements to which its contained set of transformations apply.

For more information see "The transform element".

The uri property

This element is used to hold a URI.

For more information see "The uri element".

The window property

This option specifies the window size to use when performing incremental loads.

For more information see "The window element".

The xsl property

This option controls the preprocessing of user data, of the template, or of the device control information via an XSLT interpreter.

For more information see "The xsl element".

The debug element

This option controls whether the XFA application saves a copy of a preprocessed document after the XSLT interpreter has created it. It is intended for use in debugging the XSLT script.

```
<debug

Properties:
    desc="cdata"
>
    <uri>   [0..1]
</debug>
```

The debug element is used within the following other elements:
xsl

If the `uri` element contained within the `debug` element is empty or absent the XFA application **must not** save the preprocessed document. Hence the temp file containing the preprocessed document, if any, **must** be deleted. However if the `uri` element contained with the `debug` element is non-empty the XFA application **must**, upon exiting, leave behind an XML document at the specified URI containing the output of the preprocess phase.

This option takes effect during the data-load phase if it is contained in `data`, in the template-load phase if it is contained in `template`, and in the rendering phase if it is contained in `xdc`.

For more information about this element, see XSLT Transformations.

The desc property

An attribute to hold human-readable metadata.

The uri property

This element is used to hold a URI.

For more information see "The uri element".

The excludeNS element

This option controls the exclusion of data in particular namespaces from the XFA Data DOM.

```
<excludeNS

Properties:
    desc="cdata"
>
    ...pcdata...
</excludeNS>
```

The excludeNS element is used within the following other elements:
data

Elements and attributes with the specified namespace are not loaded from the XML Data DOM into the XFA Data DOM. The schema allows any number of `excludeNS` elements so any number of namespaces can be excluded. In addition namespaces associated with XML and XFA are always excluded. Namespace exclusion is described in detail in the section Extended Mapping Rules.

When an `excludeNS` element is present but empty it **must** have no effect.

This option takes effect during the data-load phase.

For more information about this element, see Extended Mapping Rules.

Content

The full URI (not the prefix) of a namespace to be excluded.

The desc property

An attribute to hold human-readable metadata.

The groupParent element

This option controls grouping of a set of related elements under a parent element.

```
<groupParent

Properties:
    desc="cdata"
>
    ...pcdata...
</groupParent>
```

The groupParent element is used within the following other elements:
transform

Sometimes XML data documents do not express the full structure of the data in markup. This option provides a way to restore structure to XML data documents in which the hierarchy has been flattened. It causes a contiguous group of related elements to be placed under a parent group node in the XFA Data DOM.

This option applies only to elements that are named in the `ref` attribute of the enclosing `transform` element.

There are a number of subtleties involved in the use of this option. See Extended Mapping Rules for more information.

Content

The name of the parent group node to insert.

The desc property

An attribute to hold human-readable metadata.

The ifEmpty element

This option controls the representation of empty elements in the XFA Data DOM.

```
<ifEmpty

Properties:
    desc="cdata"
>
    ...pcdata...
</ifEmpty>
```

The ifEmpty element is used within the following other elements:
transform

This option applies only to empty elements with tags that match the value of the `ref` attribute of its enclosing `transform` element.

This option takes effect during the data-load phase. When the content is `ignore` it also causes data from the XML Data DOM to be blended with data from the XFA Data DOM during the data-unload phase.

For more information about this element see Extended Mapping Rules.

Content

The content must be one of the following:

`dataValue`

The empty element is represented in the XFA Data DOM by a dataValue node.

`dataGroup`

The empty element is represented in the XFA Data DOM by a dataGroup node.

`ignore`

The empty element is omitted from the XFA Data DOM.

`remove`

The empty element is omitted from the XFA Data DOM and removed from the XML Data DOM. When the updated data is written out during the data-unload phase, the empty element will be omitted from the new XML document.

DataValue and dataGroup nodes are discussed in Extended Mapping Rules.

The desc property

An attribute to hold human-readable metadata.

The locale element

This option specifies a default locale.

```
<locale

Properties:
    desc="cdata"
>
    ...pcdata...
</locale>
```

The locale element is used within the following other elements:
common

Locale determines the currency symbol, the radix symbol, calendar, day and month names, and other information which is culture- or location-specific. This option supplies an ambient locale, overriding the ambient locale supplied by the host operating system (if any). Each individual subforms may assert its own locale, inherit an asserted locale from its parent, or use the ambient locale.

Content

The content must be a two-letter language code or a combined country code and language code as defined by *IETF RFC 1766* [RFC1766] and *ISO 639* [ISO-639-1] /*ISO 3166* [ISO-3166-1] specifications. This is the same set of locale names used by the `xml:lang` attribute defined in [XML1.0]. Note that three-letter combined codes are not supported by XFA. Combined codes must consist of a language code, followed by a hyphen or underscore character, followed by a country code, followed optionally by a modifier.

When this element is omitted or empty the locale supplied by the host operating system is used as the ambient locale. If the host operating system does not supply a locale then en_us (or equivalently en-us) is used.

XFA applications are not expected to support all possible locale values, especially as new country codes are issued from time to time. If the supplied code is anything other than en_us and it is not contained in the locale set packet, then it is implementation-defined whether the locale is supported or not. All XFA applications are required to support en_us.

The desc property

An attribute to hold human-readable metadata.

The nameAttr element

This option controls the renaming of nodes in the XFA Data DOM based upon the value of an attribute.

```
<nameAttr

Properties:
    desc="cdata"
>
    ...pcdata...
</nameAttr>
```

The nameAttr element is used within the following other elements:
transform

This option applies only to nodes corresponding to elements with tags that match the value of the `ref` attribute of its enclosing `transform` element.

This option takes effect during the data-load phase.

For more information about this element see the narrative description of this element at Extended Mapping Rules.

Content

The value of `nameAttr` is the name of an attribute. If the element in the XML document has an attribute with this name and the attribute's value is not the empty string (""), the XFA application renames the corresponding node in the XFA Data DOM as the value of the attribute. For example, if the configuration document contains the fragment

```
<transform ref="foo">   <nameAttr>bar</nameAttr> </transform>
```

and the following element is present in the input XML document

```
<foo bar="blort">some text</foo>
```

then the XFA application renames the corresponding node in the XFA Data DOM from "foo" to "blort".

The desc property

An attribute to hold human-readable metadata.

The outputXSL element

This element contains the elements that govern the postprocessing of the output XML document by an XSLT interpreter.

```
<outputXSL

Properties:
    desc="cdata"
>
    <uri>   [0..1]
</outputXSL>
```

The outputXSL element is used within the following other elements:
data

The `outputXSL` element can contain a `uri` element. If the `uri` element is present and non-empty the output XML document is passed through an XSLT interpreter, otherwise the output XML document is left alone. The URI which is the content of the `uri` element supplies the location of the XSLT stylesheet.

XSLT is defined by *XML Transformations (XSLT) Version 1.0* [XSLT].

For more information about this element see XSLT Transformations.

The desc property

An attribute to hold human-readable metadata.

The uri property

This element is used to hold a URI.

For more information see "The uri element".

The picture element

This option controls the formatting of updated data.

```
<picture

Properties:
    desc="cdata"
>
    ...pcdata...
</picture>
```

The picture element is used within the following other elements:
transform

This option applies only to nodes corresponding to elements with tags that match the value of the `ref` attribute of its enclosing `transform` element.

When the XFA Data DOM is updated, the new data for the node is validated against, and if it matches formatted in accordance with, the picture clause. This is in addition to any validation or calculation supplied by other means.

This option takes effect during the data update phase.

For more information about this element see Extended Mapping Rules.

Content

The content must be a picture clause. The format of the picture clause is defined in *XFA Picture Clause 2.0 Specification* Picture Clause Specification.

The desc property

An attribute to hold human-readable metadata.

The presence element

This option controls the inclusion of nodes in the XFA Data DOM.

```
<presence

Properties:
    desc="cdata"
>
    ...pcdata...
</presence>
```

The presence element is used within the following other elements:
transform

This option applies only to nodes corresponding to elements with tags that match the value of the `ref` attribute of its enclosing `transform` element.

This option takes effect during the data-load phase. When the content is `ignore` it also causes data from the XML Data DOM to be blended with data from the XFA Data DOM during the data-unload phase.

For more information about this element see Concealing Content Depending on View and Extended Mapping Rules.

Content

The content must be one of the following:

`preserve`

The node is included in the XFA Data DOM.

`dissolve`

The node itself is excluded from the XFA Data DOM but its children are promoted to become children of the dissolved node's parent.

`dissolveStructure`

The node itself is included, but its children and their descendents are excluded from the XFA Data DOM.

`ignore`

The node is excluded from the XFA Data DOM along with all of its descendents (that is, nodes corresponding to content of the element to which the node corresponds). However during the data-unload phase (when writing out a new XML document) the affected content is copied out from the XML Data DOM into the new document.

`remove`

The node and is descendents are excluded from the XFA Data DOM and in addition the corresponding nodes are removed from the XML Data DOM. The result of this is that during the data-unloading phase a new XML document is produced which lacks the removed data.

The desc property

An attribute to hold human-readable metadata.

The range element

This option controls which records are processed.

```
<range

Properties:
    desc="cdata"
>
    ...pcdata...
</range>
```

The range element is used within the following other elements:
data

When range has non-null content the XFA application loads only those records indicated into the XFA Data DOM. During the data-unload phase the skipped records are restored to the output XML document by copying them from the XML Data DOM. When the content of range is the null string, or the range element is omitted, the range defaults to the entire document. Note that this is the reverse of the usual default behavior - the range element names the records which are to be processed, yet when no records are named the result is not to exclude all records but to include all records.

This option affects processing during data load and data unload.

For more information about this element see Extended Mapping Rules.

Content

When it is not the null string, the value of range is a comma-separated list of one or more record numbers and/or record number ranges. A record number is a non-negative decimal integer, where 0 (zero) indicates the first record. A record number range is a record number, followed by a - character, followed by another record number which is numerically equal to or greater than the other record number. Record number ranges and record numbers are allowed to overlap.

For example, the following

```
<range>3-5,9,4,5-6</range>
```

causes records 3, 4, 5, 6 and 9 to be processed and all other records to be ignored.

The desc property

An attribute to hold human-readable metadata.

The record element

This option controls the division of the document into records.

```
<record

Properties:
    desc="cdata"
>
    ...pcdata...
</record>
```

The record element is used within the following other elements:
data

Records are processed sequentially, almost as separate documents. For example, in a form-printing application, the cycle of operations for each record is:

- preprocess the record via an XSLT interpreter
- load the preprocessed record into the XML Data DOM
- load the record data from the XML Data DOM into the XFA Data DOM
- merge the record data with the template
- lay out the merged template and record data upon the page
- render the layed-out template and record data into printer language
- send the rendering to the printer

A consequence of this cyclical processing is that a new merge operation and a new layout operation are performed for each new record. Because of this, the resulting printed document starts each record at the top of a new page, even though nothing in the template specifies that this should happen.

Data that is outside any record is still available via SOM expression in scripts and in data references (dataRef elements). See Basic Data Handling for more information about records. See XFA Template Specification for more information about dataRef elements.

This option affects processing during every phase.

For more information about this element see Extended Mapping Rules.

Content

The value of `record` determines the granularity at which the document is divided into records. The value can be empty or a non-negative decimal integer or a tag name.

If the `record` element is absent or empty the entire document is treated as a single record.

If the value of `record` is an integer, it specifies the level in the tree at which the XFA application treats each node as the root of a record. `0` represents the root of the whole XML Data DOM. For example, if the value is `2`, each element which is two levels in from the outermost element is considered as enclosing a record. Content that is at a higher level is considered as outside any record.

If the value of `record` is not an integer, it is interpreted as a tag name. The first element in the XML data document with a tag matching the value of `record` determines the level of a record within the tree. The

XFA application treats nodes in the XFA Data DOM that correspond to the same level and have a name matching the value of `record` as root nodes of records. All data that is not descended from such a node are treated as outside any record.

The desc property

An attribute to hold human-readable metadata.

The relevant element

This option specifies what views of the template are to be included.

```
<relevant

Properties:
    desc="cdata"
>
    ...pcdata...
</relevant>
```

The relevant element is used within the following other elements:
template

Various elements within the template may have `relevant` attributes. When present on a template element this attribute means that the element is to be loaded only in some views and ignored in others. This option names the active view or views. For example if the value of this element is invoice then any portion of the template marked with a `relevant` attribute which includes the token invoice is loaded.

Content

The content may be a single token or a space-separated list of tokens. Each token names a view that is active simultaneously with the others in the list. Note that unlike the `relevant` attribute there is no support for a - prefix; views can only be included by this element, not excluded.

The desc property

An attribute to hold human-readable metadata.

The rename element

This option controls the renaming of nodes in the XFA Data DOM.

```
<rename

Properties:
    desc="cdata"
>
    ...pcdata...
</rename>
```

The rename element is used within the following other elements:
transform

This option applies only to nodes corresponding to elements with tags that match the value of the `ref` attribute of its enclosing `transform` element.

If the `rename` element is absent or empty the name of the node in the XFA Data DOM is taken from the tag name. If the value is non-empty the node in the XFA Data DOM corresponding to the element is renamed to the value of the `rename` element. For example, an XFA configuration document includes the fragment

```
<transform ref="foo">   <rename>bar</rename> </transform>
```

and as a result for each node in the XFA Data DOM corresponding to an element named "foo", the XFA application renames the node to "bar". This does not affect the XML Data DOM.

This option takes effect during the data-load phase.

For more information, see Extended Mapping Rules.

Content

The contents must be a valid XFA node name. See XFA Names for a discussion of XFA node names.

The desc property

An attribute to hold human-readable metadata.

The startNode element

This option controls the subtree of the input document which is processed by the XFA application.

```
<startNode

Properties:
    desc="cdata"
>
    ...pcdata...
</startNode>
```

The startNode element is used within the following other elements:
data

When the `startNode` element is absent or empty the entire XML data document is processed. When it is non-empty it supplies an expression which identifies the root of the subtree that is processed, and the XFA application does not process any data outside the subtree.

This option affects processing during every phase.

For more information about this element, see Extended Mapping Rules.

Content

The form of the expression is "`xfasom(somExpr)`" where *somExpr* is a restricted XFA-SOM expression. The general syntax of SOM expressions is defined in the *XFA Scripting Object Model 2.0 Specification* Scripting Object Model. The expression in the `startNode` element is restricted to a fully-qualified path of element types (tag names) starting with the root of the XML-data-document and referring to a single element. The Extended Mapping Rulesgives an example of `startNode` usage.

The desc property

An attribute to hold human-readable metadata.

The startPage element

This option specifies the page number to be applied to the first page.

```
<startPage

Properties:
    desc="cdata"
>
    ...pcdata...
</startPage>
```

The startPage element is used within the following other elements:
template

Content

The content must be an integer.

The desc property

An attribute to hold human-readable metadata.

The template element

This option controls the location of the template file.

```
<template

Properties:
    desc="cdata"
>
    <relevant>  [0..1]
    <startPage>  [0..1]
    <uri>  [0..1]
</template>
```

The template element is used within the following other elements:
common

The `template` element **must** enclose a non-empty `uri` element. The XFA application **must**obtain the template using the URI specified by the content of the `uri` element.

This option takes effect during the template-load phase.

The desc property

An attribute to hold human-readable metadata.

The relevant property

This option specifies what views of the template are to be included.

For more information see "The relevant element".

The startPage property

This option specifies the page number to be applied to the first page.

For more information see "The startPage element".

The uri property

This element is used to hold a URI.

For more information see "The uri element".

The transform element

A transform element nominates a set of input data elements to which its contained set of transformations apply.

```
<transform

Properties:
    desc="cdata"
    ref="cdata"
>
    <groupParent>  [0..1]
    <ifEmpty>  [0..1]
    <nameAttr>  [0..1]
    <picture>  [0..1]
    <presence>  [0..1]
    <rename>  [0..1]
    <whitespace>  [0..1]
</transform>
```

The transform element is used within the following other elements:
data

Multiple `transform` elements can overlap in their effects. This happens when multiple `transform` elements have the same value for "ref". It also happens when one `transform` causes a node to be renamed, whereupon the new name matches the value of the `ref` attribute of another `transform`. The effect of such overlaps is sensitive to the order of the `transform` elements in the configuration document. The interaction of `transform` elements is described at length in Extended Mapping Rules. Note that, as specified in Extended Mapping Rules, it is **not recommended** to configure multiple `transform` elements with the same value for "ref".

This option takes effect during the data-load phase. Some option settings also affect the data-unload phase.

The desc property

An attribute to hold human-readable metadata.

The groupParent property

This option controls grouping of a set of related elements under a parent element.

For more information see "The groupParent element".

The ifEmpty property

This option controls the representation of empty elements in the XFA Data DOM.

For more information see "The ifEmpty element".

The nameAttr property

This option controls the renaming of nodes in the XFA Data DOM based upon the value of an attribute.

For more information see "The nameAttr element".

The picture property

This option controls the formatting of updated data.

For more information see "The picture element".

The presence property

This option controls the inclusion of nodes in the XFA Data DOM.

For more information see "The presence element".

The ref property

The `ref` attribute controls to which elements in the XML data document the `transform` applies.

If the `ref` attribute is absent the XFA application ignores the `transform` element and its content. If the value of the `ref` attribute is the empty string ("") or `*` the XFA application applies the `transform` element and its content to all data. If the `ref` attribute is present and its value is not an empty string it must be a set of element tags separated by white space. The XFA application applies the `transform` element and its content to those data elements and only those data elements with a tag matching (case-sensitive) one of the tokens in the `ref` attribute.

The rename property

This option controls the renaming of nodes in the XFA Data DOM.

For more information see "The rename element".

The whitespace property

This option controls the processing of whitespace in character data in the XFA Data DOM.

For more information see "The whitespace element".

The uri element

This element is used to hold a URI.

```
<uri

Properties:
    desc="cdata"
>
    ...pcdata...
</uri>
```

The uri element is used within the following other elements:
data debug outputXSL template xsl

This option takes effect during a phase that depends on its context. It merely supplies additional information to the element which encloses it, hence takes effect during the same phase as its enclosing element.

Content

The content of this element may be empty in some contexts. In contexts where it may be empty, emptiness indicates that the action governed by the enclosing element is not to be performed. For example, when the `uri` element inside `outputXSL` is empty, its emptiness signifies that no output XSLT processing is to be done. The `uri` element **may** be empty when it is inside a `debug`, `outputXSL`, or `xsl` element. The `uri` element **must not** be empty when its immediate container is a `data` element, a `template` element, or a `xdc` element.

When the `uri` element is not empty, its content **must** be either a filename in an **implementation-defined** format or a URI in accordance with *RFC 2396* [RFC2396]. The set of supported schemes ("http:", "ftp:", "file:", etc.) is **implementation-defined**.

The desc property

An attribute to hold human-readable metadata.

The whitespace element

This option controls the processing of whitespace in character data in the XFA Data DOM.

```
<whitespace

Properties:
    desc="cdata"
>
    ...pcdata...
</whitespace>
```

The whitespace element is used within the following other elements:
transform

This option applies only to nodes corresponding to elements with tags that match the value of the `ref` attribute of its enclosing `transform` element.

This option takes effect during the data-load phase.

For more information about this element, see Extended Mapping Rules.

Content

The content must be one of the following:

`preserve`

All white space characters are preserved.

`ltrim`

Contiguous white space characters at the start (left) end of the character data are deleted.

`normalize`

Contiguous white space characters at both ends are trimmed as for `trim`, but in addition internal contiguous groups of white space characters are replaced by single space characters.

`rtrim`

Contiguous white space characters at the end (right) of the character data are deleted.

`trim`

Contiguous white space characters at both ends of the character data are deleted, equivalent to a combination of `ltrim` and `rtrim`.

White space characters in this context include space (U0020), tab (U0009), carriage return (U000D), and line feed (U000A). This is the set of white space characters defined by [XML1.0].

The desc property

An attribute to hold human-readable metadata.

The window element

This option specifies the window size to use when performing incremental loads.

```
<window

Properties:
    desc="cdata"
>
    ...pcdata...
</window>
```

The window element is used within the following other elements:
data

The window size is in records. There are two separate settings. The first setting is the number of records before the current record that are retained in memory. These are records earlier than the current record in document order and have already been processed. The second setting is the number of records after the current record that are pre-loaded into memory. These are records that are later than the current record in document order and have not yet been processed.

Content

The content must be either a single non-negative integer or a pair of non-negative integers separated by a comma. If only one number is supplied then it is used for both settings. If two numbers are supplied the first one controls the before window and the second one controls the after window. For example, a value of 1 causes the XFA processor to hold the previous record, the current record, and the next record in memory. By contrast a value of 0,1 causes the XFA processor to hold only the current record and the next record in memory.

The desc property

An attribute to hold human-readable metadata.

The xsl element

This option controls the preprocessing of user data, of the template, or of the device control information via an XSLT interpreter.

```
<xsl

Properties:
    desc="cdata"
>
    <debug>  [0..1]
    <uri>  [0..1]
</xsl>
```

The xsl element is used within the following other elements:
data

When this element is contained by a `data` element it controls the preprocessing of data from the XML data document before loading into the XML Data DOM. When it is contained by a `template` element it controls the preprocessing of the template definition from the template document before loading into the Template DOM. When it is contained by an `xdc` element it controls the preprocessing of the device control information.

If `xsl` is empty or if it contains a `uri` element which is empty preprocessing does not take place - instead the XFA application loads the supplied data directly into the XML Data DOM or template directly into the Template DOM. If `xsl` contains a non-empty `uri` element the XFA application gets an XSLT script from the URI which is the content of `uri` and invoke an XSLT interpreter to process the supplied XML data or template document The XFA application then loads the preprocessed data into the XML Data DOM or into the Template DOM instead of loading the original data.

XSLT is defined by *XML Transformations (XSLT) Version 1.0* [XSLT].

This option takes effect during the data-load phase if it is contained in `data` or in the template-load phase if it is contained in `template`.

For more information about this element, see XSLT Transformations.

The debug property

This option controls whether the XFA application saves a copy of a preprocessed document after the XSLT interpreter has created it. It is intended for use in debugging the XSLT script.

For more information see "The debug element".

The desc property

An attribute to hold human-readable metadata.

The uri property

This element is used to hold a URI.

For more information see "The uri element".

19 Locale Set Specification

The Locale Set contains locale-specific data used in localization and canonicalization. Such data includes picture clauses for representing dates, times, numbers, and currencies. It also contains the localized names of items that appear in dates, times and currencies, such as the names of months and the names of the days of the week. It also contains mapping rules that allow picture clauses to be converted into a localized string that can be used in UI captions and prompts.

All of the elements and attributes described in this specification must belong to the following namespace:

`http://www.xfa.org/schema/xfa-locale-set/2.5/`

Note: The trailing "/" is required.

The calendarSymbols element

An element to describe the symbols of the calendar.

```
<calendarSymbols

Properties:
   name="gregorian"
>

Children:
   <dayNames>  [2]
   <monthNames>  [2]
   <eraNames>  [1]
   <meridiemNames>  [1]
</calendarSymbols>
```

The name property

Specifies the name of the calendar.

gregorian

The Gregorian calendar -- the only one supported.

The dayNames child

Each child describes the (abbreviated) names of the days of the week. There may be two such elements: one for the abreviated names and another for the unabreviated names.

The monthNames child

Each child describes the names of the months of the year. There may be two such elements: one for the abreviated names and another for the unabreviated names.

The eraNames child

Each child describes the names of the eras of the calendar.

The meridiemNames child

Each child describes the names of the meridiems.

The currencySymbol element

An element to describe currency symbols.

```
<currencySymbol

Properties:
   name="symbol| isoname|decimal"
>
   ...pcdata...
</currencySymbol>
```

Content

The data-content is interpreted as the name of currency symbol, where the name is given by the element's name property (described next).

The name property

Specifies the name of the currency symbol and the corresponding element's data-content currency symbol.

symbol

The currency symbol.

isoname

The 3-letter [ISO 4217] currency name.

decimal

The currency decimal point.

The currencySymbols element

An element to describe currency symbols.

```
<currencySymbols>

Children:
   <currencySymbol>  [3]
</currencySymbols>
```

The currencySymbol child

Each child describes a name of currency symbol.

The datePattern element

An element to describe the format of a standard date pattern.

```
<datePattern

Properties:
   name="full| long|med| short"
>
   ...pcdata...
</datePattern>
```

Content

The data-content is interpreted as one format of a date pattern, where the format is given by the element's name property (described below).

The name property

Specifies one format of a date pattern and the corresponding element's data-content date pattern.

full

The full date format.

long

The long date format.

med

The medium date format.

short

The short date format.

The datePatterns element

An element to describe the locale's standard date patterns.

```
<datePatterns>

Children:
   <datePattern>  [4]
</datePatterns>
```

The datePattern child

Each child describes the format of a date pattern.

The dateTimeSymbols element

An element to define the localized date and time pattern symbols.

```
<dateTimeSymbols>
   ...pcdata...
</dateTimeSymbols>
```

Content

The data-content is interpreted as a fixed-position array of localized date and time pattern symbols.

The day element

An element to describe the name of one of the days of the week.

```
<day>
   ...pcdata...
</day>
```

Content

The data-content is interpreted as the name of the week. Specifically, the first occurrence of this element specifies the name of the first day of the week (Sunday).The second occurrence of this element specifies the name of the second day of the week (Monday), etc...The seventh and last occurrence of this element specifies the name of the seventh day of the week (Saturday).

The dayNames element

An element to describe the names of the days of the week.

```
<dayNames

Properties:
   abbr="0|1"
>

Children:
   <day>  [7]
</dayNames>
```

The abbr property

Specifies whether the corresponding element's data-content day name is an abbreviation or not.

0

The names of the days are not abbreviated.

1

The names of the days are abbreviated.

The day child

Each child describes the name of a day of the week.

The era element

An element to describe the name of one of the eras of the calendar.

```
<era>
   ...pcdata...
</era>
```

Content

The data-content is interpreted as the name of the era. Specifically, the first occurrence of this element specifies the name of the first era of the calendar (BC).The second and last occurrence of this element specifies the name of the second era of the calendar (AD).

The eraNames element

An element to describe the names of the eras of the calendar.

```
<eraNames>

Children:
   <era>  [2]
</eraNames>
```

The era child

Each child describes the name of one era of the calendar.

The locale element

An element to describe the symbols of the locale. All symbols are localized.

```
<locale

Properties:
   name="isoname"
   desc="cdata"
>

Children:
   <calendarSymbols>  [1]
   <datePatterns>  [1]
   <timePatterns>  [1]
   <dateTimeSymbols>  [1]
   <numberPatterns>  [1]
   <numberSymbols>  [1]
   <currencySymbols>  [1]
   <typeFaces>  [1]
</locale>
```

The name property

Specifies the [RFC 1766] name of the locale.

The desc property

Specifies the description of the locale.

The calendarSymbols child

Each child describes the locale's calendric symbols.

The datePatterns child

Each child describes the locale's standard date patterns. Date patterns (date pictures) are described in "Picture Clauses and Localization" on page 358.

The timePatterns child

Each child describes the locale's standard time patterns. Time patterns (time pictures) are described in "Picture Clauses and Localization" on page 358.

The dateTimeSymbols child

Each child describes the locale's date time symbols.

The numberPatterns child

Each child describes the locale's standard number patterns. Number patterns (numeric pictures) are also described in "Picture Clauses and Localization" on page 358.

The numberSymbols child

Each child describes the locale's numeric symbols.

The currencySymbols child

Each child describes the locale's currency symbols.

The typeFaces child

Each child names a fallback font for the locale. This is new in XFA 2.5.

The localeSet element

An element to describe the symbols of the locale.

```
<localeSet>

Children:
   <locale>  [0..n]
</localeSet>
```

The locale child

Each child describes a locale's symbols.

The meridiem element

An element to describe the name of one of the aspects of the meridiem.

```
<meridiem>
   ...pcdata...
</meridiem>
```

Content

The data-content is interpreted as the name of the aspect of the meridiem. Specifically, the first occurrence of this element specifies the name of the ante-meridiem (AM).The second and last occurrence of this element specifies the name of the post-meridiem (PM).

The meridiemNames element

An element to describe the names of the meridiem.

```
<meridiemNames>

Children:
   <meridiem>  [2]
</meridiemNames>
```

The meridiem child

Each child describes the name of one aspect of the meridiem.

The month element

An element to describe the name of one of the months of the year.

```
<month>
   ...pcdata...
</month>
```

Content

The data-content is interpreted as the name of the month. Specifically, the first occurrence of this element specifies the name of the first month of the year (January).The second occurrence of this element specifies the name of the second month of the year (February), etc...The twelfth and last occurrence of this element specifies the name of the twelfth month of the year (December).

The monthNames element

An element to describe the names of the months of the year.

```
<monthNames

Properties:
   abbr="0|1"
>

Children:
   <month>  [12]
</monthNames>
```

The abbr property

Specifies whether the corresponding element's data-content month name is an abbreviation or not.

0

The names of the months are not abbreviated.

1

The names of the months are abbreviated.

The month child

Each child describes the name of a month of the year.

The numberPattern Element

An element to describe the format of a standard number pattern.

```
<numberPattern

Properties:
   name="full| long|med| short"
>
   ...pcdata...
</numberPattern>
```

Content

The data-content is interpreted as the format of a number pattern, where the format is given by the element's name property (described below).

The name property

Specifies the format of a number pattern and the corresponding element's data-content number pattern.

full

The full number format.

long

The long number format.

med

The medium number format.

short

The short number format.

The numberPatterns Element

An element to describe the locale's standard number patterns.

```
<numberPatterns>

Children:
   <numberPattern>  [4]
</numberPatterns>
```

The numberPattern child

Each child describes the format of a number pattern.

The numberSymbol element

An element to describe a number symbol.

```
<numberSymbol

Properties:
   name="decimal| grouping|percent| minus | zero"
>
   ...pcdata...
</numberSymbol>
```

Content

The data-content is interpreted as the kind of number symbol, where the kind is given by the element's name property (described below).

The name property

Specifies the name of the number symbol and the corresponding element's data-content number symbol.

decimal

The decimal radix symbol.

grouping

The grouping separator symbol.

percent

The percent symbol.

minus

The minus symbol.

zero

The zero symbol. The remaining 1-9 digits' values are assumed to follow this symbol's Unicode value.

The numberSymbols element

An element to describe number symbols.

```
<numberSymbols>

Children:
   <numberSymbol>  [5]
</numberSymbols>
```

The numberSymbol child

Each child describes a kind of number symbol.

The timePattern element

An element to describe the format of a standard time pattern.

```
<timePattern

Properties:
   name="full| long|med| short"
>
   ...pcdata...
</timePattern>
```

Content

The data-content is interpreted as the format of a time pattern, where the format is given by the element's name property (described below).

The name property

Specifies the format of a time pattern and the corresponding element's data-content time pattern.

full

The full time format.

long

The long time format.

med

The medium time format.

short

The short time format.

The timePatterns element

An element to describe the locale's standard time patterns.

```
<timePatterns>

Children:
   <timePattern>  [4]
</timePatterns>
```

The timePattern child

Each child describes the format of a time pattern.

The typeFace element

An element to name a fallback font for the locale. New in XFA 2.5.

```
<typeFace

Properties:
   name="typefacename"
/>
```

The name property

The value is the name of a typeface or font family. For example, a Japanese-language locale might specify “Kozuka Mincho Pro VI-R”.

The typeFaces element

An element to contain fallback fonts for the locale. New in XFA 2.5.

```
<typeFaces>

Children:
   <typeFace> [0..N]
</typeFaces>
```

The typeFace element

Each child names a fallback font.

20 Connection Set Specification

About the Connection Set Grammar

In XFA terminology a *connection* is a link between a subset of the data in the Form DOM and some external entity. An XDP can potentially have many data descriptions and potentially many connections for each data description.

The set of connections is contained within the XDP inside a `connectionSet` packet. There can be at most one `connectionSet` packet per XDP. Within the `connectionSet` packet there can be any number of `wsdlConnection` elements, but at most one element that is either an `xmlConnection` or an `xsdConnection`. The order of multiple `wsdlConnection` elements is not significant, but they must each have a unique name attribute.

An `xmlConnection` element associates a data description with sample XML data. An `xsdConnection` element associates a data description with an external schema in [XMLSchema] format. Both `xmlConnection` and `xsdConnection` are used for data which is to be exported as and/or imported from a standalone XML document. By contrast a `wsdlConnection` element describes how the form interacts with a WSDL-based service using SOAP doc-literal operations.

Any subform, field, or exclusion group in the Form DOM can be associated with a connection by a connect element. This causes the content of the subform, field, or exclusion group to be included in the data transferred by that connection. A given subform, field, or exclusion group can be associated with any number of different connections.

An event element in the template controls the invocation of a connection. This element can only be used with a connection defined by a `wsdlConnection` element. Invoking the connection causes the web service to be invoked using the sequence of operations listed under "Using Web Services" on page 382.

All of the elements and attributes described in this section must belong to the following namespace:

`http://www.xfa.org/schema/xfa-connection-set/2.4/`

Note: The trailing "/" is required.

The "2.4" represents the most recent version of this specification in which the connection set schema was revised. XFA applications written to this specification are expected to accept namespaces with larger version numbers and silently ignore elements and attributes that are not part of this specification. On the other hand when a future XFA application written for a later version of this specification sees the "2.4" it will know that it may have to make fixups to bring the connection set up to date with the later specification.

The following skeleton summarizes the structure of the `connectionSet` element and its descendants:

```
<connectionSet
   xmlns="http://www.xfa.org/schema/xfa-connection-set/2.4/">
   <!-- zero or more of...-->
   <wsdlConnection dataDescription="ddName" name="cxnName">
      <operation input="inputElementName"
         output="outputElementName"
         >wsdlOperationName</operation>
      <soapAction>actionURI</soapAction>
```

```
      <soapAddress>endpointURI</soapAddress>
      <wsdlAddress>wsdlURI</wsdlAddress>
   </wsdlConnection>
   <!-- at most one of eitherthis... -->
   <xmlConnection dataDescription="ddName" name="cxnName">
      <uri>sampleDataURI</uri>
   </xmlConnection>
   <!-- ...or this... -->
   <xsdConnection dataDescription="ddName" name="cxnName">
      <rootElement>elementName</rootElement>
      <uri>schemaURI</uri>
   </xsdConnection>
</connectionSet>
```

A reference to the XML Schema is available at [XMLSchema].

Connection Set Element Reference

The connectionSet element

This element is the container for the set of connections.

```
<connectionSet

Properties:
>

Children:
    <wsdlConnection>  [0..n]
    <xmlConnection>  [0..n]
    <xsdConnection>  [0..n]
</connectionSet>
```

This is the container for all information related to the connection set. There must be at most one `connectionSet` element in an XDP.

This element may contain number of `wsdlConnection` elements and at most one of either `xmlConnection` or `xsdConnection` elements.

The wsdlConnection child

This element represents one connection to a web service.

For more information see "The wsdlConnection element".

The xmlConnection child

This element represents a connection to sample data.

For more information see "The xmlConnection element".

The xsdConnection child

This element represents a connection to a schema.

For more information see "The xsdConnection element".

The operation element

This element declares the SOAP operation that is associated with its parent.

```
<operation

Properties:
    id="xml-id"
    input="cdata"
    name="xml-id"
    output="cdata"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</operation>
```

The operation element is used within the following other elements:
wsdlConnection

SOAP allows multiple operations to share the same name. When this happens the input and output attributes are used to disambiguate.

Note that the SOAP operation must use the document style and literal encoding. See [SOAP1.1] for more information about doc-literal operations.

Content

The name of the selected operation.

The id property

An identifier which may be used to identify this element in URIs.

The input property

The name of the operation's input element. If this attribute is not supplied the operation takes the default input name as specified in the WSDL specification [WSDL1.1] section 2.4.5.

The name property

An identifier that may be used to identify this element in script expressions.

The output property

The name of the operation's output element. If this attribute is not supplied the operation takes the default output name as specified in the WSDL specification [WSDL1.1] section 2.4.5.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The rootElement element

This element declares the starting point (root) within the associated schema.

```
<rootElement

Properties:
     id="xml-id"
     name="xml-id"
     use="cdata"
     usehref="cdata"
>
     ...pcdata...
</rootElement>
```

The rootElement element is used within the following other elements:
xsdConnection

Content

The name of the outermost element that was used when generating the data description from the associated W3C [XMLSchema] schema.

The id property

An identifier which may be used to identify this element in URIs.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The soapAction element

This element declares the SOAP action for its parent.

```
<soapAction

Properties:
    id="xml-id"
    name="xml-id"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</soapAction>
```

The soapAction element is used within the following other elements:
wsdlConnection

Content

The URI for the SOAP action. When the request is sent to the server, this is the value of the soapAction attribute of the soap:operation element. The soap:operation element is specified in [WSDL1.1].

The id property

An identifier which may be used to identify this element in URIs.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The soapAddress element

This element declares the host location for its parent.

```
<soapAddress

Properties:
    id="xml-id"
    name="xml-id"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</soapAddress>
```

The soapAddress element is used within the following other elements:
wsdlConnection

Content

The address of the SOAP end point. A SOAP end point consists of a protocol and a data format bound to a network address. When the request is sent to the server, this is the value of the `location` attribute of the soap:address element. The value must be a URI in the format specified by [RFC2396]. The soap:address element is specified in [WSDL1.1].

The id property

An identifier which may be used to identify this element in URIs.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The uri element

This element declares the location of the sample document or schema for its parent.

```
<uri

Properties:
     id="xml-id"
     name="xml-id"
     use="cdata"
     usehref="cdata"
>
     ...pcdata...
</uri>
```

The uri element is used within the following other elements:
xmlConnection xsdConnection

For security reasons this URI should not be honoured if it points outside the enclosing XDP or PDF package, unless the entire package is trusted.

Content

The URI for the sample document or schema.

The id property

An identifier which may be used to identify this element in URIs.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The wsdlAddress element

This element identifies the location of the service description to which its parent corresponds.

```
<wsdlAddress

Properties:
    id="xml-id"
    name="xml-id"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</wsdlAddress>
```

The wsdlAddress element is used within the following other elements:
wsdlConnection

Content

The URI for the service description.

The id property

An identifier which may be used to identify this element in URIs.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The wsdlConnection element

This element represents one connection to a web service.

```
<wsdlConnection

Properties:
    dataDescription="cdata"
    name="xml-id"
>
    <operation>  [0..1]
    <soapAction>  [0..1]
    <soapAddress>  [0..1]
    <wsdlAddress>  [0..1]
</wsdlConnection>
```

The wsdlConnection element is used within the following other elements:
connectionSet

This connection corresponds to a particular action requested from a particular service with data going in a particular direction or directions.

The dataDescription property

The name of the associated data description. If this attribute is not supplied the value defaults to the name of the root subform in the template. The data description does not include the SOAP envelope.

The name property

The name of the connection. The name must be unique among connections.

A WSDL connection may be invoked by any XFA event handler in the template. Hence the name specified by this attribute may appear as the value of the connection attribute of the `execute` element in the template.

The operation property

This element declares the SOAP operation that is associated with its parent.

For more information see "The operation element".

The soapAction property

This element declares the SOAP action for its parent.

For more information see "The soapAction element".

The soapAddress property

This element declares the host location for its parent.

For more information see "The soapAddress element".

The wsdlAddress property

This element identifies the location of the service description to which its parent corresponds.

For more information see "The wsdlAddress element".

The xmlConnection element

This element represents a connection to sample data.

```
<xmlConnection

Properties:
    dataDescription="cdata"
    name="xml-id"
>
    <uri>  [0..1]
</xmlConnection>
```

The xmlConnection element is used within the following other elements:
connectionSet

This connection encapsulates the information that a data description was derived from a particular sample document. This information is not needed by consumers of the form but may be useful for applications that modify the form template or the associated data description.

The dataDescription property

The name of the associated data description. If this attribute is not supplied the value defaults to the name of the root subform in the template.

The name property

The name of the connection. The name must be unique among connections.

The uri property

This element declares the location of the sample document or schema for its parent.

For more information see "The uri element".

The xsdConnection element

This element represents a connection to a schema.

```
<xsdConnection

Properties:
    dataDescription="cdata"
    name="xml-id"
>
    <rootElement>  [0..1]
    <uri>  [0..1]
</xsdConnection>
```

The xsdConnection element is used within the following other elements:
connectionSet

This connection encapsulates the information that a data description was derived from a particular W3C [XMLSchema] schema. This information is not needed by consumers of the form but may be useful for applications that modify the form template or the associated data description.

The dataDescription property

The name of the associated data description. If this attribute is not supplied the value defaults to the name of the root subform in the template.

The name property

The name of the connection. The name must be unique among connections.

The rootElement property

This element declares the starting point (root) within the associated schema.

For more information see "The rootElement element".

The uri property

This element declares the location of the sample document or schema for its parent.

For more information see "The uri element".

21 Data Description Specification

This chapter describes the data description grammar and provides a reference that specifies that data description syntax.

"Using Web Services" on page 382 provides more specific information on using the data description grammar in the context of a web services

About the Data Description Grammar

The XFA data description grammar describes the structure of data in a data document. It can specify a single, optional, default data description and any number of optional, context-specific data descriptions. A data description element is separate from the data document.

The XFA data description grammar is useful in the following situations:

- *Multiple views*. A data description in XFA is analogous to a "view" in a relational database. There may be multiple views for the same data because different web service operations require different subsets of the data, organized in different ways. The output document, if there is one, may require yet another organization. Hence, a single view expressed by a schema bound to the data with a schema declaration is not enough.
- *Support for a range of data*. XFA processing applications can accept existing data instance documents without alteration, despite the inconsistency of schema declarations in XML documents.

The XFA data description syntax is more concise and readable than XML Schema [XMLSchema] but does not do as much. XFA data descriptions do not include defaults and do not support validation of text content. They do, however, fully describe the namespaces, element names, attribute names, and the hierarchy which joins them. Data descriptions are described in the "Data Description Grammar" on page 834 and "Data Description Element Reference" on page 837 sections of this specification.

In keeping with the general principle that XFA is tolerant when importing data, data descriptions are *not* used to validate data coming into the XFA application. Indeed, most of any given data description is only used during output from the XFA application. (The sole part used on input is the `dd:nullType` attribute, which affects both input and output.) This means that data descriptions are not required for most processing, so XFA continues to support *ad hoc* datasets. Of course, validation of data is still possible using scripts contained in "validate" elements in the template. See "Template Specification" on page 482 for more information about validation scripts.

XFA applications produce XML output in several different contexts. The XML output document may be the final product of the XFA application. It may be sent to an HTTP host via the `SUBMIT` action. Or it may be sent to a web service. In the last case the XFA application will probably receive a reply, itself in XML format. Each of these contexts is known as a "connection". Connections are described in the "Connection Set Specification" on page 820. Web Services are described in "Using Web Services" on page 382. Submission via HTTP is described in "Submitting Data and Other Form Content to a Server" on page 375.

Data Description Grammar

Data descriptions are contained in an XDP inside the `dataSets` packet ("XDP Specification" on page 873). Each data description is enclosed within a `dataDescription` element. The order of the

dataDescription elements is not significant. dataDescription elements and their content must use the following namespace:

http://ns.adobe.com/data-description/

Note: The trailing "/" **is** required.

It is conventional to represent this namespace with the prefix dd but this is only a convention. Any prefix may be used as long as it maps to the namespace given above. Within this specification dd: is used as shorthand for "a namespace prefix mapped to the data description namespace". Similarly xsi: is used as shorthand for "a namespace prefix mapped to the XML Schema Instance namespace (http://www.w3.org/2001/XMLSchema-instance) ". The XML Schema Instance namespace is defined in [XMLSchema].

Each dataDescription element has a name attribute. The value of the name attribute must be unique. One dataDescription element may have a name equal to the name attribute of the template's root subform. This data description is the default.

The data description itself (i.e. the content of the dataDescription element) is a picture of the data structure, without content but with optional namespace markup. For example, consider the following sample data.

Example 21.1 Sample data

```
<po:order xmlns:po="http://www.example.com/order">
   <po:orderid>A314</po:orderid>
   <po:customer>
      <po:lastname>Coyote</po:lastname>
      <po:firstname>Wile</po:firstname>
   </po:customer>
   <po:item>
      <po:desc>super magnet</po:desc>
      <po:qty>1</po:qty>
   </po:item>
</po:order>
```

The simplest data description for this document is generated simply by stripping out the text content from the sample data, as follows.

Example 21.2 Basic data description

```
<dd:dataDescription
   xmlns:dd="http://ns.adobe.com/data-description/"
   name="order">
   <po:order xmlns:po="http://www.example.com/order">
      <po:orderid/>
      <po:customer>
         <po:lastname/>
         <po:firstname/>
      </po:customer>
      <po:item>
         <po:desc/>
         <po:qty/>
      </po:item>
   </po:order>
</dd:dataDescription>
```

The simple data description shown above requires that the data document have exactly the same structure as the description. Namespaced markup provides a way to specify alternate structure and repeated or optional elements. It corresponds to a subset of W3C XML Schema [XMLSchema]. Most of the markup is `dd:` attributes applied to non-`dd:` elements. In addition a `dd:group` element, which does not correspond to an element in the data document, is provided for more complicated cases.

For example, the following data description relaxes the previous data description using added `dd:` markup (highlighted in **bold**).

Example 21.3 Data description augmented with markup

```
<dd:dataDescription
xmlns:dd="http://ns.adobe.com/data-description/"
name="order">
   <po:order xmlns:po="http://www.example.com/order">
      <po:orderid/>
      <po:customer>
         <po:lastname/>
         <po:firstname dd:minOccur="0">
      </po:customer>
      <po:item dd:maxOccur="-1">
         <po:desc/>
         <po:qty/>
      </po:item>
   </po:order>
</dd:dataDescription>
```

This data description still matches the original sample data, but the markup makes it more flexible. For example, it also matches the following data which has no `po:firstname` element and has multiple `po:item` elements.

Example 21.4 Data which also matches the augmented data description

```
<po:order xmlns:po="http://www.example.com/order">
   <po:orderid>A314</po:orderid>
   <po:customer>
      <po:lastname>Coyote</po:lastname>
   </po:customer>
   <po:item>
      <po:desc>super magnet</po:desc>
      <po:qty>1</po:qty>
   </po:item>
   <po:item>
      <po:desc>steel ball bearing</po:desc>
      <po:qty>1000</po:qty>
   </po:item>
   <po:item>
      <po:desc>mallet (large)</po:desc>
      <po:qty>1</po:qty>
   </po:item>
</po:order>
```

Data Description Element Reference

All of the elements and attributes described in this section must belong to the following namespace:

`http://ns.adobe.com/data-description/`

Note that the trailing "/" **is** required.

dd:dataDescription Element

This element is the container for a data description. More than one `dataDescription` element may reside in a `dataset` element.

dd:name attribute

This attribute supplies a name for the data description. The attribute must be supplied and the name ("XFA Names" on page 62) must be unique across `dataDescription` elements. If the name is the same as the name of the template's root subform, this data description is the default data description.

dd:group Element

This element groups together its child elements, but without there being a corresponding element in the data document. For example, the "location" element in a data document contains either an "`x`" element followed by a "`y`" element, or an "`r`" element followed by a `theta` element. Hence the following is a valid fragment:

```
<location>
   <x>1.234</x>
   <y>5.678</y>
</location>
```

But the following is also valid:

```
<location>
   <r>5.432</r>
   <theta>31.97</theta>
</location>
```

This can be declared in a data description as follows:

```
<location dd:model="choice">
   <dd:group>
      <x/>
      <y/>
   </dd:group>
   <dd:group>
      <r/>
      <theta/>
   </dd:group>
</location>
```

Another use of `dd:group` is to provide a way to apply dd: attributes to groups of elements, again without any corresponding element appearing in the data document. For example, the "polyline" element always contains sets of coordinates. Each coordinate consists of an "x" element followed by a "y" element. There must be at least two coordinates, but there is no maximum number. This can be declared in a data description as follows:

```
<polyline>
   <dd:group dd:minOccur="2" dd:maxOccur="-1">
      <x/>
      <y/>
   </dd:group>
</polyline>
```

This matches the following sample data:

```
<polyline>
   <x>1</x>
   <y>1</y>
   <x>1</x>
   <y>6</y>
   <x>20</x>
   <y>15</y>
</polyline>
```

Note that the white space in the data description differs from the white space in the above sample data. Nonetheless the sample data matches the data description. This is a consequence of the fact that white space in data groups is not significant, as explained in "White Space Handling" on page 131.

dd:maxOccur Attribute

This attribute sets the maximum number of times the element may occur in a contiguous sequence. The default is 1, that is, by default no more than one occurrence is allowed. The special value-1 means that there is no limit to the number of times the element may repeat. If the value is not -1 it must be a positive integer.

The `dd:maxOccur` attribute corresponds in function to the XML Schema `xsd:maxOccurs` attribute. Note however that the attribute name differs (no final "s" on `dd:maxOccur`) and that `xsd:maxOccurs` uses the value unbounded rather than-1.

Note that when an element has `dd:model` set to unordered, its direct children must not have `dd:maxOccur` set to anything larger than 1. This is the same restriction that applies in [XMLSchema] to the children of the analogous element `xsd:all`.

For example, the following fragment declares that it is acceptable for the `po:item` element to repeat without limit. Hence a single purchase order can list any number of purchased items.

```
<po:item dd:maxOccur="-1">
   <po:desc/>
   <po:qty/>
</po:item>
```

The following fragment declares that the number of "attendee" elements inside the "meeting" element is limited to twelve (perhaps the capacity of the room):

```
<meeting>
   <attendee dd:maxOccur="12">
</meeting>
```

dd:minOccur Attribute

This attribute sets the minimum number of times the element must occur in a contiguous sequence. The default is 1, that is, by default at least one occurrence is required. The value0 means that the element is optional. If the value is not 0 it must be a positive integer.

The dd:minOccur attribute corresponds in function to the XML Schema xsd:minOccurs attribute. Note however that the attribute name differs (no final "s" on dd:minOccur).

Note that when an element has dd:nullType set to exclude it must also have dd:minOccur set to0.

The following fragment declares that "firstname" is not required in the purchase order. Without the dd:minOccur attribute the value of "firstname" could be the empty string but the element could not be omitted entirely. On the other hand "lastname" continues to be mandatory.

```
<po:customer>
   <po:lastname/>
   <po:firstname dd:minOccur="0"/>
</po:customer>
```

Hence the following fragment of data is valid:

```
<po:customer>
   <po:lastname>Smith</po:lastname>
</po:customer>
```

The following fragment declares that a meeting must be attended by at least two people:

```
<meeting>
   <attendee dd:minOccur="2"/>
</meeting>
```

dd:model Attribute

This attribute controls the way in which the children of the element are related. The value of dd:model must be one of the following:

choice

The data must have a child element or elements corresponding to just one of the children of this element. This corresponds to the xsd:choice element in [XMLSchema].

ordered

The data must have child elements corresponding to each of the children of this element (except for children with dd:minOccur equal to 0, which are optional). The children must occur in the same order that they are declared here. This corresponds to the xsd:sequence element in [XMLSchema]. This is the default.

unordered

The data must have child elements corresponding to each of the children of this element (except for children with dd:minOccur equal to 0, which are optional). The children may occur in any order. This corresponds to the xsd:all element in XML Schema.

Note: When an element has dd:model set to unordered, its direct children must not have dd:maxOccur set to anything larger than1. This is the same restriction imposed by [XMLSchema] upon the children of the analogous element xsd:all.

The following fragment illustrates a simple use of dd:model with a value of choice:

```
<payment dd:model="choice">
   <cash/>
   <visa/>
   <amex/>
</payment>
```

The following fragment has been cooked up to illustrate what happens when a child element governed by choice has a `dd:maxOccur` greater than one. In the example, a pizza order can be the house special or it can have *à la carte* toppings. If the pizza is *à la carte*, multiple toppings may be specified. On the other hand if the house special is chosen toppings must not be specified. The `dd:model` attribute with a value of choice makes "houseSpecial" and "topping" mutually exclusive:

```
<pizza dd:model="choice">
   <houseSpecial/>
   <topping dd:maxOccur="-1">
<pizza/>
```

Hence the following fragment of data is valid:

```
<pizza>
   <houseSpecial>
</pizza>
```

But the following is also valid:

```
<pizza>
   <topping>pepperoni</topping>
   <topping>green peppers</topping>
   <topping>onions</topping>
</pizza>
```

The following fragment illustrates a simple use of `dd:model` with a value of ordered. The fragment declares that the address in the data must have each of the child elements in the exact order given. Any child element may be empty but it must be present.

```
<address dd:model="ordered">
   <streetNumber/>
   <streetName/>
   <city/>
   <postalCode/>
</address>
```

Hence the following fragment of data is valid:

```
<address>
   <streetNumber>47</streetNumber>
   <streetName>Main Street</streetName>
   <city/>
   <postalCode/>
</address>
```

Since ordered is the default, the same result would be obtained by omitting the `dd:model` attribute entirely.

The following fragment illustrates a simple use of `dd:model` with a value of unordered. It is the same as the previous example except for the value of `dd:model`.

```
<address dd:model="unordered">
   <streetNumber/>
   <streetName/>
   <city/>
   <postalCode/>
</address>
```

The result is almost the same as the previous example using ordered, but more forgiving. Any data document that matches the previous example will also match this data description. In addition, this data description also matches data documents in which the order of the “streetNumber”, “streetName”, “city”, and “postalCode” elements is switched around. However they are all still required to be present. Hence the following fragment of data is valid:

```
<address>
   <city/>
   <streetName>Main Street</streetName>
   <postalCode/>
   <streetNumber>47</streetNumber>
</address>
```

The following fragment illustrates the effect of combining unordered with one or more children having `dd:minOccur` set to a value of 0. Any element with `dd:minOccur` set to a value of 0 is optional. Consider the following fragment:

```
<address dd:model="unordered">
   <streetNumber/>
   <streetName/>
   <city dd:minOccurs="0"/>
   <postalCode dd:minOccurs="0"/>
</address>
```

Given the above data declaration fragment, the following data fragment is valid:

```
<address>
   <streetName>Main Street</streetName>
   <streetNumber>47</streetNumber>
   </address>
```

dd:nullType Attribute

This attribute controls the mapping between data elements and null nodes in a DOM. A null node is distinct from a node with content of the empty string. A null node has no value at all – it is null in the database sense. The base XML 1.0 standard [XML1.0] does not provide a standard way to represent null nodes. Sometimes an empty element is represented internally as a null node, but other times it is represented as a normal node with a value of the empty string. XML Schema [XMLSchema] defines a syntax using the namespaced attribute `xsi:nil`. The `dd:nullType` attribute specifies which method is used for this element and, unless overridden, inherited its descendants.

The value of the attribute must be one of the following:

`empty`

On output null nodes are represented by empty elements. On input empty elements are mapped to null nodes, as are elements marked as null using `xsi:nil="true"`. This is the default.

`exclude`

On output null nodes are excluded from the XML document. On input elements marked as null using `xsi:nil="true"`are mapped to null nodes. Elements that are empty but not marked using `xsi:nil="true"` are mapped to regular nodes with values of the empty string.

Note: When the element has `dd:nullType` set to exclude it must also have a `dd:minOccur` attribute set to 0. Failure to uphold this rule would lead to a schema violation when the node was null because `dd:nullType` would require that the element be omitted and at the same time `dd:minOccur` would require that it be included.

xsi

On output null nodes are represented by empty elements with the attribute `xsi:nil` equal to true, as defined in XML Schema [XMLSchema]. On input any element (empty or not) with `xsi:nil="true"` is mapped to a null node, while empty elements that do not have `xsi:nil="true"` are mapped to regular nodes with values of the empty string.

Note: This applies only to elements. Attributes with the value of empty string are governed by the `dd:reqAttrs` attribute.

dd:reqAttrs Attribute

This attribute lists the names of mandatory attributes for the element. The names in the list are separated by white space. The order of the names is not significant. Each name in the list must match the name (including namespace) of an attribute on the element. If an attribute name in the list has no namespace prefix it is imputed to inherit the namespace of the element, just as it does when used in the instance document.

On input an attribute with the value of empty string is treated the same way as any other attribute. On output, when an attribute is mandatory but the attribute is not present in the DOM, the XFA application generates an attribute with the value of the empty string. By contrast when the attribute is not mandatory and it is not present in the DOM it is omitted from the XML document.

For example, the following fragment declares that in the “shirt” element the attributes “color” and “size” are mandatory, but “supplier” is optional. All of the attributes inherit the namespace of their element.

```
<t:shirt color="" supplier="" size="" dd:reqattrs="size color"/>
```

The following example declares two mandatory attributes. The element is in the default namespace. One of its mandatory attributes is also in the default namespace, but the other is in a different namespace.

```
<animal name="" vet:species=""
   dd:reqattrs="name vet:species"/>
```

22 Source Set Specification

The Source Set Data Object Model (DOM) provides a centralized mechanism for specifying connections between XFA applications and Microsoft® ActiveX® Data Objects (ADO) acting as data sources and/or sinks. Most of the time ADO is used to connect to databases, although in principle it can be used for other things. The connection information may be specified in XML format and passively loaded at the start of processing. However the Source Set DOM also supports a scripting interface which allows user-supplied scripts to initiate and control ADO transactions.

In order to use the Source Set DOM it is necessary to understand the Microsoft® ADO model. XFA supports version 2.6 of the ADO Application Programming Interface (API). There is a close correlation between the ADO API and the structure of the Source Set DOM. The ADO API is defined in [ADO].

The actual location of the source set document is variable. XFA provides a convenient mechanism for packaging XML data ("XDP Specification" on page 873), which might be used to bundle source set information with the template and other relevant material. However, XFA applications are free to use scripting instead of or in addition to XDP.

Support for ADO is optional for XFA processors. This is necessarily so because ADO is platform-dependent while XFA is a platform-independent standard.

The names used for nodes in the Source Set DOM are taken directly from the ADO API, as defined in [ADO].

The Source Set Data Object Model

The Source Set Data Object Model (Source Set DOM) encapsulates the XFA source set information and provides standard interfaces to it. The Source Set DOM contains data objects organized in a tree structure. The source set tree is itself a branch within a larger XFA tree.

The `sourceSet` element and the standard contents shown below must belong to the namespace `http://www.xfa.org/schema/xfa-source-set/1.0/` which is known as the source set namespace. Custom elements may belong to the source set namespace but are not required to.

Defaults

Many of the elements and attributes in the source set DOM have default values defined. When the application loads the source set document, where an element or attribute has been omitted, the application inserts the default value. The defaults appear as nodes in the source set DOM, just as though they had been loaded from the source set document.

Scripting Interface

The Source Set DOM is part of a larger tree that holds all exposed XFA objects. The single large tree makes it possible to refer to XFA objects using a unified format known as a Scripting Object Model (SOM) expression. The grammar of SOM expressions is described in "Scripting Object Model" on page 73. Briefly, an expression consists of a sequence of node names separated by periods ("." characters). Starting from some point in the XFA tree, each successive name identifies which child of the current node to descend to. The root of the Source Set DOM is a child of the root `xfa` node. Hence, the `sourceSet` node itself is

`xfa.sourceSet`. The timeout value for the connection used by the first source would be referenced by the SOM expression `xfa.sourceSet.source.connection.timeout`.

In addition SOM expressions recognize the short-form "$sourceSet" as equivalent to `xfa.sourceSet`. Thus for example the timeout value mentioned in the preceding paragraph could be referenced as `$sourceSet.source.connection.timeout`.

The scripting interface makes it possible for user-supplied scripts to inspect and modify the contents of the Source Set DOM. Some of these objects make available methods that allow scripts to exert detailed control over the database transaction.

Source Set Element Reference

The bind element

Associates an item of data from the database with a node in the Data DOM

```
<bind

Properties:
    contentType="cdata"
    id="xml-id"
    name="xml-id"
    ref="cdata"
    transferEncoding="none | base64"
    use="cdata"
    usehref="cdata"
>
</bind>
```

The bind element is used within the following other elements:
source

The contentType property

The type of data retrieved from and/or inserted into the database, expressed as a MIME type. For more information, please see [RFC2046].

The following values are allowed for textual data:

text/plain

Unadorned text. The XFA application may accept content that does not conform strictly to the requirements of the MIME type.

pcdata

Support for other text types, such as text/html is implementation-defined.

When the data is an image, a suitable MIME-type must be supplied for this property to tell the application that the content is an image. However, the application is free to override the supplied value if upon examining the image data it determines that the image data is of a different type. Which image types are supported is implementation-defined.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The ref property

An XFA-SOM expression pointing to a node in an XFA DOM. Normally this is a node in the Data DOM or Form DOM. This expression must be fully-qualified.

The transferEncoding property

The encoding of binary content in the referenced document.

`none`

The referenced document is not encoded. If the referenced document is specified via a URI then it will be transferred as a byte stream. If the referenced document is inline it must conform to the restrictions on PCDATA.

`base64`

The binary content is encoded in accordance with the base64 transfer encoding s specified in [RFC2045].

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*`SOM_expr`*) where *`SOM_expr`* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The boolean element

Encloses custom data that is represented as a Boolean value.

```
<boolean

Properties:
     id="xml-id"
     name="xml-id"
     use="cdata"
     usehref="cdata"
>
     ...pcdata...
</boolean>
```

The boolean element is used within the following other elements:
extras

Content

The be empty or contain one of the following:

`0`

The content represents a logical value of false.

`1`

The content represents a logical value of true.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The command element

A command to the database.

```
<command

Properties:
    id="xml-id"
    name="xml-id"
    timeout="30 | integer"
    use="cdata"
    usehref="cdata"
>
    <delete>  [0..1]
    <insert>  [0..1]
    <query>  [0..1]
    <update>  [0..1]
</command>
```

The command element is used within the following other elements:
source

When a database operation is invoked the set of sibling command elements is executed sequentially in document order. The operations can not nest, that is, each command must terminate before the next one starts.

The delete property

A SQL command to perform a simple delete operation.

For more information see "The delete element".

The id property

A unique identifier that may be used to identify this element as a target.

The insert property

A SQL command to perform a simple insert operation.

For more information see "The insert element".

The name property

An identifier that may be used to identify this element in script expressions.

The query property

Controls a cursor and associated record set.

For more information see "The query element".

The timeout property

Time in seconds to wait before giving up on the operation. The default value is 30 seconds.

The value of this attribute must be a non-negative integer. A value of `0` means no timeout.

The update property

A SQL command to perform a simple update.

For more information see "The update element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*`SOM_expr`*) where *`SOM_expr`* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The connect element

This element controls a connection to a database.

```
<connect

Properties:
    delayedOpen="0 | cdata"
    id="xml-id"
    name="xml-id"
    timeout="15 | integer"
    use="cdata"
    usehref="cdata"
>
    <connectString>  [0..1]
    <password>  [0..1]
    <user>  [0..1]

Children:
    <extras>  [0..n]
</connect>
```

The connect element is used within the following other elements:
source

The connectString property

A string identifying a data source.

For more information see "The connectString element".

The delayedOpen property

Controls how and when the connection is opened. The value must be one of the following:

0

Automatically opens the connection on first use.

1

Does not automatically open the connection. The connection must be opened by script, using the db object, before it can be used.

The extras child

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The password property

The password used to connect to the database.

For more information see "The password element".

The timeout property

The number of seconds to wait for the connection to open before timing out. The default value is 15 seconds.

The value of this attribute must be a non-negative integer. A value of `0` means no timeout.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*`SOM_expr`*) where *`SOM_expr`* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The user property

Username to be used when connecting to the database.

For more information see "The user element".

The connectString element

A string identifying a data source.

```
<connectString

Properties:
     id="xml-id"
     name="xml-id"
     use="cdata"
     usehref="cdata"
>
     ...pcdata...
</connectString>
```

The connectString element is used within the following other elements:
connect

Content

The content of this element corresponds to the "connectString" parameter supplied to the "Create" method described in the ADOX API Reference [ADO].

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The delete element

A SQL command to perform a simple delete operation.

```
<delete

Properties:
    id="xml-id"
    name="xml-id"
    use="cdata"
    usehref="cdata"
>
    ...pcdata...
</delete>
```

The delete element is used within the following other elements:
command

The query element can be used to delete record(s) using a cursor and record set. By contrast this element does not use a cursor or record set. Instead it performs a one-time operation with no persistent context.

Content

The SQL command to send to the database. Operation suspends until the database reports that the operation is complete.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The extras element

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

```
<extras

Properties:
    id="xml-id"
    name="xml-id"
    use="cdata"
    usehref="cdata"
>

Children:
    <boolean>  [0..n]
    <extras>  [0..n]
    <integer>  [0..n]
    <text>  [0..n]
</extras>
```

The extras element is used within the following other elements:
connect extras recordSet

The boolean child

Encloses custom data that is represented as a Boolean value.

For more information see "The boolean element".

The extras child

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The integer child

Encloses custom data that is represented as an integer.

For more information see "The integer element".

The name property

An identifier that may be used to identify this element in script expressions.

The text child

Encloses a custom property which is represented as text.

For more information see "The text element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*`SOM_expr`*) where *`SOM_expr`* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The insert element

A SQL command to perform a simple insert operation.

```
<insert

Properties:
     id="xml-id"
     name="xml-id"
     use="cdata"
     usehref="cdata"
>
     ...pcdata...
</insert>
```

The insert element is used within the following other elements:
command

The query element can be used to insert record(s) using a cursor and record set. By contrast this element does not use a cursor or record set. Instead it performs a one-time operation with no persistent context.

Content

The SQL command to send to the database. Operation suspends until the database reports that the operation is complete.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The integer element

Encloses custom data that is represented as an integer.

```
<integer

Properties:
     id="xml-id"
     name="xml-id"
     use="cdata"
     usehref="cdata"
>
     ...pcdata...
</integer>
```

The integer element is used within the following other elements:
extras

Content

This element may enclose integer-data which is an optional leading minus sign (Unicode character U+002D), followed by a sequence of decimal digits (Unicode characters U+0030 - U+0039). Alternatively it may be empty.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The map element

Maps a database column to a bind element descended from the same source element.

```
<map

Properties:
     bind="cdata"
     from="cdata"
     id="xml-id"
     name="xml-id"
     use="cdata"
     usehref="cdata"
>
</map>
```

The map element is used within the following other elements:
query

The bind property

Name of the bind element to use for this column.

The from property

Name of the database column controlled by this element.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The password element

The password used to connect to the database.

```
<password

Properties:
     id="xml-id"
     name="xml-id"
     use="cdata"
     usehref="cdata"
>
     ...pcdata...
</password>
```

The password element is used within the following other elements:
connect

Content

The password to be sent at connect time. This element may be omitted or empty. In interactive contexts when authorization is required but the element is omitted or empty the XFA processor prompts for the password. In non-interactive contexts when authorization is required the password must be supplied here.

Communicating any password in clear text is risky. If the password is supplied, steps should be taken to ensure that the form's distribution is limited.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The query element

Controls a cursor and associated record set.

```
<query

Properties:
    commandType="unknown | text | table | storedProc"
    id="xml-id"
    name="xml-id"
    use="cdata"
    usehref="cdata"
>
    <recordSet>  [0..1]
    <select>  [0..1]

Children:
    <map>  [0..n]
</query>
```

The query element is used within the following other elements:
command

Despite the name, this element can also be used to delete, insert, and update records.

The commandType property

The type of select string in the child select element. This has the same function as "CommandTypeEnum" defined in the ADO API Reference [ADO], but it has a more restricted range of values. The value must be one of the following:

unknown

Indicates that the type of select string is not known. The XFA application or database is required to parse the string to find out.

text

Indicates that the select string is a SQL command or textual stored procedure call.

table

Indicates that the select string is the name of a table. All of the columns in the table are returned by the query.

storedProc

Indicates that the select string is the name of a stored procedure.

The id property

A unique identifier that may be used to identify this element as a target.

The map child

Maps a database column to a bind element descended from the same source element.

For more information see "The map element".

The name property

An identifier that may be used to identify this element in script expressions.

The recordSet property

Controls the type and behavior of the cursor for the record set.

For more information see "The recordSet element".

The select property

String identifying the records in the record set.

For more information see "The select element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The recordSet element

Controls the type and behavior of the cursor for the record set.

```
<recordSet

Properties:
    bofAction="moveFirst | stayBOF"
    cursorLocation="client | server"
    cursorType="forwardOnly | unspecified | keyset | dynamic |
                static"
    eofAction="moveLast | stayEOF | addNew"
    id="xml-id"
    lockType="readOnly | unspecified | pessimistic | optimistic |
              batchOptimistic"
    max="0 | integer"
    name="xml-id"
    use="cdata"
    usehref="cdata"
>

Children:
    <extras>  [0..n]
</recordSet>
```

The recordSet element is used within the following other elements:
query

The bofAction property

Controls what happens when the cursor reaches a position in front of the first record in the record set.

moveFirst

Move forward to the first record in the set.

stayBOF

Stay at the beginning. If an insert is performed the new record will be placed in front of the record that is currently the first in the set.

The cursorLocation property

Controls where the cursor is held. Note that this choice has side-effects. See the section "The Significance of Cursor Location" in Chapter 8 of the ADO Programmer's Guide for more information [ADO].

client

The cursor is held on the client.

server

The cursor is held on the server.

The cursorType property

Controls what type of cursor is used. The text within quotation marks for each value below is taken directly from the ADO 2.6 API Reference [ADO], under the entry for "CursorTypeEnum".

`forwardOnly`

"Default. Uses a forward-only cursor. Identical to a static cursor, except that you can only scroll forward through records. This improves performance when you need to make only one pass through a Recordset."

`unspecified`

"Does not specify the type of cursor." The database or XFA processor decides on the type of cursor. XFA does not provide a way to determine what type of cursor was chosen, so the form creator must make pessimistic assumptions.

`keyset`

"Uses a keyset cursor. Like a dynamic cursor, except that you can't see records that other users add, although records that other users delete are inaccessible from your Recordset. Data changes by other users are still visible."

`dynamic`

"Uses a dynamic cursor. Additions, changes, and deletions by other users are visible, and all types of movement through the Recordset are allowed, except for bookmarks, if the provider doesn't support them."

`static`

"Uses a static cursor. A static copy of a set of records that you can use to find data or generate reports. Additions, changes, or deletions by other users are not visible."

The eofAction property

Controls what happens when the cursor reaches a position past the last record in the record set.

`moveLast`

Notionally reposition to the last record in the set. When using a `forwardOnly` cursor the effect is to stay at the last record rather than advancing to EOF.

`stayEOF`

Stay at the end. If an attempt is made to fetch another record in the forward direction the request will return a null.

`addNew`

Insert a new record at the end.

The extras child

An enclosure around one or more sets of custom properties. The content of this element may be used by custom applications.

For more information see "The extras element".

The id property

A unique identifier that may be used to identify this element as a target.

The lockType property

Controls what type of lock to apply against other users and/or processors. The text within quotation marks below is taken from the section "Types of Locks" in Chapter 8 of the ADO Programmer's Guide [ADO].

`readOnly`

"Indicates read-only records. You cannot alter the data. A read-only lock is the 'fastest' type of lock because it does not require the server to maintain a lock on the records."

`unspecified`

"Does not specify a type of lock." The lock type is determined by the database and/or XFA application. XFA does not provide a way to find out what type of lock was chosen, so the form creator must make pessimistic assumptions.

`pessimistic`

"Indicates pessimistic locking, record by record. The provider does what is necessary to ensure successful editing of the records, usually by locking records at the data source immediately before editing. Of course, this means that the records are unavailable to other users once you begin to edit, until you release the lock by calling Update. Use this type of lock in a system where you cannot afford to have concurrent changes to data, such as in a reservation system."

`optimistic`

"Indicates that the provider uses optimistic locking - locking records only when you call the Update method. This means that there is a chance that the data may be changed by another user between the time you edit the record and when you call Update, which creates conflicts. Use this lock type in situations where the chances of a collision are low or where collisions can be readily resolved."

`batchOptimistic`

"Indicates optimistic batch updates. Required for batch update mode.
Many applications fetch a number of rows at once and then need to make coordinated updates that include the entire set of rows to be inserted, updated, or deleted. With batch cursors, only one round trip to the server is needed, thus improving update performance and decreasing network traffic. Using a batch cursor library, you can create a static cursor and then disconnect from the data source. At this point you can make changes to the rows and subsequently reconnect and post the changes to the data source in a batch." XFA does not provide a way to control disconnecting and reconnecting. Whether or not this optimization is performed is implementation-defined.

The max property

The maximum number of records to return. This corresponds to the "MaxRecords" property of the Recordset ADO object. The value of this attribute must be a non-negative integer.

The special value 0 means there is no limit. This is the default.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The select element

String identifying the records in the record set.

```
<select

Properties:
     id="xml-id"
     name="xml-id"
     use="cdata"
     usehref="cdata"
>
     ...pcdata...
</select>
```

The select element is used within the following other elements:
query

Content

Depending on the value of `commandType` in the parent query element, this string is a SQL select command, the name of a table, the name of a stored procedure, or a textual procedure call.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The source element

This element describes a database or some other external data source/sink.

```
<source

Properties:
     id="xml-id"
     name="xml-id"
     use="cdata"
     usehref="cdata"
>
     <connect>  [0..1]

Children:
     <bind>  [0..n]
     <command>  [0..n]
</source>
```

The source element is used within the following other elements:
sourceSet

The bind child

Associates an item of data from the database with a node in the Data DOM

For more information see "The bind element".

The command child

A command to the database.

For more information see "The command element".

The connect property

This element controls a connection to a database.

For more information see "The connect element".

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*`SOM_expr`*) where *`SOM_expr`* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The sourceSet element

This element contains the set of source descriptions.

```
<sourceSet

Properties:
    id="xml-id"
    name="xml-id"
    use="cdata"
    usehref="cdata"
>

Children:
    <source>  [0..n]
</sourceSet>
```

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The source child

This element describes a database or some other external data source/sink.

For more information see "The source element".

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The text element

Encloses a custom property which is represented as text.

```
<text

Properties:
     id="xml-id"
     name="xml-id"
     use="cdata"
     usehref="cdata"
>
     ...pcdata...
</text>
```

The text element is used within the following other elements:
extras

Content

This element may contain text data which is simple XML PCDATA. It may also be empty.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The update element

A SQL command to perform a simple update.

```
<update

Properties:
     id="xml-id"
     name="xml-id"
     use="cdata"
     usehref="cdata"
>
     ...pcdata...
</update>
```

The update element is used within the following other elements:
command

The query element can be used to update record(s) using a cursor and record set. By contrast this element does not use a cursor or record set. Instead it performs a one-time operation with no persistent context.

Content

The SQL command to send to the database. Operation suspends until the database reports that the operation is complete.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The user element

Username to be used when connecting to the database.

```
<user

Properties:
     id="xml-id"
     name="xml-id"
     use="cdata"
     usehref="cdata"
>
     ...pcdata...
</user>
```

The user element is used within the following other elements:
connect

Content

Username sent when connecting to the database. If authorization is not needed this element may be omitted or empty.

The id property

A unique identifier that may be used to identify this element as a target.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes another object in the same document as a prototype for this object. The content of this property is either a SOM expression (which cannot start with '#') or a '#' character followed by an XML ID.

Starting with XFA 2.4 the object used as a prototype no longer needs to be the child of `proto`. Any object of the appropriate class can be used as a prototype.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

The usehref property

Invokes an external object as a prototype for this object. The content of this property is an URL, followed by '#', followed by either an XML ID or by som(*SOM_expr*) where *SOM_expr* represents a SOM expression.

The XML ID or SOM expression is resolved in the context of the external document.

If both `use` and `usehref` are non-empty `usehref` takes precedence.

23 XDP Specification

This chapter contains a narrative description of the XML Data Packaging (XDP) grammar ("About the XDP Grammar") and a reference for the XDP root element and XDP packets ("XDP Element Language Syntax").

About the XDP Grammar

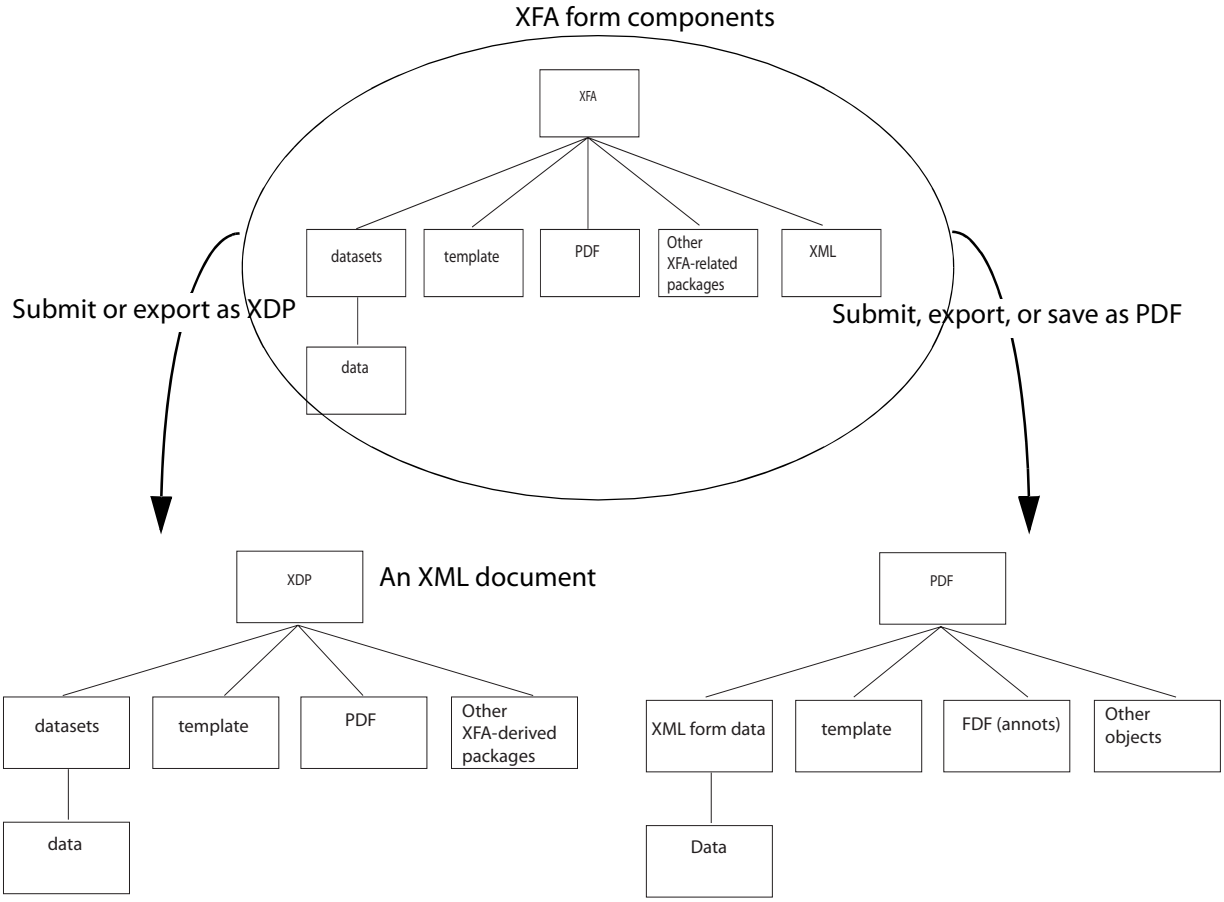
XDP is an XML grammar that provides a mechanism for packaging XFA components within a surrounding XML container. Such XFA components may include a PDF document, PDF subassemblies (annots and data), XFA form data, and custom XFA components. Packaging XFA components within an XML container may be important for XML-based applications that support XFA.

Role of XDP

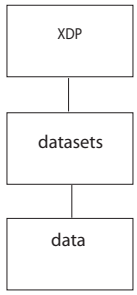
The XFA components come from various sources, each corresponding to a different type of XML grammar, language (PDF), or language subassembly (annots and PDF data). In some cases, the XFA components are serialized from a DOM representation (datasets). In other cases, the XFA components come from file-based representations (templates). The source of a particular XFA component depends on whether the in-memory representation may have changed during a session.

When an XFA processing application is requested to submit or export XFA components, it packages those components as an XDP document (below left) or as a single PDF document (below right). XDP and PDF can represent the same XFA form component; however, they differ in their root nodes and in the compliance with XML. That is, XDP is XML-compliant, while PDF is not.

Packaging of XFA form components into XDP or PDF



The most common use of XDP is to submit data to a server that expects to process XML. Such a data-only XDP document is shown at right.



The types of XFA components packaged within XDP is discretionary. It can be used to submit any combination of packages containing XFA components. Packages may include custom XFA components, provided those components comply with the guidelines described later in this section.

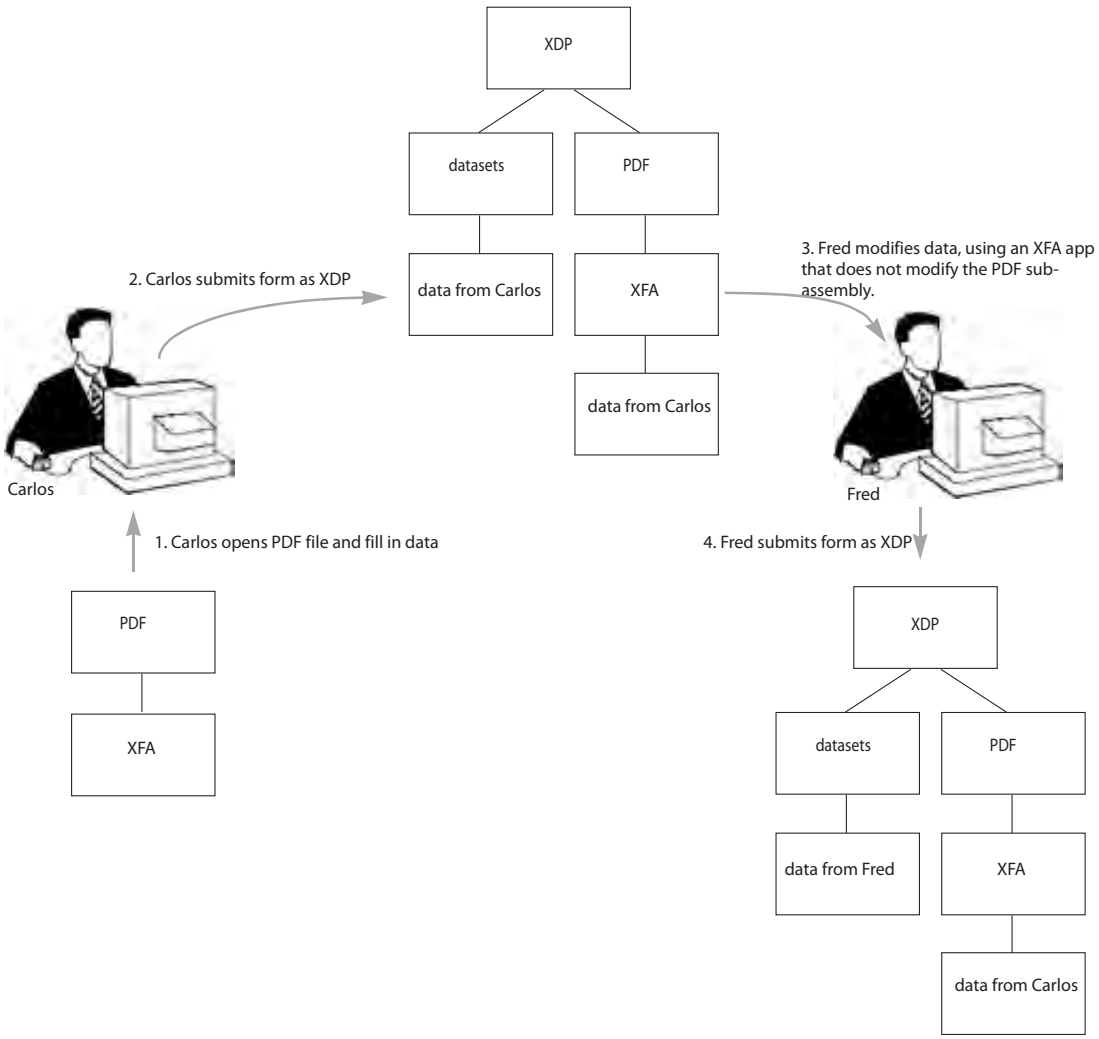
Overview of Packaging a PDF Document in XDP Format

While the PDF format may be most recognized as a visual representation of a document, PDF is also a packaging format that encloses many different types and ranges of content. Each of these units of content is referred to as a subassembly by this document. For example, a PDF document representing an interactive form may enclose an XML fragment representing the form-data subassembly of the document.

Consider an XML processing application that wishes to process the XML form-data subassembly of a PDF form. Such an application could not directly consume this XML-based subassembly of the PDF because it is enclosed within a non-XML format: PDF.

While extending such an application to interpret the PDF and navigate to the XML data content contained within may be straightforward, this cannot occur by solely employing commonly available XML tools such as an XML parser.

The XDP format provides an alternate means of expressing the PDF document in which the outer packaging is described with an XML-based syntax, rather than a PDF-based syntax. Instances of, typically XML, subassemblies are copied from the original PDF document and expressed as a package within an XDP document. The subassemblies in the original PDF document remain unchanged. The *PDF Reference* [PDF] states that content in XFA packages take precedence over their counterparts embedded within the PDF package. This rules resolves the potential conflict over which content (XDP package or embedded PDF subassembly) an XDP importer should use.



XDP packages the PDF document to comply with XML conventions. The PDF document is enclosed within the XDP as a region of character-encoded content because of the inability for XML to directly enclose binary content. As a result, the XDP contains all of the information that was formerly enclosed within the PDF, though some of the information may now be expressed in XML. All of the information survives the transformation process. Therefore, a PDF document can be transformed into an XDP and subsequently transformed back into a PDF document without loss of information.

A benefit of the XDP format is that PDF documents can now successfully operate directly within XML workflows because the XDP format provides a means for selectively expressing a PDF document in an XML

compatible manner without loss of information. Because the transformations are lossless, document workflows can choose arbitrarily when to process documents in a PDF format vs. when to process the same document in an XML-based format.

Extensibility of XDP and PDF

In addition to providing a format for expressing one or more subassemblies of a PDF document, the XDP format has the capability to host arbitrary XFA components. This capability to host arbitrary content is also a feature of PDF. In particular, XDP is an XML-based format with an open content model; the format itself does not prescribe a closed set of components and can therefore be arbitrarily extended.

XDP Element Language Syntax

This chapter provides a reference for the XDP element. XDP provides a mechanism for packaging units of XFA components within a surrounding XML container.

Note: This document describes the XDP format, but does not describe the transformation mechanism between XDP and PDF nor between XDP any other format.

The XDP format is comprised of a single optional processing instruction ("The xfa Processing Instruction" on page 876) and a single element, known as the `xdp` element ("The xdp Element" on page 883).

The xfa Processing Instruction

The `xfa` processing instruction may be used to hold meta-data about the XDP document. By convention this processing instruction is placed near the beginning of the document, ahead of the `xdp` element.

There are two parameters used within the `xfa` processing instruction. They may occur in any order.

`generator="`*`name`*`"`

The *name* identifies the program and version of that program that generated the document.

`APIVersion="`*`version`*`"`

The *version* identifies the version of the scripting API for which the contained scripts were generated. The XFA 2.5 scripting API corresponds to version 2.5.6290.0.

For example, the following fragment comes from an XDP generated by Adobe LiveCycle Designer 8.0.

Example 23.1 XDP with generator tag

```
<?xml version="1.0" encoding="UTF-8" ?>
<?xfa generator="AdobeLiveCycleDesigner_V8.0" APIVersion="2.5.6290.0"?>
<xdp:xdp ...>
   ...
</xdp:xdp>
```

XDP Namespace

The `xdp` element must belong to the namespace of `http://ns.adobe.com/xdp/`, which is known as the XDP namespace.

The xdp element should make use of explicitly prefixed namespace notation rather than declaring the XDP namespace as a default namespace. If the xdp element declared the XDP namespace as the default namespace it would have the unfortunate side effect of placing any packet that lacks namespace information into the XDP namespace itself.

The following example demonstrates the proper way to declare the XDP namespace.

Example 23.2 XDP with properly declared namespace

```
<xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/">
   <xfa:datasets xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
      <xfa:data>
         <book>
            <ISBN>15536455</ISBN>
            <title>Introduction to XML</title>
            <author>
               <firstname>Charles</firstname>
               <lastname>Porter</lastname>
            </author>
         </book>
      </xfa:data>
   </xfa:datasets>
</xdp:xdp>
```

In the above example the namespace declaration on the xdp element does not impact the default namespace and therefore the "book" fragment does not inadvertently inherit the XDP namespace.

The following example illustrates the *discouraged* practice of an XDP that expresses the XDP namespace as the default namespace.

Example 23.3 XDP with namespace declared using a discouraged method

```
<!-- Declaring the XDP namespace as the default namespace is discouraged --!>
<xdp xmlns="http://ns.adobe.com/xdp/">
   <xfa:datasets xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
      <xfa:data>
         <book>
            <ISBN>15536455</ISBN>
            <title>Introduction to XML</title>
            <author>
               <firstname>Charles</firstname>
               <lastname>Porter</lastname>
            </author>
         </book>
      </xfa:data>
   </xfa:datasets>
</xdp>
```

Important. In the above example, the xdp element is not prefixed and declares its namespace via the namespace attribute syntax of xmlns="http://ns.adobe.com/xdp/". The impact of this approach is that any descendant packet that does not declare a namespace is at risk of inheriting the XDP namespace. Concretely, in this example, the result is that the "book" fragment resides in the XDP namespace, which is problematic because such an element is certainly not a valid element of the XDP format, and downstream XML processors intending to interpret this element may no longer recognize the fragment because it has inadvertently been namespaced.

XDP Packets

The role of an XDP packet is to encapsulate an XFA component.

All child elements of the `xdp` element are considered to be XDP packets. Conversely, an XDP packet must be located as a child element of the `xdp` element. An XDP packet must not belong to the XDP namespace. The application of the XDP namespace on child elements of the `xdp` element is reserved for future use.

This section will describe the particular packets supported by Acrobat 6.0. However, the XDP format is also able to enclose packets that are implementation-defined to a particular processing application. Acrobat 6.0 or other processing applications may ignore such packets.

Consider the following example XDP.

Example 23.4 XDP containing several packets

```
<xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/"
   uuid="…"
   timeStamp="1994-11-05T13:15:30Z">
   <xfa:datasets xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
      <xfa:data>
         <book>
            <ISBN>15536455</ISBN>
            <title>Introduction to XML</title>
            <author>
               <firstname>Charles</firstname>
               <lastname>Porter</lastname>
            </author>
         </book>
      </xfa:data>
   </xfa:datasets>
   <pdf xmlns="http://ns.adobe.com/xdp/pdf/">
      <document>
         <chunk> JVBERi0xLjMKJeTjz9IKNSAwIG9iago8PC9MZW5...
            ZQo+PgpzdHJlYW0KeJylWEtv3DYQvutX8FKgPZj...
            Z/iUBGstoTDg9cfVfPPgcPjJDxUnDH7wt3GCtPv...
         </chunk>
      </document>
   </pdf>
   <my:example xmlns:my="http://www.example.com/">
      <my:message>This packet does not represent a PDF subassembly</my:message>
   </my:example>
</xdp:xdp>
```

The above example XDP encloses the following XDP packets:

- *datasets*. The first packet is represented by the `xfa:datasets` element that encloses the XML form-data subassembly of a PDF form.
- *pdf*. The second packet is represented by the `pdf` element that encloses an encoded PDF form. The PDF object still retains the form-data presented in the first packet; however, the XDP packet version of the form-data takes precedence over the form-data embedded in the PDF object.
- *my:example*. The third packet is represented by the `my:example` element that encloses an XFA component meaningful to the creator of the XDP but does not represent a subassembly of the PDF form.

The above example XDP also uniquely identifies the template (`uuid`) and indicates when the template was last modified (`timeStamp`).

XDP Reference

The config Element (an XDP Packet)

This packet encloses the configuration settings ("Config Specification" on page 761).

The following shows the format of a `config` packet:

```
<xfa:config
   xmlns:xfa="http://www.xfa.org/schema/xci/1.0/">
   XFA and application-specific configuration elements
</xfa:config>
```

Portions of the `config` packet are defined elsewhere in this specification and other portions are application-defined. The `config` MIME-type is `text/xml`.

The connectionSet Element (an XDP Packet)

The `connectionSet` packet describes the connections used to initiate or conduct web services. Such a set defines connections for web services (WSDL), sample data (XML), and schema files (XSD) ("Connection Set Specification" on page 820).

The following shows the format of a `connectionSet` packet:

```
<connectionSet
   xmlns="http://www.xfa.org/schema/xfa-connection-set/2.4/">
   XFA connection set elements
</connectionSet> </xfd>
```

The `connectionSet` MIME-type is `text/xml`.

The datasets Element (an XDP Packet)

The `datasets` element encloses XML data content that may have originated from an Adobe XML form and/or may be intended to be consumed by an Adobe XML form ("Data Description Specification" on page 834). The `datasets` element may also include XML digital signatures, as described "XML Digital Signatures" on page 471.

The following shows the format of a `datasets` packet:

```
<xfa:datasets
   xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
   <xfa:data>
      arbitrary XML data content
   </xfa:data>
   <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
      XMLDSIG digital signature
   </Signature>
</xfa:datasets>
```

The `datasets` MIME Type is `text/xml`.

The localeSet Element (an XDP Packet)

The `localeSet` packet encloses information about locales. A locale set includes predefined set of conventions for representing dates, times, numbers, and currency. For more information, see "Localization and Canonicalization" on page 138 and "Locale Set Specification" on page 794.

```
<xdp xmlns="http://ns.adobe.com/xdp/">
   <localeSet
      xmlns="http://www.xfa.org/schema/xfa-locale-set/2.5/">
      XML locale set content
   </localeSet>
</xdp>
```

The `localeSet` MIME-type is `text/xml`.

The pdf Element (an XDP Packet)

An XDF `pdf` element encloses a PDF packet (MIME-type `application/pdf`).

The PDF packet has the following format:

```
<pdf xmlns="http://ns.adobe.com/xdp/pdf/">
   <document>
      <chunk>
         base64 encoded PDF content
      </chunk>
   </document>
</pdf>
```

XML is a text format, and is not designed to host binary content. PDF files are binary and therefore must be encoded into a text format before they can be enclosed within an XML format such as XDP. The most common method for encoding binary resources into a text format, and the method used by the PDF packet, is *base64 encoding* [RFC2045].

chunk element

The chunk element must enclose a single base64 encoded PDF document. PDF content cannot be broken into smaller chunks; however, the packet may contain processing instructions that explain how to process the embedded PDF.

href

The PDF packet may contain a reference to an external file, as shown in the following example. The value of `href` is a URI to the original copy of the PDF document. The processing application obtains this value from the XFDF F-key path. The F-key path is relative to the system on which the PDF document was created.

Example 23.5 pdf packet making reference to an external file

```
<pdf
   href="pathname/filename.pdf"
   xmlns="http://ns.adobe.com/xdp/pdf/"
/pdf>
```

Note: For security reasons an XFA processor may refuse to process a reference to an external file. Whether it does or not is application and configuration dependent.

The pdf MIME-type is application/pdf.

The signature Element (an XDP Packet)

The signature packet encloses a detached digital signature. Such a signature may be used to establish the integrity of part or all of the data in the XFA Data DOM and to support signer authentication. This XDP packet is not used to save digital signatures that are enveloped in the in the dataset. See "XML Digital Signatures" on page 471.

The signature packet has the following format:

```
<signature xmlns="http://www.w3.org/2000/09/xmldsig#">
   XMLDSIG signature content
</signature>
```

The sourceSet Element (an XDP Packet)

The sourceSet packet contains ADO database queries, used to describe data binding to ADO data sources. The ADO grammar is defined in the *ADO API Reference* [ADO].

The sourceSet packet has the following format:

```
<sourceSet
   xmlns="http://www.xfa.org/schema/xfa-source-set/1.0/">
   XFA source set content
</sourceSet>
```

The sourceSet MIME-type is text/xml.

The stylesheet Element (an XDP Packet)

The stylesheet packet encloses a single XSLT stylesheet. The XSLT packet is expressed with an appropriately namespaced stylesheet element, as defined by the W3C "XSL Transformations" specification [XSLT].

The XDP format may enclose multiple stylesheet packets. Each stylesheet packet should be labelled with an XML ID attribute so that the configuration packet can refer to it individually. For more information see "XSLT Transformations" on page 458.

The following shows the format of a stylesheet packet:

```
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   ID="identifier">
   XSL stylesheet elements
</xsl:stylesheet>
```

The stylesheet MIME-type is text/css.

The template Element (an XDP Packet)

This packet contains the form template, as defined by "Template Specification" on page 482.

The following shows the format of a template packet:

```
<xfa:template
   xmlns:xfa="http://www.xfa.org/schema/xfa-template/2.5/">
   XFA template elements
</xfa:template>
```

The `template` MIME-type is `application/x-xfa-template`.

The xdc Element (an XDP Packet)

The xdc packet encloses application-specific XFA driver configuration instructions. The format of an xdc packet does not have a formal grammar. That is, each implementation of an XDC grammar may be unique.

```
<xsl:xdc
   xmlns:xdc="http://www.xfa.org/schema/xdc/1.0/">
   application-defined XDC elements
</xsl:xdc>
```

There is no standard MIME-type for the `xdc` packet because it is application-specific.

The xdp Element

The xdp element is the root of an XDP document. It encapsulates any number of XFA components, which are packaged as XDP packets.

The format is as follows. The notation `[0..1]` means the packet can occur no more than once. The notation `[0..n]` means the packet can occur any number of times. The packets can occur in any order and in the case of multiply occurring packets can interleave with other packets.

```
<xdp:xdp
   xmlns:xdp="http://ns.adobe.com/xdp/"
   uuid="cdata"
   timeStamp="cdata">
   <config/>  [0..1]
   <connectionSet/>  [0..1]
   <datasets/>  [0..1]
   <localeSet/>  [0..1]
   <pdf/>  [0..1]
   <sourceSet/>  [0..1]
   <stylesheet/>  [0..n]
   <template/>  [0..1]
   <xdc/>  [0..1]
   <xfdf/>  [0..1]
   <xmpmeta/>  [0..1]
   <element containing custom packet/>  [0..n]
</xdp:xdp>
```

The `xdp` element encloses zero or more occurrences of XFA components, each represented as an XDP packet, that is described in "XDP Packets" on page 878. Because the XDP format itself is comprised only of the `xdp` element, the functionality and behavior imparted by an XDP is wholly derived from the packets within the XDP document. It is the packets within the `xdp` element that is of real significance, not the `xdp` element itself. The `xdp` element may also contain any number of custom packets, distinguished from XFA packets by name and namespace.

The `xdp` MIME-type is `text/xml`. It is likely that a future version of this specification will define another MIME-type but if and when that happens XFA processors will still be required to accept `text/html`.

xmlns:xdp

See "XDP Namespace" on page 876.

uuid

A Universally Unique Identifier (UUID), which is assigned by the template designing application and is retained by all applications that subsequently serialize (write-out) the template. The template designing application uses a product-specific algorithm to create the `uuid` value. See "Tracking and Controlling Templates Through Unique Identifiers" on page 460..

The `uuid` attribute inherits the namespace of its container, which in this case is `xdp`.

If the XDP does not contain a `template` packet, the `uuid` attribute is meaningless.

timeStamp

A date-time value that follows the ISO8601 convention: `YYYY-MM-DDTHH:MM:SS z`. The value of timeStamp specifies when the XFA template was last modified, as described in "Tracking and Controlling Templates Through Unique Identifiers" on page 460.

where:

- The capital letter "T" separates the date and time
- "z" represents the time zone designator (Z or +hh:mm or -hh:mm) - following UTC or using local time. For example: 1994-11-05T08:15:30-05:00 corresponds to November 5, 1994, 8:15:30 am, US Eastern Standard Time. Extending this example, 1994-11-05T13:15:30Z corresponds to the same instant.

The `timeStamp` attribute inherits the namespace of its container, which in this case is `xdp`.

If the XDP does not contain a `template` packet, the `timeStamp` attribute is meaningless.

The xfdf Element (an XDP Packet)

The xfdf (annotations) packet encloses collaboration annotations placed upon a PDF document. PDF annotations are converted into XFDF following the rules described in Adobe XFDF format [XFDF].

The format of an `xfdf` packet is as follows:

```
<xfdf xmlns="http://ns.adobe.com/xfdf/"
   xml:space="preserve">
   XFDF elements
</xfdf>
```

The `xfdf` MIME-type is `application/vnd.adobe.xfdf`.

The xmpmeta Element (an XDP Packet)

An XMP packet contains XML representation of PDF metadata. Such metadata includes information about the document and its contents, such as the author's name and keywords, that can be used by search utilities. XMP is described in the XMP Specification [XMPMeta].

The format of an `xmpmeta` packet is as follows:

```
<xmpmeta
   xmlns="http://ns.adobe.com/xmpmeta/"
   xml:space="preserve">
   xmpmeta elements
</xmpmeta>
```

The `xmpmeta` MIME-type is `application/rdf+xml`.

Part 3: Other XFA-Related References

Each chapter in this part contains reference material for non-XML expressions used with XFA. Although the standards described in these chapters are an important part of XFA processing, they are not considered XFA grammars.

24 Canonical Format Reference

This reference describes the canonical format used to represent certain values. *Canonical format* is a locale-agnostic, standardized way to represent date, time, numbers, and currencies.

This appendix describes the set of ISO-8601 date and time forms supported by XFA. It is a a profile of *Data elements and interchange forms — Information interchange — Representation of dates and times* [ISO-8601]. ISO-8601 is the International Standard for the representation of dates and times. ISO-8601 describes a large number of date/time formats.

This specification does not imply any presentation behaviors (such as display or print formatting) of dates or times — it only specifies how the data content should be represented within a document object model (DOM), such as the XFA Data DOM and the XFA Form DOM.

The International Standard for the representation of dates and times is ISO-8601. *A Summary of the International Date and Time Notation,* by Markus Kuhn, provides an introduction to this standard. This document is available at http://www.cl.cam.ac.uk/~mgk25/iso-time.html.

The following sections present the acceptable formats for dates, times, date-times, numbers, and text.

The syntax of each canonical format is described using the following conventions:

- Letters are placeholders for a numeric value.
- Square brackets enclose an optional portion of the format.
- All other characters should be assumed to be literal characters.

Date

Dates must be expressed in any of the following forms:

```
YYYY[MM[DD]]
YYYY[-MM[-DD]]
```

Symbol	Meaning
YYYY	Zero-padded 4-digit year.
MM	Zero-padded 2 digit (01-12) month of the year.
DD	Zero-padded 2 digit (01-31) day of the month.

The above symbols echo the date picture-clause symbols; however, they serve only to describe (to the reader) expected values.

Example 24.1 Date picture clauses

```
19970716
199707
1997
1997-07-16
1997-07
1997
```

Notes

- The only punctuation character that is acceptable between date components is the hyphen character (Unicode character U+002D.
- Not all of these formats constitute a date to the precision of an actual day, hence it is up to the application to determine whether "1997-07" is an acceptable date, i.e. is the application looking for a particular *day*, in which case 1997-07 is not precise enough because it doesn't specify a day.

Time

Time must be expressed in any of the following forms:

```
HH[MM[SS[.FFF][z]]]
HH[MM[SS[.FFF][+HH[MM]]]]
HH[MM[SS[.FFF][-HH[MM]]]]
HH[:MM[:SS[.FFF][z]]]
HH[:MM[:SS[.FFF][-HH[:MM]]]]
HH[:MM[:SS[.FFF][+HH[:MM]]]]
```

Where the symbols have the following meaning:

Symbol	Meaning
HH	Zero-padded 2 digit (00-23) hour of the day, expressed as a 24-hour clock. (The meridiem symbols AM and PM are not supported.)
MM	2-digit (00-59) minute of the hour.
SS	2-digit (00-59) second of the minute.
FFF	Decimal fraction of a second. Any fraction of a second is **always** preceded by a dot (Unicode character U+002E) and must include exactly 3 digits.
z	[ISO-8601] time-zone format: Z, +HH[MM], or -HH[MM]. In the examples on page 889, H is a placeholder for an hour digit, and the M is a placeholder for a minute digit. The value for z may have the following values: • Omitted. If time zone is omitted, states a local time with an unknown time zone. Omitting the time zone information may cause a time data value to be useless to applications that operate within other timezones. Producing time data that omit timezone designators is discouraged. • Z. A time zone of 'Z' (Unicode character U+005A) indicates the time zone is 'zero meridian', or 'Zulu Time'. The [ISO-8601] section titled *Universal Time Coordinated* describes a method of defining time absolutely. Another helpful document is *A Few Facts Concerning GMT, UT, and the RGO*, by R. Langley, 20 January 1999, which is available at http://www.apparent-wind.com/gmt-explained.html. • +HH[MM] or -HH[MM]. A time zone expressed as an offset of plus or minus states that the offset can be added to the time to indicate that the local time zone is HH hours and MM minutes ahead or behind GMT. The plus or minus sign must be included.

The symbols in the above table echo the time picture-clause symbols; however, they serve only to describe (to the reader) expected values.

Example 24.2 Time picture clauses

```
202045.321Z
192045.321-0100
192045.321-01
192045.321
19:20:45
1920
19
```

Notes

- The only punctuation character that is acceptable between the hours, minutes, and seconds components is the colon character (Unicode character u003a).

Date-Time

A date and time specified according to the previous sections can be combined into a single date-time value by concatenating the values together, separated by a 'T' (Unicode character U+0054). The requirement for the 'T' character is a particularly annoying and controversial part of the [ISO-8601] specification; but that's the way it is. If the 'T' is deemed confusing to human users, then the software should take care of transforming the 'T' in and out of existence during read/writes of data.

Example 24.3 Date-time picture clause

```
1997-07-16T20:20:45.4321Z
```

Number

A number literal is a sequence of mostly digits consisting of an integral part, a decimal point, a fractional part, an e (or E) and an optionally signed exponent part.

```
(['+'|'-'])['0'-'9']*('.')(['e'|'E']('+'|'-')['0'-'9']*)
(['+'|'-'])(['0'-'9']*)'.'['0'-'9']*(['e'|'E']('+'|'-')['0'-'9']*)
```

where the symbols in the above expressions are described in "Notational Conventions" on page 895.

Examples of canonical numbers appear below:

```
    1
   +1
 1234
-1234
    1E100
 1234e-4
   -1.e-3
 1234.E+10
    2.1
    +.1234
    -.12e2
```

It is important to distinguish canonical format from issues related to the conversion of a canonical number into a representation specific for a number type, such as integer, float, decimal, and boolean. Such conversion reflects the container's value properties and application-dependent issues such as precision.

Text

The canonical format for text is any sequence of Unicode characters. Note that there is no Unicode character assigned to code point U+0000 and this code point is forbidden in XFA.

25 FormCalc Specification

This document, as part of a family of specifications referred to as the *XML Forms Architecture*, describes an XML based language, *XFA-Template*, for modeling electronic form templates. XFA provides for the specific needs of electronic forms and the applications that use them. XFA addresses the needs of organizations to securely capture, present, move, process, output and print information associated with electronic forms. This document specifically describes a simple calculation language optimized for creating e-form centric logic and calculations.

Grammar and Syntax

This section describes the building blocks that compose FormCalc expressions and how those building blocks can be assembled into such expressions.

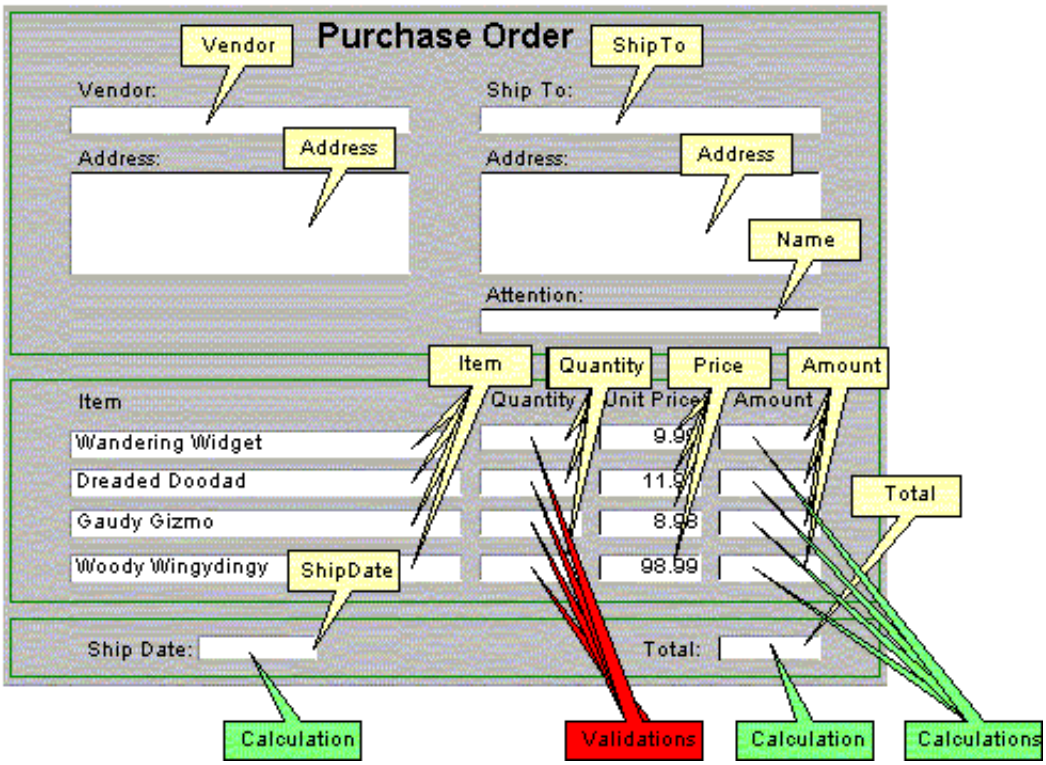
Language Overview

FormCalc is a simple calculation language whose roots lie in electronic form software from Adobe, and common spreadsheet software. It is an expression-based language. It is also a type less language, where values of type string or type number can be promoted to strings, numbers or booleans to suit the context.

FormCalc is tailored to the skills of the non-programmer who is comfortable with spreadsheet-class application software. This user can, with the addition of a few expressions, validate user input and/or unburden the form user from the spreadsheet-like calculations.

To that aim, the language provides a large set of built-in functions to perform arithmetic, and financial tasks. Locale-sensitive date and time functions are provided, as are string manipulation functions.

To better illustrate the capabilities of the **FormCalc** language, we present a simple purchase order application, and focus on those spreadsheet like calculations and validations typically required of such forms.



Down-pointing call-outs indicate all the field names on this form. In the tabular area of the form are four fields called Item, four fields called `Quantity`, four fields called `Price`, and four fields called `Amount`. We will focus on these shortly.

Green up-pointing call-outs indicate fields with embedded calculations, and the red up-pointing call-outs indicate fields with embedded validations.

A subset of the XFA template syntax used to define this purchase order form might be as follows.

Example 25.1 Purchase order form using calculations and validations

```
<xfa>
   <template name="FormCalc Example">
      <subform name="PO">
         <subform name="Table">
            <field name="Item"> ... </field>
            <field name="Quantity">
               <validate>
                  <script>Within($, 0, 19)</script>
               </validate>
            </field>
            <field name="Price"> ... </field>
            <field name="Amount">
               <calculate>
                  <script>Quantity * Price</script>
               </calculate>
            </field>
            <field name="Item"> ... </field>
            <field name="Quantity">
```

```
            <validate>
               <script>Within($, 0, 19)</script>
            </validate>
         </field>
         <field name="Price"> ... </field>
         <field name="Amount">
            <calculate>
               <script>Quantity * Price</script>
            </calculate>
         </field>
         <field name="Item"> ... </field>
         <field name="Quantity">
            <validate>
               <script>Within($, 0, 19)</script>
            </validate>
         </field>
         <field name="Price"> ... </field>
         <field name="Amount">
            <calculate>
               <script>Quantity * Price</script>
            </calculate>
         </field>
         <field name="Item"> ... </field>
         <field name="Quantity">
            <validate>
               <script>Within($, 0, 19)</script>
            </validate>
         </field>
         <field name="Price"> ... </field>
         <field name="Amount">
            <calculate>
               <script>Quantity * Price</script>
            </calculate>
         </field>
      </subform>
      <subform Name="Summary" ...>
         <field name="ShipDate">
            <calculate>
               <script>Num2Date(Date() + 2, DateFmt())</script>
            </calculate>
         </field>
         <field name="Total">
            <calculate>
               <script>Str(Sum(Amount[*]), 10, 2)</script>
            </calculate>
         </field>
      </subform>
   </subform>
</template>
</xfa>
```

Focusing our attention on the contents of the <script> elements, we see text such as the following, all of which are real-world examples of form calculations.

Script	What it does (further explanation follows)
`Within($, 0, 19)`	Ensures that the entered field value is within the range of 0 to 19
`Quantity * Price`	Computes the product of two fields
`Num2Date(Date() + 2, DateFmt())`	Displays a date that is two days hence from the current date
`Str(Sum(Amount[*]), 10, 2)`	Sums all occurrences of the field `Amount`, formats the resulting number to have a precision of two decimal places in a string, and then adds spaces to make the number 10 characters wide

Some of these expressions are continually being re-executed as the user interacts with the form and enters new data.

On each of the four `Quantity` fields is the validation:

```
Within($, 0, 19)
```

This is used to limit the user's input to between 0 and 19 items. Any other value entered in these fields will cause a validation error, requiring the user to modify his input. Here the symbol $ is an identifier that refers to the value of the field to which this form calculation is bound; in this case, the `Quantity` field.

On each of the four `Amount` fields is the calculation:

```
Quantity * Price
```

which multiplies the value of the `Quantity` field by the value of the Price field on that row, and stores the resulting product in the Amount field. Whenever the user changes any of the quantity fields, this calculation is re-executed and the new value is displayed in the corresponding `Amount` field.

Below the column of `Amount` fields is the `Total` field. It contains the calculation:

```
Str(Sum(Amount[*]), 10, 2)
```

This sums all occurrences of the field `Amount`, and formats the resulting number to two decimal places in a string, 10 characters wide. Whenever any of the amount fields change, this calculation is re-executed and a new value is displayed in the `Total` field.

Finally, the field named `ShipDate` also contains a calculation, specifically, a date calculation

```
Num2Date(Date() + 2, DateFmt())
```

This calculation gets the value of the current date (in days), adds 2 days to it and then formats this date value into a locale-sensitive date string. Were that user to be in the United States, in the year 2000, and on the ides of March, the result that would be displayed in the `ShipDate` field, is:

```
Mar 17, 2000
```

A user in Germany, on that same day, would see the following value displayed in the same field.

```
17.03.2000
```

The above is an illustration of the built-in internationalization capabilities of **FormCalc**'s date and time functions.

Admittedly, this is a very simple application. A real-world purchase order form would be significantly more complex, with perhaps several dozen calculations and validations. Hopefully this example will suffice to introduce some of the capabilities of the **FormCalc** language.

We will now proceed to formalize the definition of this language. More complex language examples will be presented throughout.

Grammar

The **FormCalc** language is defined in terms of a context-free grammar. This is a specification of the lexical and syntactic structure of **FormCalc** calculations.

A context-free grammar is defined as a number of productions. Each production has an abstract symbol called a nonterminal as its left-hand side, and a sequence of one or more nonterminal and terminal symbols as its right-hand side. The grammar specifies the set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production in which the nonterminal is the left-hand side.

Notational Conventions

The following convention in notation is used to describe the grammar of **FormCalc**:

Metasymbol	Description	Syntax example
::=	Start of the definition of a nonterminal symbol.	`FormCalculation ::= ExpressionList` defines the production FormCalculation as an ExpressionList symbol.
| *symbol*	Alternative symbol.	`'+' | '-'` allows alternate additive operator symbols.
[*symbol* **]**	One from the set of enclosed symbol(s).	`['E' 'e']` allows one symbol from the set 'E', 'e' of symbols.
[*symbol* **–***symbol* **]**	Range of symbols.	`['0'-'9']` allows one symbol from the consecutive set '0', '1', ..., '9' of symbols.
symbol **\–***symbol*	Set difference of symbols.	`Character \- LineTerminator` allows one symbol from the set of Characters that is not a LineTerminator symbol.
(*symbol* **)**	One occurrence of the enclosed symbol(s).	`( '+' | '-' )` allows for one occurrence of either alternative symbol.
(*symbol* **)***	Zero or more occurrences of the enclosed symbol(s).	`( ',' SimpleExpression )*` allows for zero or more occurrence of the ',' symbol followed by a SimpleExpression symbol.
(*symbol* **)?**	At most one occurrence of the enclosed symbol(s).	`( ArgumentList )?` allows for zero or one occurrence of the ArgumentList symbol.

The nonterminal symbols of the grammar are always in normal print, often sub scripted by a production key, e.g., ProductionName$_{[ProductionKey]}$, as in LogicalAndExpression$_{70}$. The terminal symbols of the grammar are always enclosed in single quotes, as in '=' and 'then'.

Lexical Grammar

This section describes the lexical grammar of the **FormCalc** language. It defines a set of productions, starting from the nonterminal symbol Input$_1$, to describe how sequences of Unicode characters are translated into a sequence of input elements.

The grammar has as its terminal symbols the characters of the Basic Multilingual Plane (BMP) of the [Unicode-2.1] character set; this limitation allows us to hold onto the *"one character = one storage unit"* paradigm the original Unicode standard promised, a bit longer.

Input elements other than white spaces, line terminators and comments form the terminal symbols for the syntactic grammar of **FormCalc**, and are called tokens. These tokens are the literals, identifiers, keywords, separators and operators of the **FormCalc** language.

```
1  Input ::= WhiteSpace | LineTerminator | Comment | Token
```

The source text for a **FormCalc** calculation is a sequence of characters using the Unicode character encoding. These Unicode characters are scanned from left to right, repeatedly taking the longest possible sequence of characters as the next input element.

```
2  Character ::=  [#x9-#xD] | [#x20-#xD7FF] | [#xE000-#xFFFD]
```

Note: Not all **FormCalc** hosting environments recognize these characters, e.g., XML does not allow the vertical tab (#xB) and form feed (#xC) characters as input.

White Space

White space characters are used to separate tokens from each other and improve readability but are otherwise insignificant.

```
3  WhiteSpace ::=  #x9 | #xB | #xC | #x20
```

These are the horizontal tab (#x9), vertical tab (#xB), form feed (#xC), and space (#x20) characters.

Line Terminators

Line terminators, like white spaces are used to separate tokens and improve readability but are otherwise insignificant.

```
4  LineTerminator ::=  #xA | #xD
```

These are the linefeed (#xA), and carriage return (#xD) characters.

Comments

Comments are used to improve readability but are otherwise insignificant.

A comment is introduced with a semi-colon (;) character, or a pair of slash (/) characters, and continues until a line terminator is encountered.

```
5  Comment ::= ';' ( Character \- LineTerminator )* |
      '/' '/' ( Character \- LineTerminator )*
```

Note: "Notational Conventions" on page 895 explains the significance of the * and ? symbols.

String Literals

A string literal is a sequence of Unicode characters enclosed within double quote characters, e.g., "the cat jumped over the fence." The string literal "" defines an empty sequence of text characters called the empty string.

To embed a double quote within a string literal, specify two double quote characters, as in "He said ""She said.""". Moreover within string literals, any Unicode character may be expressed as a Unicode escape sequence of 6 characters consisting of \u followed by four hexadecimal digits, e.g.,

```
"\u0047\u006f \u0066\u0069\u0073\u0068\u0021"
```

To embed a control character with a string literal, specify its Unicode escape sequence, e.g., specify \u000d for a carriage return, and \u000a for a newline character.

```
6   HexDigit ::= ['0'-'9'] | ['m'-'f'] | ['A'-'F']
7   EscapedCharacter ::= '"' '"' |
      '\' 'U' HexDigit HexDigit HexDigit HexDigit |
      '\' 'U' HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit
8   StringLiteral ::= '"' ( Character \- ['"'] | EscapedCharacter )* '"'
```

Note: "Notational Conventions" on page 895 explains the significance of the * and ? symbols.

Number Literals

A number literal is a sequence of mostly digits consisting of an integral part, a decimal point, a fractional part, an e (or E) and an optionally signed exponent part. Either the integral part or the fractional part may be missing, but not both. In the fractional part, either the decimal point or the e and exponent part may be missing, but not both.

```
9   Integer ::= ['0'-'9']+ ( ['0'-'9'] )*
10  Exponent ::= ['E' 'e'] ['+' '-']? Integer
11  NumberLiteral ::= Integer '.' ['0'-'9']* Exponent? | '.' Integer Exponent? |
Integer Exponent? | 'nan' | 'inf'
```

Note: "Notational Conventions" on page 895 explains the significance of the * and ? symbols.

Examples of number literals include 12, 1.2345, .12, 1e-2, and 1.2E+3.

All number literals are internally converted to [IEEE754] 64-bit binary values. However, IEEE 754 values can only represent a finite quantity of numbers.Just as some numbers, such as 1/3, are not representable precisely as decimal fractions, other numbers are not precisely representable as binary fractions. Specifically, but not limited to, number literals having more than 16 significant digits in the non-exponent part will be the rounded to the nearest representable IEEE 754 64-bit value using a round-to-nearest mechanism. The following table provides examples of such rounding imprecision, all of which are conformant to the IEEE 754 standard.

Input number	Is rounded to …
123456789.012345678	123456789.01234567
99999999999999999	100000000000000000

Such rounding behavior behaviour can sometimes lead to surprising results. **FormCalc** provides a function, `Round()`, which returns a given number rounded to a given number of decimal places. When the given number is exactly halfway between two representable numbers, it is rounded away from zero; up if positive, and down if negative.

Example 25.2 Rounding down

```
round(.124, 2)
```

This example returns 0.12.

Example 25.3 Rounding up

```
round(.125, 2)
```

This example returns 0.13.

Example 25.4 Unexpected rounding

Given this convention, one might expect then that

```
round(0.045, 2)
```

returns 0.05. It doesn't. 0.045 cannot be exactly represented in a finite number of bits. IEEE 754 dictates that the number literal

0.045

be approximated to

0.0449999999999999

This approximation is closer to 0.04 than to 0.05, so

```
Round(0.045, 2)
```

returns 0.04. This too is conformant to the IEEE 754 standard.

IEEE 754 64-bit values also support representations like NaN (not a number), +Inf (positive infinity), and -Inf (negative infinity). **FormCalc** does not support these; currently, any intermediate expression that evaluates to NaN, +Inf, or -Inf results in an error exception which is propagated in the remainder of the expression. This behaviour may change in future.

Literals (General)

```
12 NullLiteral ::= 'null'
13 Literal ::= StringLiteral | NumberLiteral| NullLiteral
```

The NullLiteral equates to the null value

Identifiers

An identifier is a sequence of characters of unlimited length but always beginning with an alphabetic character, or an underscore (_) character, or a dollar sign ($) character, or an exclamation mark (!) character.

FormCalc identifiers are case sensitive, i.e., identifiers whose characters only differ in case, are considered distinct. Case sensitivity is mandated by **FormCalc**'s hosting environments.

```
14 Identifier ::= ( AlphabeticCharacter | '_' | '$' | '!' ) (
AlphaNumericCharacter[1] | '_' | '$' )
```

1. An alphabetic character is any Unicode character classified as a letter in the Basic Multilingual Plane (BMP). An alphanumeric character is any Unicode character classified as either a letter or digit in the BMP.

Keywords

Keywords in **FormCalc** are reserved words and are case insensitive. Of these, the 'if', 'then', 'elseif', 'else', 'endif' keywords delimit the parts of an If Expressions.The 'nan' and 'inf' keywords denote special number literals, whereas the 'null' keyword denotes the null literal. The 'this' keyword denotes a specific accessor.The remaining keywords are keyword operators.

The following are keywords and may not be used as identifiers:

```
15 Keyword ::= 'if' | 'then' | 'elseif' | 'else' | 'endif' | 'or' | 'and' |
'not' | 'eq' | 'ne' | 'le' | 'ge' | 'lt' | 'gt' | 'this' | 'null' | 'nan' |
'infinity'
```

```
16 ReservedKeyword ::= | 'while' | 'do' | 'endwhile' | 'end' | 'for' | 'upto' |
'downto' | 'step' | 'endfor' | 'foreach | 'in' | 'break' | 'continue' | 'var' |
'func' | 'endfunc' | 'throw' | 'return' | 'exit'
```

Operators

FormCalc defines a number of operators; they include unary operators, multiplicative operators, additive operators, relational operators, equality operators, logical operators, and the assignment operator.

FormCalc operators are symbols common to most other scripting languages:

```
17 Operator ::= '=' | '|' | '&' | '==' | '<>' | '<=' | '>=' | '<' | '>' | '+' |
'-' | '*' | '/'
```

Several of the **FormCalc** operators have an equivalent mnemonic operator keyword. These keyword operators are useful whenever **FormCalc** expressions are embedded in HTML and XML source text, where symbols <, >, and & have predefined meanings and must be escaped. Here's an enumeration of all **FormCalc** operators, illustrating the symbolic and mnemonic forms of various operators.

```
18 LogicalOrOperator ::= '|' | 'or'
19 LogicalAndOperator ::= '&' | 'and'
20 EqualityOperator ::= '==' | '<>' | 'eq' | 'ne'
21 RelationalOperator ::= '<=' | '>=' | '<' | '>' | 'le' | 'ge' | 'lt' | 'gt'
22 AdditiveOperator ::= '+' | '-'
23 MultiplicativeOperator ::= '*' | '/'
24 UnaryOperator ::= '-' | '+' | 'not'
```

Tokens

```
25 Separator ::= '(' | ')' | '[' | ']' | ',' | '.' | '..' | '.#' | '.*'
26 Token ::= Literal | Keyword | Identifier | Operator | Separator
```

Syntactic Grammar

The syntactic grammar for **FormCalc** has the tokens defined in the preceding lexical grammar as its terminal symbols. It defines the set of productions, starting from the nonterminal symbol FormCalculation, to describe how sequences of tokens can form a syntactically valid calculation.

The following subsections describe the expressions in this syntactic grammar.

```
27 FormCalculation ::= ExpressionList
```

```
28 ExpressionList ::= Expression | ExpressionList Expression

29 Expression ::= IfExpression |
   WhileExpression |
   ForExpression |
   ForEachExpression |
   AssignmentExpression |
   DeclarationExpression |
   SimpleExpression

30 SimpleExpression ::= LogicalOrExpression

31 LogicalOrExpression ::= LogicalAndExpression |
   LogicalOrExpression LogicalOrOperator LogicalAndExpression

32 LogicalAndExpression ::= EqualityExpression |
   LogicalAndExpression LogicalAndOperator EqualityExpression

33 EqualityExpression ::= RelationalExpression |
   EqualityExpression EqualityOperator RelationalExpression

34 RelationalExpression ::= AdditiveExpression |
   RelationalExpression RelationalOperator AdditiveExpression

35 AdditiveExpression ::= MultiplicativeExpression |
   AdditiveExpression AdditiveOperator MultiplicativeExpression

36 MultiplicativeExpression ::= UnaryExpression |
   MultiplicativeExpression MultiplicativeOperator UnaryExpression

37 UnaryExpression ::= PrimaryExpression | UnaryOperator UnaryExpression

38 LogicalOrOperator ::= '|' | 'or'

39 LogicalAndOperator ::= '&' | 'and'

40 EqualityOperator ::= '==' | '<>' | 'eq' | 'ne'

41 RelationalOperator ::= '<=' | '>=' | '<' | '>' | 'le' | 'ge' | 'lt' | 'gt'

42 AdditiveOperator ::= '+' | '-'

43 MultiplicativeOperator ::= '*' | '/'

44 UnaryOperator ::= '-' | '+' | 'not'

45 PrimaryExpression ::= Literal |
     FunctionCall |
     Accessor ( '.*' )? |
     '(' SimpleExpression ')'

46 IfExpression ::= 'if' '(' SimpleExpression ')' 'then' ExpressionList
     ('elseif' '(' SimpleExpression ')' 'then' ExpressionList )*
     ('else' ExpressionList )?
     'endif'
```

```
47 WhileExpression ::=
     'while' '(' SimpleExpression ')' 'do' ExpressionList 'endwhile'

48 ForExpression ::=
     'for' Assignment 'upto' Accessor ('step' SimpleExpression)?
       'do' ExpressionList 'endfor' |
     'for' Assignment 'downto' Accessor ('step' SimpleExpression)?
       'do' ExpressionList 'endfor'

49 ForeachExpression ::=
     'foreach' Identifier 'in' '(' ArgumentList ')'
     'do' ExpressionList 'endfor'

50 BlockExpression ::= 'do' ExpressionList 'end'

51 ContinueExpression ::= 'continue'

52 BreakExpression ::= 'break'

53 ParameterList

54 DeclarationExpression ::=
       'var' Variable |
       'var' Variable '=' SimpleExpression |
       'Func' Identifier '(' ParameterList ')' do ExpressionList 'EndFunc'

55 AssignmentExpression ::= Accessor '=' SimpleExpression

56 FunctionCall ::= Function '(' ( ArgumentList )? ')'

57 Function ::= Identifier

58 Accessor ::= Container | Accessor [ '.' '..' '.#' ] Container

59 Container ::= Identifier | Identifier ' [' '*' ']'
       | Identifier '[' SimpleExpression ']'
       | MethodCall

60 ContainerList ::= Container (',' Container)*

61 MethodCall ::= Method '(' ( ArgumentList )? ')'

62 Method ::= Identifier

63 ArgumentList ::= SimpleExpression (',' SimpleExpression )*
```

Basic Expressions

Expressions Lists

```
64 FormCalculation ::= ExpressionList
65 ExpressionList ::= Expression | ExpressionList Expression
```

A FormCalculation[64] is a list of expressions. Under normal circumstances, each Expression[66] evaluates to a value, and the value of an ExpressionsList[65] is the value of the last expression in the list.

Example 25.5 An expression list containing three expressions

The following FormCalculation evaluates to 50.

```
5 + Abs(Price)     "Hello World"     10 * 3 + 5 * 4
```

Assuming the example is the calculation for a field, after the above expression list is evaluated the value of the associated field is 50.

Simple Expressions

```
66 Expression ::= SimpleExpression | ...
67 SimpleExpression ::= LogicalOrExpression
```

The above grammar for a SimpleExpression[67] is common to conventional languages.

Operator Precedence

Operator precedence rules behave as expected. Enumerating all the **FormCalc** operators in order, from high precedence to lowest precedence yields:

```
= (unary) - + not * / + - < <= > >= lt le gt ge == <> eq ne & and | or
```

Example 25.6 Operator precedence affects expression evaluation

Simple expression	Evaluates to …
`10 * 3 + 5 * 4`	50
`0 and 1 or 2 > 1`	1

Numeric operations on non-numeric operands

When performing numeric operations involving non-numeric operands, the non-numeric operands are first promoted to numbers; if the non-numeric operand can be fully converted to a numeric value then that is its value; otherwise its value is zero (0). When promoting null-valued operands to numbers, their value is always zero.

Example 25.7 Operand conversions affect expression evaluation

Simple expression	Evaluates to …	Explanation
`(5 - "abc") * 3`	15	`"abc"` converts to 0
`"100" / 10`	1	`"100"` converts to 100
`5 + null + 3`	8	`null` converts to 0

Boolean operations on non-Boolean operands

When performing boolean operations on non-boolean operands, the non-boolean operands are first promoted to booleans; if the non-boolean operand can be fully converted to a nonzero value then its value is true (1); otherwise its value is false (0). Null-valued operands are converted to false (0).

Example 25.8 Non-Boolean operands evaluated as Booleans

Simple expression	Evaluates to ...	Explanation
`"abc" | 2`	true (1)	`"abc"` converts to false, and 2 converts to true.
`false | true = true`	true (1)	
`if ("abc") then 10 else 20 endif`	20	`"abc"` converts to false.

String operations on non-string operands

When performing string operations on non-string operands, the non-string operands are first promoted to strings by using their value as a string. When promoting null-valued operands to strings, their value is always the empty string. For example, the following expression evaluates to `"The total is 2 dollars and 57 cents."`

Example 25.9 Non-string operands promoted to strings

```
concat("The total is ", 2, " dollars and ", 57, " cents.")
```

All the intermediate results of numeric expressions are evaluated as double precision IEEE 754 64 bit values. The final result is displayed with up to 11 fractional digits of precision. Should an intermediate expression yield an NaN, +Inf or -Inf,**FormCalc** will currently generate an error exception and propagate that error for the remainder of that expression, and the expression's value will always be zero.

Example 25.10 Illegal expression (divide by zero)

```
3 / 0 + 1
```

FormCalc terminates when an exception is raised.

Logical Or Expressions

```
68 LogicalOrExpression ::= LogicalAndExpression |
     LogicalOrExpression LogicalOrOperator LogicalAndExpression
69 LogicalOrOperator ::= '|' | 'or'
```

A LogicalOrExpression[68] returns the result of a logical disjunction of its operands, or null if both operands are null. If not both null, the operands are promoted to numeric values, and a numeric operation is performed.

The LogicalOrOperators '|' and 'or', represent the same logical-or operator. The logical-or operator returns the boolean result true, represented by the numeric value 1, whenever either operand is not 0 and returns the boolean result false, represented by the numeric value 0, otherwise.

Logical And Expressions

```
70 LogicalAndExpression ::= EqualityExpression |
     LogicalAndExpression LogicalAndOperator EqualityExpression
71 LogicalAndOperator ::= '&' | 'and'
```

A LogicalAndExpression[70] returns the result of a logical conjunction of its operands, or null if both operands are null. If not both null, the operands are promoted to numeric values, and a numeric operation is performed.

The LogicalAndOperators '&' and 'and', both represent the same logical-and operator. The logical-and operator returns the boolean result true, represented by the numeric value 1, whenever both operands are not 0 and returns the boolean result false, represented by the numeric value 0, otherwise.

Equality Expressions

```
72 EqualityExpression ::= RelationalExpression |
     EqualityExpression EqualityOperator RelationalExpression
73 EqualityOperator ::= '==' | '<>' | 'eq' | 'ne'
```

An EqualityExpression$_{72}$returns the result of an equality comparison of its operands.

If either operand is null, then a null comparison is performed. Null valued operands compare identically whenever both operands are null, and compare differently whenever one operand is not null.

If both operands are references ("References" on page 909), then both operands compare identically when they both refer to the same object, and compare differently when they don't refer to the same object.

If both operands are string valued, then a locale-sensitive lexicographic string comparison is performed on the operands. Otherwise, if not both null, the operands are promoted to numeric values, and a numeric comparison is performed.

The EqualityOperators '==' and 'eq', both denote the equality operator. The equality operator returns the boolean result true, represented by the numeric value 1, whenever both operands compare identically and returns the boolean result false, represented by the numeric value 0, otherwise.

The EqualityOperators '<>' and 'ne', both denote the inequality operator. The inequality operator returns the boolean result true, represented by the numeric value 1, whenever both operands compare differently and returns the boolean result false, represented by the numeric value 0, otherwise.

Relational Expressions

```
74 RelationalExpression ::= AdditiveExpression |
     RelationalExpression RelationalOperator AdditiveExpression
75 RelationalOperator ::= '<=' | '>=' | '<' | '>' | 'le' | 'ge' | 'lt' | 'gt'
```

A RelationalExpression$_{74}$returns the result of a relational comparison of its operands.

If either operand is null valued, then a null comparison is performed. Null valued operands compare identically whenever both operands are null and the relational operator is less-than-or-equal or greater-than-or-equal, and compare differently otherwise.

If both operands are string valued, then a locale-sensitive lexicographic string comparison is performed on the operands. Otherwise, if not both null, the operands are promoted to numeric values, and a numeric comparison is performed.

The RelationalOperators '<' and 'lt', both denote the same less-than operator. The less-than-or-equal relational operator returns the boolean result true, represented by the numeric value 1, whenever the first operand is less than the second operand, and returns the boolean result false, represented by the numeric value 0, otherwise.

The RelationalOperators '<=' and 'le', both denote the less-than-or-equal operator. The less-than-or-equal relational operator returns the boolean result true, represented by the numeric value 1, whenever the first operand is less than or equal to the second operand, and returns the boolean result false, represented by the numeric value 0, otherwise.

The RelationalOperators '>' and 'gt', both denote the same greater-than operator. The greater-than relational operator returns the boolean result true, represented by the numeric value 1, whenever the first operand is greater than the second operand, and returns the boolean result false, represented by the numeric value 0, otherwise.

The RelationalOperators '>=' and 'ge', both denote the greater-than-or-equal operator. The greater-than-or-equal relational operator returns the boolean result true, represented by the numeric value 1, whenever the first operand is greater than or equal to the second operand, and returns the boolean result false, represented by the numeric value 0, otherwise.

Additive Expressions

```
76 AdditiveExpression ::= MultiplicativeExpression |
     AdditiveExpression AdditiveOperator MultiplicativeExpression
77 AdditiveOperator ::= '+' | '-'
```

An AdditiveExpression[76] returns the result of an addition (or subtraction) of its operands, or null if both operands are null. If not both null, the operands are promoted to numeric values, and a numeric operation is performed.

The AdditiveOperator '+', is the addition operator; it returns the sum of its operands.

The AdditiveOperator '-', is the subtraction operator; it returns the difference of its operands.

Multiplicative Expressions

```
78 MultiplicativeExpression ::= UnaryExpression |
     MultiplicativeExpression MultiplicativeOperator UnaryExpression
79 MultiplicativeOperator ::= '*' | '/'
```

A MultiplicativeExpression[78] returns the result of a multiplication (or division) of its operands, or null if both operands are null. If not both null, the operands are promoted to numeric values, and a numeric operation is performed.

The MultiplicativeOperator '*', is the multiplication operator; it returns the product of its operands.

The MultiplicativeOperator '/', is the division operator; it returns the quotient of its operands.

Unary Expressions

```
80 UnaryExpression ::= PrimaryExpression | UnaryOperator UnaryExpression
81 UnaryOperator ::= '-' | '+' | 'not'
```

A UnaryExpression[80] returns the result of a unary operation of its operand.

The UnaryOperator '-' denotes the unary minus operator; it returns the arithmetic negation of its operand, or null if its operand is null. If its operand is not null, it is promoted to a numeric value, and the unary minus operation is performed.

The UnaryOperator '+' denotes the unary plus operator; it returns the arithmetic value of its operand, or null if its operand is null. If its operand is not null, it is promoted to a numeric value, and the unary plus operation is performed.

The UnaryOperator 'not' denotes the logical negation operator. it returns the logical negation of its operand. Its operand is promoted to a boolean value, and the logical operation is performed.

The logical negation operation returns the boolean result true, represented by the numeric value 1, whenever its operand is 0, and returns the boolean result false, represented by the numeric value 0, otherwise.

Note: The arithmetic negation of a null operand yields the result null, whereas the logical negation of a null operand yields the boolean result true. This is justified by the common sense statement: If null means nothing then "not nothing" should be something.

Primary Expressions

```
82 PrimaryExpression ::=
       Literal | Accessor | MethodCall | '(' SimpleExpression ')'
```

A PrimaryExpression$_{82}$ is the building block of all simple expressions. It consists of literals, variables, accessors, function calls, and parenthesized simple expressions.

The value of the PrimaryExpression is the value of its constituent Literal$_{13}$, Accessor$_{90}$, SimpleExpression$_{67}$.

Declaration Expressions: Variables and User-Defined Functions

```
83 Expression ::= DeclarationExpression | ...
84 DeclarationExpression ::=
       'var' Variable |
       'var' Variable '=' SimpleExpression |
       'Func' Identifier '(' ParameterList ')' do ExpressionList 'EndFunc'
85 AssignmentExpression ::= Variable '=' Variable | SimpleExpression
```

Assignment Expressions are described on page 907.

Variables and user-defined functions are objects that reside in **FormCalc** storage, as opposed to objects that reside in the processing application's object model. Each variable or function has a *scope*, which is that region of a FormCalculation where the variable is known. The scope of a variable begins at its declaration and persists to the end of a block. See "Block Expressions, Explicit and Implied" on page 915.

Variables

One kind of DeclarationExpression$_{83}$defines a **FormCalc** variable identified by the *Variable* identifier and assigns it the value of the SimpleExpression$_{67}$ if included, or the empty string value if the SimpleExpression is omitted. The value of this kind of DeclarationExpression is the value assigned to the variable.

In the following example, the variable comes into existence and is given the empty string value, before the SimpleExpression is ever evaluated.

Example 25.11 Variable is created and initialized before assignment is performed

Declaration	Is semantically equivalent to the expression list ...
var *Variable* = SimpleExpression	var *Variable Variable* = SimpleExpression

When used on the left-hand side of an AssignmentExpression$_{85}$, the storage contents of the variable identified are modified, and when used in a SimpleExpression$_{67}$, the storage contents of the variable identified are retrieved.

The names of **FormCalc** variables are case sensitive. Thus, in the following valid example, variables A and a coexist.

Example 25.12 Variable names are case-sensitive

```
var A = 1;    declare variable A and assign it the value 1.
var a = 2;    declare variable a and assign it the value 2.
```

User-Defined Functions

Another kind of DeclarationExpression[83] defines a **FormCalc** user-defined function. Such a declaration is identified by the `func` identifier. It also allocates memory for passing parameters to the function and for the expressions bracketed by the `do` and `endfunc` expressions.

The value returned from the function is the last value calculated by the function. That is, there is not return statement, as with C-language functions.

If FormCalc provides a built-in function with the same name as a user-defined function, FormCalc invokes the built-in function.

The following example shows a function being declared and that same function being called.

Example 25.13 Function declared and invoked

```
func MyFunction(param1) do param1*param1 endfunc // Declares a function
MyFunction(3) //Invokes the function, which returns 9
```

Assignment Expressions

```
86 Expression ::= AssignmentExpression  | ...
87 AssignmentExpression ::= Accessor '=' SimpleExpression
```

An AssignmentExpression[87] sets the property identified by the Accessor[90] to the value of the SimpleExpression[67].

The value of the AssignmentExpression[87] is the value of the SimpleExpression[67].

Accessors

```
88 PrimaryExpression ::= Accessor ( '.*' )? | ...
89 AssignmentExpression ::= Accessor '=' SimpleExpression
| ...
90 Accessor ::= Container | Accessor [ '.' '..' '.#' ] Container
91 Container ::= Identifier | Identifier '[' '*' ']'                |
Identifier '[' SimpleExpression ']' | MethodCall
```

FormCalc provides access to object properties and values, which are all described in "Scripting Object Model" on page 73. An Accessor[90] is the syntactic element through which object values and properties are assigned, when used on the left-hand side of an AssignmentExpression[89], or retrieved, when used in a SimpleExpression[67].

Example 25.14 Expression using accessors

```
Invoice.VAT = Invoice.Total * (8 / 100)
```

Accessors may consist of a fully qualified hierarchy of objects, as in:

```
$form.subform.subform.field[10].Price = "255.99"
```

and optionally followed by an object property, as in:

```
Invoice.border.edge[1].color.#value = "255,9,9"
```

The object property is indicated by the use of the '.#' separator.

Accessors may equally consist of a partially qualified hierarchy of objects, again optionally followed by an object property, as in:

```
Invoice..edge[1].color.#value = "255,9,9"
```

The hierarchy is indicated by the use of the '..' separator.

When terminated with the '.*' separator, instead, what is referred to is the collection of sub-objects of the object identified by the accessor.

A container is simply the name of an object or object property.

A hierarchy of objects presupposes the architectural model described in "Scripting Object Model" on page 73. Such a model is important because there can be multiple instances of objects with the same name on a form, each instance gets assigned an occurrence number, starting from zero. To refer to a specific instance of an object which bears the same ambiguous name as other objects, it is required that the name be qualified by an occurrence number corresponding to the desired ordinal instance of the object.

Aside from a referral to the absolute occurrence of an object, there also exists the need to refer to the relative occurrence of an object, and to all occurrences of an object. To that end, **FormCalc** uses the notation:

Notation	**Refers to ...**
`Identifier`	An occurrence of the object that bears the same ordinal occurrence number as the referencing object.
`Identifier[ SimpleExpression ]`	The occurrence of the object identifiedby the runtime value of the expression.
`Identifier[+ ( SimpleExpression )]`	N'th **succeeding** occurrence of the object identified by the runtime value of the expression, relative to the referencing object's occurrence number.
`Identifier[- ( SimpleExpression )]`	N'th **preceding** occurrence of the object identified by the runtime value of the expression, relative to the referencing object's occurrence number.
`Identifier[*]`	Every occurrence of the identified object.

Thus, `Identifier[0]` refers to the first occurrence of the identified object, and by convention,`Identifier[+0]` and `Identifier[-0]` refer to the object whose occurrence number is the same as the referencing object.

The notation `Identifier [ SimpleExpression ]` involves an indexing operation, which must yield a numeric result. If the SimpleExpression[67] operand is non-numeric, then it will be promoted to a number

using the rule for a SimpleExpression; if the non-numeric indexing operand can be fully converted to a numeric value then that is its value; otherwise its value is zero (0), and, when promoting a null-valued indexing operand to a number, its value is always zero.

Some accessor expressions can often evaluate to a set of values, and some built-in functions, such as the following, are designed to accept a set of values.

`Avg()`, `Count()`, `Max()`, `Min()`, `Sum()`, and `Concat()`

However, it is not always possible to determine the exact number of arguments passed to a function at time of compilation. For example, consider the following form calculation `Max(Price[*])`. If there are no occurrences of object `Price`, then the function `Max()` will generate an error exception. If there is a single occurrence of object `Price`, then the function `Max()` will return the value of that object occurrence. If there are multiple occurrences of object `Price`, then the function `Max()` will return the maximum value of all those object occurrences.

For all occurrences of a given object to be included in a calculation, the object must be specified using the [*]-style of accessor referral.

Example 25.15 Legal use of [*]

Expression	Result
`Sum(Price[*])`	Sums all occurrences of object `Price`
`Sum(Price)`	Sums a single occurrence of object `Price`

In most other built-in functions, the description of the formal arguments stipulates that it must be a single value, but it may be that the passed argument evaluates to a set of values. In such circumstances, the function will generate an error exception. This rule applies to all binary and unary operands involving accessors that use the [*]-style of referral.

Example 25.16 Illegal use of [*]

Expression	Result
`Abs(Quantity[*])`	Generates an error exception, irrespective of the number of occurrences of object `Quantity`
`Quantity[*] + 10`	Generates an error exception, respectively of the number of occurrences of object `Quantity`

As noted earlier, the dollar sign ($) character is a valid character in Identifier names ("Identifiers"). However, this specification recommends that processing applications forbid including the dollar sign ($) character in the names of objects and properties — object names and properties containing this character can thus be reserved for special application-defined tasks.

References

In the processing application's model hierarchy, not all objects simply have values. Many only contain sub-objects. It is often useful to manipulate objects indirectly, rather than through explicitly named accessors, particularly if such objects are difficult and/or expensive to locate. The mechanism for indirectly manipulating an object in **FormCalc** is called a reference.

Simply stated, references are handles to existing model objects.

When a reference is:

- assigned to variable,
- passed as an argument to a method or function,
- returned from a method or function,
- compared for equality and inequality, or
- further qualified by an accessor or method,

some special rules apply. In all other contexts where a reference is used, it simply refers to the value of the object it's handling at the time.

Assignment of References

For variable **V**, expression **e** and object **O**, the assignment `var V = e` is evaluated as follows:

- If expression **e** is a reference, variable **V** becomes a reference to the same object referred to by **e**; otherwise, if **e** is a null reference, **V** becomes a variable with the value null;
- If expression **e** is not a reference, and variable **V** is a reference, then the value of the object referred to by variable **V** becomes the value **e**;
- Otherwise, variable **V** becomes the value **e**.

The following table illustrates the evaluation of `var V = e`, given different combinations of referencing. The symbol **&** designates a reference to an object, and ❑ designates the de-referencing of an object then we have the following table:

Variable V	Variable e contains	var V = e evaluates as ...
Reference	Reference to an object	V = &O
	Null value	V = null
	Non-null value	❑V = e
Non-reference	Reference to an object	V = &O
	Null value	V = null
	Non-null value	V = e

The light grey cells denote variables that result in references, whereas, the yellow cells denote variables that result in values.

To summarize then, when a variable is been assigned a valid reference, the variable becomes a reference. It stays a reference until its assigned the null-reference, or goes out of scope. This principle is illustrated in the following sequence.

Example 25.17 Code fragment using assignments of references

```
var q = 5            // variable q is assigned the value 5.
...
q = Ref(ShipDate)    // q is now a reference to object ShipDate.
...
q = 5                // object ShipDate is assigned the value 5.
```

```
q = null            // object ShipDate is assigned the value null.
…
q = Ref(null)       // q is now a null-reference.
…
q = 5               // variable q is assigned the value 5.
```

Passing References to Functions

Passing references as arguments to built-in functions involves the pair-wise assignment of the function's actual arguments to its formal arguments, in which case, the above assignment rules apply.

Example 25.18 Expressions using references as arguments to functions

```
var p = Ref(Price)
var q = Ref(p)
WalkTheDOM(p)
```

Passing References to Methods

Passing references as arguments to methods again involves the pair-wise assignment of the method's actual arguments to its formal arguments, in which case, the above assignment rules apply equally.

Methods are normally defined to take arguments of a given type, but FormCalc references are all untyped, so the user must ensure that references passed as arguments to methods are of the correct type; runtime exceptions will ensue if the user does otherwise.

Example 25.19 Expressions using references to methods

```
var p = Ref(Price)
$.Clone(p)
```

Returning References from Methods and Functions

Returning references from methods and functions either involve the assignment of the return value to a variable or the passing of the return value to another function or method. Both cases are described by the rules above.

Example 25.20 Expressions returning references

```
Ref(Quantity)
FindMySiblings(Ref(Quantity))
```

Comparing References for Equality and Inequality

Comparing references for equality and inequality involves comparing the handles of the referred-to objects, and not their values. It's an object comparison.

Example 25.21 Code fragment using comparison of references

```
var p = Ref(Price)
var q = p
// If Price is a valid object then the above
// references will both refer to the same object.
//    if (p == q) then    ...    endif
```

Qualifying References by an Accessor or Method

References may be further qualified by an accessor or method as in the following examples.

Example 25.22 Expressions using references qualified by an accessor or method

```
var p = Ref(Price)
p.Parent
p.getParent()
```

In addition references may be qualified by an index as follows.

Example 25.23 Expression using a reference qualified by an index

```
var q = Ref(Price[1])
```

The following assignment expressions each set the value of field `Price[1]` to the value of field `Price[2]`.

Example 25.24 Expressions performing the same assignment

```
q = Price[2]
q.#value = Price[2]
q..setValue(Price[2])
```

Other Reference Uses

In all other contexts, a reference simply refers to the value of the object it refers to. This further implies that for most built-in functions, passing a reference argument is equivalent to passing the value of the object the reference refers to. For example, the following expressions both return the absolute value of object `Total`.

Example 25.25 Expressions resolving to the same value

```
Abs(Ref(Total))
Abs(Total)
```

Control Expressions

Break Expressions

```
92 BreakExpression ::= 'break'
```

A break expression causes an immediate exit from the innermost enclosing `while`, `for`, or `foreach` expression loop. Control passes to the expression following the terminated loop.

Example 25.26 Code fragment that sums the receipts up to a maximum value, using continue and break expressions

```
var total = 0.0
foreach receipt in ( travel_receipt[*], parking_receipt[*] ) do
  if (receipt lt 5) then
    continue    // Causes a jump to the next iteration of the foreach loop.
  endif
  total = total + receipt
  if (total gt 1000) then
    total = 1000
    break       // Causes execution to drop out of the foreach loop.
  endif
endfor
```

The value of the break expression is always the value zero (0).

Continue Expression

```
93 ContinueExpression = 'continue'
```

A continue expression causes the next iteration of the innermost enclosing `while`, `for`, or `foreach` loop. When used in a `while` expression, control is passed to the `while` condition. When used in a `for` expression, control is passed to the step expression.

The value of the continue expression is always the value zero (0). See Example 25.26.

If Expressions

```
94 IfExpression ::= 'if' '(' SimpleExpression ')' 'then'
ExpressionList ( 'elseif' '(' SimpleExpression ')' 'then'
ExpressionList )* ( 'else' ExpressionList )? 'endif'
```

An If Expressions is a conditional expression, which, depending upon the value of the SimpleExpression$_{67}$ in the if-part, will either evaluate and return the value of the ExpressionList$_{65}$ in its then-part or, if present, evaluate and return the value of the ExpressionList in its elseif-part or else-part. See "Code blocks determine span of declarations" on page 916.

Syntax

`foreach` *`variable`* `in(` *`argument list`* `)do` *`expression list`* `endfor`

Return

The value of the SimpleExpression$_{67}$ in the if-part is promoted to a boolean value and a logical boolean operation is performed. If this boolean operation evaluates to true (1), the value of the ExpressionList$_{65}$ in the then-part is returned. Otherwise, if there's an elseif-part present, and the value of the SimpleExpression in the elseif-part evaluates to true (1), then the value of its ExpressionList is returned.If there are several elseif-parts, the SimpleExpression of each elseif-part, is evaluated, in order, and if true(1), then the value of its corresponding ExpressionList is returned. Otherwise, the value of the ExpressionList in the else-part is returned; if there is no else-part, the value 0 is returned. In any circumstance, only one of the expression lists is ever evaluated.

For Expressions

```
95 'for' Assignment 'upto' Accessor ('step' SimpleExpression)?
       'do' ExpressionList 'endfor' |
    'for' Assignment 'downto' Accessor ('step' SimpleExpression)?
       'do' ExpressionList 'endfor'
```

A For Expression is a conditionally iterative statement or loop.

The for condition declares and defines a FormCalc variable as the value of the start expression. In the upto variant, the value of the loop variable will iterate from the start expression to the end expression in step expression increments. If you omit the step expression, the step increment defaults to 1. In the downto variant, the value of the loop variable iterates from the start expression to the end expression in step expression decrements. If the step expression is omitted, the step decrements defaults to -1.

Iterations of the loop are controlled by the end expression value. Before each iteration, the end expression is evaluated and compared to the loop variable. If the value is true (1), the expression list is evaluated. After each evaluation, the step expression is evaluated and added to the loop variable. Before each iteration, the

end expression is evaluated and compared to the loop variable. In addition, after each evaluation of the do condition, the step expression is evaluated and subtracted from the loop variable.

A for loop terminates when the start expression has surpassed the end expression. This can be both in an upwards direction, if you use `upto`, or in a downward direction, if you use `downto`.

Example 25.27 Code fragment calculating X to the power Y using a for loop

The following example calculates of the mathematical value of a number raised to the power of another number.

```
var y = 1
for var x = 1 upto power do
   y = y * base
endfor
total=y
```

where `power`, `base`, and `total` are fields in the same subform as the script.

The following example uses the step feature to calculate the sum of all even numbers between 1 and 100.

Example 25.28 Code fragment calculating sum of a series using a for loop

```
var x var evensum=0
for x=2 upto 100 step 2 do evensum=evensum+x endfor
```

For Each Expressions

```
96 ForeachExpression ::=
     'foreach' Identifier 'in' '(' ArgumentList ')'
     'do' ExpressionList 'endfor'
```

A For Each expression iterates over the expression list for each value in its argument list.

The `in` condition, which is executed only once (after the loop variable has been declared) controls the iteration of loop. Before each iteration, the loop variable is assigned successive values from the argument list. The argument list cannot be empty.

Note: Use a comma (,) to separate more than one simple expression in the argurment list.

Return

The value of the last expression list that was evaluated, or zero(0), if the loop was never entered.

Example 25.29 Code fragment calculating sum of columns using a foreach loop

The following example calculates travelling expenses:

```
var total = 0.0
foreach expense in ( travel_exp[*], living_exp[*],parking_exp[*] ) do
   total = total + expense
endfor
```

While Expression

```
97 WhileExpression ::=
     'while' '(' SimpleExpression ')' 'do' ExpressionList 'endwhile'
```

A While Expression is an iterative statement or loop that evaluates a given simple expression. If the result of the evaluation is true (1), FormCalc repeatedly examines the `do` condition and returns the results of the expression lists. If the result is false (0), then control passes to the next statement.

A While Expression is particularly well suited to situations in which conditional repetition is needed. Conversely, situations in which unconditional repetition is needed is often best dealt with using a `for` expression.

Return

The result of the list of expressions associated with the do condition.

Example 25.30 Code fragment computing Pi using a while-do-endwhile loop

The following example shows pi being computed (in an inefficient way) to two decimal places.

```
var i = 0.0
while ( Cos(i) gt Sin(i) ) do
   i = i + .1
endwhile
while ( Cos(i) lt Sin(i) ) do
   i = i - .01
endwhile
i * 4
```

Block Expressions, Explicit and Implied

In **FormCalc**, the regions of a script where variables can be declared constitute a *block*. All but one of these expressions are implicit, which means that the block is a side-affect of another expression. The syntax of the explicit block expression is described in "Block Expression" on page 916

The entire script constitutes one block, but other expressions declare the beginnings and endings of a block, which means that blocks can be nested. The following expressions define blocks.

Expressions that define blocks

Expression	Beginning	End
Block Expression	do	end
If Expressions	then	else | endif (whichever comes first)
	elseif	elseif | else | endif (whichever comes first)
	else	endif (whichever comes first)
For Expressions	do	endfor
For Each Expressions	do	endfor
While Expression	do	endwhile
User-Defined Functions	do	end

This all leads to the following rule regarding variables, and their scope:

- A variable declared within block *A* is only valid within its scope in block *A*.

- If block *B* is nested within block *A*, then a variable valid in block *A* is also valid in block *B* except in a scope of block *B* where that variable has been redeclared.

Consider the following annotated FormCalculation as a valid, though contrived example of the concepts defined above.

Example 25.31 Code blocks determine span of declarations

```
D = 0;       assign object D the value 0.
var A = 1;   declare variable A (scope 1) and assign it the value 1.
var B = 2;   declare variable B (scope 1) and assign it the value 2.
var C = 3;   declare variable C (scope 1) and assign it the value 3.
func MyFunction(param1) do param1*param1 endfunc ; declare a function
if (D < A) then
   var A = -1; re-declare variable A (scope 2)and assign it the value -1.
   func MyFunction(param1) do param1/2 endfunc ; re-declare a function
   MyFunction(A)  ; invoke second user-defined function
   if (A < B) then
      var A = B + 2;re-declare variable A (scope 3) and assign it the value 4.
      D = D + A; assign object D the value 4 (= 0 + 4).
   endif
   var B = -2; re-declare variable B (scope 2) and assign it -the value 2.
   C = A - B; assign variable C in scope 1, the value 1 (= -1 - -2).
   if (A > B) then
      var B = A + 2;re-declare variable B (scope 4) and assign it the value 1.
      D = D + B; assign object D the value 5 (= 4 + 1).
   endif
   var C = A + B; re-declare variable C (scope 2) and assign it the value -3.
   D = D + C; assign object D the value 2.
endif
A + B + C + D; the sum is 6 (= 1 + 2 + 1 + 2).
MyFunction(A)  ; invoke original user-defined function
```

Block Expression

```
98 BlockExpression ::= 'do' ExpressionList 'end'
```

The block expression defines the scope for an expression list. A scope specifies the lifetime of variables that it defines. For example, the following expression returns “3”.

```
var xxx=3 do var xxx=1 end xxx
```

Function and Method Calls

Function Calls

```
99 Expression ::= FunctionCall | ...
100 FunctionCall ::= Function '(' ( ArgumentList )? ')'
101 Function ::= Identifier
102 ArgumentList ::= SimpleExpression (',' SimpleExpression )*
```

FormCalc supports a large set of built-in functions to do arithmetic, financial, logic, date, time, and string operations. It also allows you to define your own functions.

Here's a summary of the key properties of built-in functions:

- Function names are case-insensitive.
- Built-in functions are predefined, but their names are not reserved words: this means that the built-in function Max() will never conflict with an object, object property, or object method named Max.
- Many of the built-in functions have a mandatory number of arguments which can be followed by a optional number of arguments.
- Some built-in functions, accept an indefinite number of arguments. Examples of such functions include: Avg(), Count(), Max(), Min(), Sum(), and Concat().
- Built-in functions take precedence over user-defined functions. That is, if the user defines a function with the same name as a built-in function, the built-in function is executed.

Note: If you write a function that has the same name (ignoring case) as one of the built-in functions, your function is NOT invoked. Rather, the built-in function is invoked. FormCalc provides several undocumented built-in functions: acos, asin, atan, cos, deg2rad, exp, log, pi, pow, rad2deg, sin, sqrt, and tan.

Method Calls

```
103 MethodCall ::= Method '(' ( ArgumentList )? ')'
104 Method ::= Identifier
105 ArgumentList ::= SimpleExpression (',' SimpleExpression )*
```

FormCalc also provides access to object methods, not just objects and object properties. The syntax for Accessors$_{90}$ permits this.Object methods are all described in *Adobe XML Form Object Model Reference* [FOM].

Methods are application-defined operators that act upon objects and their properties; these operators are invoked like a function call, in that arguments may be passed to methods exactly like function calls. The number and type of arguments in each method are prescribed by each object type. Objects of different types will support different methods.

Example 25.32 Code fragment showing method calls

```
v = $.getValue();          // retrieve this referencing object's value.
$host.MessageBox(1, v);    // display it in this host's dialog box.
```

Case Sensitivity

The names of functions and methods are case insensitive, but are not reserved. This means that calculations on forms with objects whose names coincide with the names of functions do not conflict; any object method or function can be called equally.

Argument List

All functions and methods take an ArgumentList$_{105}$, although that list may be empty. The number and type of arguments varies with each function. Some, such as Date() and Time() take no arguments. Others, such as Num2Date() take multiple arguments, the first argument being a number, with the remaining arguments being strings. Many functions accept a variable number of arguments. Leading arguments are mandatory, and trailing arguments are often optional. This maintains the complexity of most functions at a low level. Increased functionality is provided to those users who need it by requiring them to supply the additional arguments.

All arguments in an ArgumentList[105] are evaluated in order, leading arguments first. If the number of mandatory arguments passed to a function is less than the number required, the function generates an error exception.

Many functions require numeric arguments. If any of the passed arguments are non-numeric, they are promoted to numbers. Some function arguments only require integral values; in such cases, the passed arguments are always promoted to integers by truncating the fractional part.

FormCalc Support for Locale

A number of built-in date and time functions are provided, to allow the form designer to do the following:

- Select locale-specific date formats (`DateFmt()` and `LocalDateFmt()`)
- Parse strings into numbers according to locale-specific date formats (`Date2Num()`)
- Get the current date (`Date()`)
- Do basic arithmetic on dates
- Format numbers into strings, according to locale-specific date picture clauses (`Num2Date()`).
- Select locale-specific time formats (`TimeFmt()` and `LocalTimeFmt()`),
- Parse strings into numbers according, to locale-specific time picture clauses (`Time2Num()`),
- Get the current time (`Time()`)
- Do basic arithmetic on times
- Format numbers into strings according to locale-specific time formats (`Num2Time()`).
- Parse ISO-8601 date strings and time strings into numbers (`IsoDate2Num()` and `IsoTime2Num()`).

To properly parse and format a date or time, we need to do the following:

- Prompt the user for dates and times, choosing from a set of conventional styles (date/time format style) and using symbols appropriate for the users locale to represent the chosen style (localized date/time format). For example, the conventional date styles include short, medium, long, and full. Further, the English-language short-style localized date format is “MM/DD/YY”.
- Provide the input parser with a picture clause to use for interpreting user-entered values. Picture clauses serve as a template for converting between user-entered data and canonical data.
- Provide an output formatter with a picture clause to use in formatting data for output.
- Provide the input parser with a locale identifier to us in the above parsing.

Locales

When developing internationalized applications, a **locale** is the standard term used to identify a particular nation (language and/or country). A locale defines (but is not limited to) the format of dates, times, numeric and currency punctuation that are culturally relevant to a specific nation. A properly internationalized application will always rely on the locale to supply it with the format of dates, and times. This way, users operating in their locale will always be presented with the date and time formats they are accustomed to.

“Localization and Canonicalization” on page 138 in the chapter “Exchanging Data Between an External Application and a Basic XFA Form” provides additional information about localization.

Specifying a Locale (Locale Identifier String)

A *locale identifier string* is a unique string representing a locale, as described in "Convention for Explicitly Naming Locale" on page 995 in the "Picture Clause Specification".

Determining Which Locale to Use

FormCalc functions that are influenced by locale consider several sources for locale information. Such functions accept an optional stand-alone locale identifier string. If such an argument is not supplied, FormCalc functions determine the locale to use (called the *prevailing locale*), by examining the following:

1. Locale identifier string enclosed in the picture clause argument ("Locale-Specific Picture Clauses" on page 998)
2. Template field or subform declarations, using the `locale` property.
3. Ambient locale. *Ambient locale* is the system locale declared by the application or in effect at the time the XFA processing application is started. In the event the application is operating on a system or within an environment where a locale is not present, the ambient locale defaults to English United States (`en_US`).

Date Format Styles

FormCalc functions support date format styles. A *date format style* is a locale-independent style of representing date. Supported date styles include short, medium, long, and full. One date style is designated the default. The date/time format styles may be defined in the `localeSet` element, described in "The localeSet Element" on page 150.

The format of dates is governed by an ISO standards body whereby each nation gets to specify the form of its default, short, medium, long, and full date formats. Specifically, the locale (as described in the localeSet element) is responsible for identifying the format of dates that conform to the standards of that nation.

Style	Appearance example (en_US)	Description
short	10/2/70	Short date format styles tend to be purely numeric
medium	10-Feb-70	Medium date format styles specify use of abbreviated month names The medium date style tends to be the default style.
long	February 10, 1970	
full	Thursday, February 10, 1970	Full date format styles tend to include the weekday name

Properly internationalized applications then, will always query the locale for a date format. The form designer has the option of choosing from either the default, short, medium, long or full formats, and will never present to the user a hand-crafted date format. Except for the need of a common format for data interchange, use of hand-crafted date formats are best avoided.

Date Picture Clauses

A **date picture clause** specifies the format for a date. It consists of punctuations, literals, and pattern symbols, e.g., "D/M/YY" is a date picture clause.

For a specification of how to construct date picture clauses, refer to "Date Picture Clauses" on page 1006 within the "Picture Clause Specification".

Example 25.33 Date picture clauses

```
MM/DD/YY
MM/DD/YY
DD.MM.YYYY
DD MMM YYYY
MMMM DD, YYYY
EEEE,' le 'D MMMM, YYYY
```

Specifically, in the default en_US locale, the **default date picture clause** is the following:

```
MMM D, YYYY
```

Date picture clauses are used to format and parse date strings, using the built-in functions `Num2Date()` and `Date2Num()`. All formatting and parsing is strict; when formatting, all literals and punctuations are included, and when parsing, all literals and punctuations must be matched exactly. If the date picture clause is meaningless, no formatting nor parsing is attempted.

Localized Date Formats

Properly internationalized e-forms prompt the user with characters appropriate for the locale. Such localized prompts specify the format in which the user must supply the date, a format that must reflect the picture clause used for input parsing.

The following table provides examples of the prompts used for various locales, depending on the picture clause used to parse the user-provided value.

Note: The prompts used by any locale may change over time. The prompts presented in the following table are currently typical; however, they may change in future years.

Examples of localized date prompts

Picture clause used for input parsing	Localized date format (a localized prompt)		
	English	French as used in Canada	German as used in Switzerland
`"YY/MM/DD"`	YY/MM/DD	aa-MM-jj	jj/MM/tt
`"EEEE, D. MMMM YYYY"`	n/a	EEEE, j. MMMM aaaa	EEEE, t. MMMM jjjj

FormCalc provides several functions that return or process localized date formats. For example, `LocalDateFmt()` returns a localized date format that might be used to prompt a user to enter a date. The resulting localized date format uses the date symbols specific for the locale.

Time Format Styles

FormCalc functions support time format styles, which are a locale-independent style of representing time. Supported date and time styles include short, medium, long, and full. One time style is designated the default. The time format styles may be defined in the localeSet element, described in "The localeSet Element" on page 150.

In much the same way that date format styles are governed by an ISO standards body, so are time formats. Again, each nation gets to specify the form of its default, short, medium, long, and full time formats. The locale is responsible for identifying the format of times that conform to the standards of that nation.

The default time format tends to coincide with the medium time format.

Time Picture Clauses

Specifically, in the default en_US locale, the **default time picture clause** is the following:

```
h:MM:SS A
```

Just as with a date format styles, a **time picture clause** is a shorthand specification to format a time. It consists of punctuations, literals, and pattern symbols, e.g., "HH:MM:SS" is a time picture clause.

For a specification of how to construct time picture clauses, refer to "Time Pictures" on page 1012 in the "Picture Clause Specification".

Example 25.34 Time picture clauses

```
h:MM A
HH:MM:SS
HH:MM:SS 'o''clock' A Z
```

Any time picture clause containing incorrectly specified picture clause symbols, e.g., HHH are invalid. When parsing, time picture clauses with multiple instances of the same pattern symbols, e.g., HH:MM:HH are invalid, as are time picture clauses with conflicting pattern symbols, e.g., h:HH:MM:SS. Time picture clauses with adjacent one letter pattern symbols, e.g., HMS, are inherently ambiguous and should be avoided.

Localized Time Formats

As with localized date formats, properly internationalized e-forms prompt the user with characters appropriate for the locale. Such localized prompts specify the format in which the user must supply the time, a format which must correspond to the picture clause used for input parsing.

Date and Time Values

To do basic arithmetic on dates and times, we introduce the concept of date values and time values. Both of these are of numeric type, but their actual numeric value is implementation defined and thus meaningless in any context other than a date or time function. In other words, a form calculation obtains a date value from a date function, performs some arithmetic on that date value, and only passes that value to another date function. These same rules apply to time values.

Both date values and time values have an associated origin or **epoch** — a moment in time when things began. Any date value prior to its epoch is invalid, as is, any time value prior to its epoch.

The unit of value for all date function is the number of days since the epoch. The unit of value for all time functions is the number of milliseconds since the epoch.

The reference implementation defines the epoch for all date functions such that day 1 is Jan 1, 1900, and defines the epoch for all time functions such that millisecond 1 is midnight, 00:00:00, GMT. This means negative time values may be returned to users in timezones east of Greenwich Mean Time.

Arithmetic Built-in Functions

Abs()

This function returns the absolute value of a given number.

Syntax

```
Abs(n1)
```

Parameters

n1

is the number to evaluate.

Returns

The absolute value or null if its parameter is null.

Example 25.35 Expressions using Abs()

```
Abs(1.03)
```

returns 1.03.

```
Abs(-1.03)
```

returns 1.03.

```
Abs(0)
```

returns 0.

Avg()

This function returns the average of the non-null elements of a given set of numbers.

Syntax

```
Avg(n1 [, n2...])
```

Parameters

n1

is the first number in the set.

n2, ...

are optional additional numbers in the set.

Returns

The average of its non-null parameters, or null if its parameter are all null.

Example 25.36 Expressions using Avg()

Calling Avg() as follows …	Returns
`Avg(Price[0], Price[1], Price[2], Price[3])`	9 if `Price[0]` has a value of 8, `Price[1]` has value 10, and `Price[2]` and `Price[3]` are null
`Avg(Quantity[*])`	9 if `Quantity` has two occurrences with values of 8 and 10
`Avg(Quantity[*])`	null if all occurrences of `Quantity` are null

Ceil()

This function returns the whole number greater than or equal to a given number.

Syntax

```
Ceil(n1)
```

Parameters

n1

is the number to evaluate.

Returns

The ceiling or null if its parameter is null.

Example 25.37 Expressions using Ceil()

```
Ceil(1.9)
```

returns 2.

```
Ceil(-1.9)
```

returns -1.

```
Ceil(A)
```

is 100 if the value A is 99.999

Count()

This function returns the count of the non-null elements of a given set of numbers.

Syntax

```
Count(n1 [, n2...])
```

Parameters

n1

is the first argument to count.

n2, ...

are optional additional arguments in the set.

Returns

The count.

Example 25.38 Expressions using Count()

```
Count(5, "ABCD", "", null)
```

returns 3.

```
Count(Quantity[*])
```

returns the number of occurrences of `Quantity` that are non-null, and returns 0 if all of occurrences of `Quantity` are null.

Floor()

This function returns the largest whole number that is less than or equal to a given value.

Syntax

```
Floor(n1)
```

Parameters

n1

is the number to evaluate.

Returns

The floor or null if its parameter is null.

Example 25.39 Expressions using Floor()

```
Floor(6.5)
```

returns 6.

```
Floor(7.0)
```

returns 7.

```
Floor(Price)
```

returns 99 if the value of `Price` is 99.999.

Max()

This function returns the maximum value of the non-null elements of a given set of numbers.

Syntax

```
Max(n1 [, n2...])
```

Parameters

n1

is the first number in the set.

n2, ...

are optional additional numbers in the set.

Returns

The maximum of its non-null parameters, or null if all its parameters are null.

Example 25.40 Expressions using Max()

Calling Max() as follows …	Returns
`Max(Price[*], 100)`	The maximum value of all occurrences of the object Price or 100, whichever is greater
`Max(7, 10, null, -4, 6)`	10
`Max(null)`	null

Min()

This function returns the minimum value of the non-null elements of a given set of numbers.

Syntax

```
Min(n1 [, n2...])
```

Parameters

n1

is the first number in the set.

n2, ...

are optional additional numbers in the set.

Returns

The minimum of its non-null parameters, or null if all its parameters are null.

Example 25.41 Expressions using Min()

Calling Min() as follows …	Returns
`Min(7, 10, null, -4, 6)`	-4
`Min(Price[*], 100)`	The minimum value of all occurrences of the object Price or 100, whichever is less
`Min(null)`	null

Mod()

This function returns the modulus of one number divided by another.

Syntax

```
Mod(n1, n2)
```

Parameters

n1

is the dividend number.

n2

is the divisor number.

Returns

The modulus or null if any of its parameter are null.

The modulus is the remainder of the implied division of the dividend and the divisor. The sign of the remainder always equals the sign of the dividend.

For integral operands, this is simple enough. For floating point operands, the floating point remainder `r` of `Mod(n1, n2)` is defined as `r = n1 - (n2 * q)` where `q` is an integer whose sign is negative when `n1 / n2` is negative, and positive when `n1 / n2` is positive, and whose magnitude is the largest integer less than the quotient `n1 / n2`.

If the divisor is zero, the function generates an error exception.

Example 25.42 Expressions using Mod()

Calling Mod() as follows …	Returns
`Mod(64, 2)`	0.
`Mod(-13, 3)`	-1
`Mod(13, -3)`	1
`Mod(-13.6, 2.2)`	-0.4

Round()

This function returns a number rounded to a given number of decimal places.

Syntax

```
Round(n1 [, n2])
```

Parameters

n1

is the number to be evaluated.

n2

is the number of decimal places.

If n2 is omitted, 0 is used as the default.

If n2 is greater than 12, 12 is used as the maximal precision.

Returns

The rounded value or null if any of its parameters are null.

Example 25.43 Expressions using Round()

Calling Round() as follows …	Yields
`Round(33.2345, 3)`	33.235
`Round(20/3, 2)`	6.67
`Round(-1.3)`	-1
`Round(Price, 2)`	2.33 if the value of the object Price is 2.3333

Sum()

This function returns the sum of the non-null elements of a given set of numbers.

Syntax

```
Sum(n1 [, n2...])
```

Parameters

n1

is the first number to sum.

n2, ...

are optional additional numbers in the set.

Returns

The sum of its non-null parameters, or null if all of its parameters are null.

Example 25.44 Expressions using Sum()

Calling Sum() as follows ...	Returns
`Sum(1, 2, 3, 4)`	10
`Sum(Amount[*])`	The sum of all occurrences of the object `Amount`
`Sum(Amount[2], Amount[3])`	The sum of two occurrences of the object `Amount`

Date And Time Built-in Functions

Date()

This function returns the current system date as the number of days since the epoch.

Syntax

```
Date()
```

Returns

The number of days for the current date.

Example 25.45 Expressions using Date()

```
Date()
```

returns 35733 on Oct 31 1998.

Date2Num()

This function returns the number of days since the epoch, given a date string.

Syntax

```
Date2Num(d1[, f1[, k1]])
```

Parameters

d1

is a date string in the format given by f1, governed by the locale given by k1.

f1

is a date picture clause. If f1 is omitted, the default picture clause is used.

k1

is a locale identifier string, as described in "Specifying a Locale (Locale Identifier String)" on page 919. If k1 is omitted, the prevailing locale is used.

Returns

The days since the epoch or null if any of its parameters are null.

If the given date is not in the format given, or the picture clause is invalid, or the locale is invalid, the function returns 0.

Sufficient information must be provided to determine a unique day since the epoch: if any of the day of the year and year of the era are missing, or any of the day of the month, month of the year and year of the era are missing, the function returns 0.

Example 25.46 Expressions using Date2Num()

Calling Date2Num as follows ...	Returns
`Date2Num("Mar 15, 1996")`	35138
`Date2Num("1/1/1900", "D/M/YYYY")`	1
`Date2Num("03/15/96", "MM/DD/YY")`	35138
`Date2Num("Aug 1,1996", "MMM D,YYYY")`	35277
`Date2Num("31-ago-96", "DD-MMM-YY", "es_ES")`	35307
`Date2Num ("1/3/00", "D/M/YY") - Date2Num("1/2/00", "D/M/YY")`	29, year 2000 being a leap year!

See Also

Num2Date() and DateFmt()

DateFmt()

This function returns a date picture clause, given a date format style.

Syntax

```
DateFmt([n1[, k1]])
```

Parameters

n1

An integer identifying the date format style, whose value has the following meaning:

Value supplied in first argument	Requests the locale-specific style ...
0 (default)	Default style
1	Short style
2	Medium style
3	Long style
4	Full style

k1

is a locale identifier string, as described in "Specifying a Locale (Locale Identifier String)" on page 919. If k1 is omitted, the prevailing locale is used.

Returns

The date picture clause or null if any of its mandatory parameters are null.

If the given date format style is invalid, the function returns default-style date picture clause.

Example 25.47 Expressions using DateFmt()

Calling DateFmt() as follows ...	Returns
DateFmt()	"MMM D, YYYY", which is the default date picture clause
DateFmt(1)	"M/D/YY"
DateFmt(2, "fr_CA")	"YY-MM-DD"
DateFmt(3, "de_DE")	"D. MMMM YYYY"
DateFmt(4, "es_ES")	"EEEE D' de 'MMMM' de 'YYYY"

See Also

Num2Date(), Date2Num(), and LocalDateFmt()

IsoDate2Num()

This function returns the number of days since the epoch, given an [ISO-8601] date string.

Syntax

```
IsoDate2Num(d1)
```

Parameters

d1

is a canonical date string in one of the following two formats:

YYYY[MM[DD]]

YYYY[-MM[-DD]]

or, is an ISO-8601 date-time string — the concatenation of an ISO-8601 date string with an ISO-8601 time string, separated by the character T, as in:

1997-07-16T20:20:20

Returns

The days from the epoch or null if its parameter is null.

If the given date is not in one of the accepted formats, the function returns 0.

Example 25.48 Expressions using IsoDate2Num()

Calling IsoDate2Num() as follows	Returns
`IsoDate2Num("1900")`	1
`IsoDate2Num("1900-01")`	1
`IsoDate2Num("1900-01-01")`	1
`IsoDate2Num("19960315T20:20:20")`	35138
`IsoDate2Num("2000-03-01") - IsoDate2Num("20000201")`	29

See Also

`IsoTime2Num()` and `Num2Date()`

IsoTime2Num()

This function returns the number of milliseconds since the epoch, given an [ISO-8601] time string.

Syntax

```
IsoTime2Num(d1)
```

Parameters

d1

is a canonical time string in one of the following formats:

```
HH[MM[SS[.FFF][z]]]
HH[MM[SS[.FFF][+HH[MM]]]]
HH[MM[SS[.FFF][-HH[MM]]]]
HH[:MM[:SS[.FFF][z]]]
HH[:MM[:SS[.FFF][-HH[:MM]]]]
HH[:MM[:SS[.FFF][+HH[:MM]]]]
```

or, is an ISO-8601 date-time string — the concatenation of an ISO-8601 date string with an ISO-8601 time string, separated by the character T, as in:

1997-07-16T20:20:20

Returns

The number of milliseconds from the epoch or null if its parameter is null.

If the time string does not include a timezone, the current timezone is used.

If the given time is not in a valid format, the function returns 0.

Example 25.49 Expressions using IsoTime2Num()

Calling IsoTime2Num() as follows …	Returns
`IsoTime2Num("00:00:00Z")`	1
`IsoTime2Num("13")`	64800001 to a user in Boston
`IsoTime2Num("13:13:13")`	76393001 to a user in California
`IsoTime2Num("19111111T131313+01")`	43993001

See Also

`IsoDate2Num()` and `Num2Time()`

LocalDateFmt()

This function returns a string containing a localized date format, given a date format style.

Syntax

```
LocalDateFmt([n1[, k1]])
```

Parameters

n1

is an integer identifying the date format style. The following table describes the possible values for n1.

n1	Style requested for the localized date format
0	Locale-specific default style
1	Locale-specific short style
2	Locale-specific medium style
3	Locale-specific long style
4	Locale-specific full style

k1

is a locale identifier string, as described in "Specifying a Locale (Locale Identifier String)" on page 919.

If k1 is omitted, the prevailing locale is used.

Returns

The localized date format or null if any of its parameters are null.

If the given format style is invalid, the function returns default-style localized date format.

The date picture clauses returned by this function are not usable in the functions Date2Num() and Num2Date().

Example 25.50 Expressions using LocaleDateFmt()

Calling LocaleDateFmt() as follows ...	Returns
LocalDateFmt(1, "de_DE")	"tt.MM.jj"
LocalDateFmt(2, "fr_CA")	"aa-MM-jj"
LocalDateFmt(3, "de_CH")	"t. MMMM uuuu"
LocalDateFmt(4, "es_ES")	"EEEE t de MMMM de uuuu"

See Also

DateFmt()

LocalTimeFmt()

This function returns a localized time format, given a time format style.

Syntax

```
LocalTimeFmt([n1[, k1]])
```

Parameters

n1

is an integer identifying the time format style as follows:

n1	Style requested for the localized time format
0	Locale-specific default style
1	Locale-specific short style
2	Locale-specific medium style
3	Locale-specific long style
4	Locale-specific full style

If n1 is omitted, the default style value 0 is used.

k1

is a locale identifier string, as described in "Specifying a Locale (Locale Identifier String)" on page 919.

If k1 is omitted, the prevailing locale is used.

Returns

The localized time format or null if any of its parameters are null.

If the given format style is invalid, the function returns default-style localized time format.

The time picture clauses returned by this function are not usable in the functions `Time2Num()` and `Num2Time()`.

Example 25.51 Expressions using LocalTimeFmt()

Calling LocalTimeFmt() as follows ...	Returns
`LocalTimeFmt(1, "de_DE")`	"HH:mm"
`LocalTimeFmt(2, "fr_CA")`	"HH:mm:ss"
`LocalTimeFmt(3, "de_DE")`	"HH:mm:ss z"
`LocalTimeFmt(4, "es_ES")`	"hhHmm'ss" z".

See Also

`TimeFmt()`

Num2Date()

This function returns a date string, given a number of days since the epoch.

Syntax

```
Num2Date(n1 [,f1 [, k1]])
```

Parameters

n1

is the number of days.

f1

is a date picture clause.

If f1 is omitted, the default picture clause is used.

k1

is a locale identifier string, as described in "Specifying a Locale (Locale Identifier String)" on page 919.

If k1 is omitted, the prevailing locale is used.

Returns

The date string or null if any of its parameters are null.

The formatted date is in the format given in f1, governed by the locale given in k1.

If the given date is invalid, the function returns an empty string.

Example 25.52 Expressions using Num2Date()

```
Num2Date(1, "DD/MM/YYYY")
```

returns "01/01/1900".

```
Num2Date(35139, "DD-MMM-YYYY", "de_CH")
```

returns "16-Mrz-1996".

```
Num2Date(
  Date2Num("31-ago-98",
     "DD-MMM-YY", "es_ES") - 31, "D' de 'MMMM' de 'YYYY", "pt_BR")
```

returns "31 de Julho de 1998".

See Also

Date2Num(), DateFmt() and Date()

Num2GMTime()

This function returns a GMT time string, given a number of milliseconds from the epoch.

Syntax

```
Num2GMTime(n1 [,f1 [, k1]])
```

Parameters

n1

is the number of milliseconds.

f1

is a time picture clause, as defined above.

If f1 is omitted, the default time picture clause is used.

k1

is a locale identifier string, as described in "Specifying a Locale (Locale Identifier String)" on page 919.

If k1 is omitted, the prevailing locale is used.

Returns

The GMT time string or null if any of its parameters are null.

The formatted time is in the format given in f1, governed by the locale given in k1.

The locale is used to format any timezone names.

If the given time is invalid, the function returns an empty string.

Example 25.53 Expressions using Num2GMTime()

```
Num2GMTime(1, "HH:MM:SS")
```

returns "00:00:00".

```
Num2GMTime(65593001, "HH:MM:SS Z")
```

returns "18:13:13 GMT".

```
Num2GMTime(43993001, TimeFmt(4, "de_CH"), "de_CH")
```

returns "12.13 Uhr GMT".

See Also

Num2Time()

Num2Time()

This function returns a time string, given a number of milliseconds from the epoch.

Syntax

```
Num2Time(n1 [,f1 [, k1]])
```

Parameters

n1

is the number of milliseconds.

f1

is a time picture clause, as defined above.

If f1 is omitted, the default time picture clause is used.

k1

is a locale identifier string, as described in "Specifying a Locale (Locale Identifier String)" on page 919.

If k1 is omitted, the prevailing locale is used.

Returns

The time string or null if any of its parameters are null.

The formatted time is in the format given in f1, governed by the locale given in k1.

The locale is used to format any timezone names.

If the given time is invalid, the function returns an empty string.

Example 25.54 Expressions using Num2Time()

Calling Num2Time() as follows ...	Returns
`Num2Time(1, "HH:MM:SS")`	"00:00:00" in Greenwich, England and "09:00:00" in Tokyo
`Num2Time(65593001, "HH:MM:SS Z")`	"13:13:13 EST" in Boston
`Num2Time(65593001, "HH:MM:SS Z", "de_CH")`	"13:13:13 GMT-05:00" to a German Swiss user in Boston
`Num2Time(43993001, TimeFmt(4, "de_CH"), "de_CH")`	"13.13 Uhr GMT+01:00" to a user in Zurich
`Num2Time(43993001, "HH:MM:SSzz")`	"13:13+01:00" to that same user in Zurich

See Also

`Date2Num()`, `DateFmt()` and `Date()`

Time()

This function returns the current system time as the number of milliseconds since the epoch.

Syntax

```
Time()
```

Returns

The number of milliseconds for the current time.

Example 25.55 Expressions using Time()

```
Time()
```

returns 61200001 at precisely noon to a user in Boston.

Time2Num()

This function returns the number of milliseconds since the epoch, given a time string.

Syntax

```
Time2Num(d1[, f1[, k1]])
```

Parameters

d1

is a time string in the format given by f1, governed by the locale given by k1.

f1

is a time picture clause, as defined above.

If f1 is omitted, the default time picture clause is used.

k1

is a locale identifier string, as described in "Specifying a Locale (Locale Identifier String)" on page 919.

If k1 is omitted, the prevailing locale is used.

Returns

The milliseconds from the epoch or null if any of its parameters are null.

If the time string does not include a timezone, the current timezone is used.

The locale is used to parse any timezone names.

If the given time is not in the format given, or the format is invalid, or the locale is invalid, the function returns 0.

Sufficient information must be provided to determine a second since the epoch: if any of the hour of the meridiem, minute of the hour, second of the minute, and meridiem are missing, or any of the hour of the day, minute of the hour, and second of the minute are missing, the function returns 0.

Example 25.56 Expressions using Time2Num()

```
Time2Num("00:00:00 GMT", "HH:MM:SS Z")
```

returns 1.

```
Time2Num("1:13:13 PM")
```

returns 76393001 to a user in California on Standard Time, and 76033001 when that same user is on Daylight Savings Time.

```
(Time2Num("13:13:13", "HH:MM:SS") - Time2Num("13:13:13 GMT", "HH:MM:SS Z")) / (60 * 60 * 1000)
```

returns 8 to a user in Vancouver and returns 5 to a user in Ottawa when on Standard Time. On Daylight Savings Time, the returned values are returns 7 and 4, respectively.

```
Time2Num("1.13.13 dC GMT+01:00", "h.MM.SS A Z", "it_IT")
```

returns 43993001.

See Also

`Num2Time()`, and `TimeFmt()`.

TimeFmt()

This function returns a time format given a time format style.

Syntax

```
TimeFmt([n1[, k1]])
```

Parameters

n1

is an integer identifying the time format style, whose value has the following meaning:

n1	Style requested for the localized time format
0	Locale-specific default style
1	Locale-specific short style
2	Locale-specific medium style
3	Locale-specific long style
4	Locale-specific full style

If n1 is omitted, the default style value 0 is used.

k1

is a locale identifier string, as described in "Specifying a Locale (Locale Identifier String)" on page 919.

If k1 is omitted, the prevailing locale is used.

Returns

The time format or null if any of its parameters are null.

If the given format style is invalid, the function returns default-style time format.

Example 25.57 Expressions using TimeFmt()

Calling TimeFmt() as follows ...	Returns
`TimeFmt()`	"h:MM:SS A"
`TimeFmt(1)`	"h:MM A"
`TimeFmt(2, "fr_CA")`	"HH:MM:SS"
`TimeFmt(4, "de_DE")`	"H.MM' Uhr 'Z"

See Also

`Time()`.

Financial Built-in Functions

Note: The value of the results in the examples of this section may vary slightly from platform to platform. The numbers shown here have all been rounded for presentation purposes. A number followed by a superscript asterisk (*) indicates a rounded return value.

Apr()

This function returns the annual percentage rate for a loan.

Syntax

```
Apr(n1, n2, n3)
```

Parameters

n1

is the principal amount of the loan.

n2

is the payment on the loan.

n3

is the number of periods.

Returns

The annual percentage rate or null if any of its parameters are null.

If any of n1, n2, or n3 are non-positive, the function generates an error exception.

Example 25.58 Expressions using Apr()

```
Apr(35000, 269.50, 30 * 12)
```

returns 0.085^* (8.5%) which is the annual interest rate on a loan of $35,000 being repaid at $269.50 per month over 30 years.

CTerm()

This function returns the number of periods needed for an investment earning a fixed, but compounded, interest rate to grow to a future value.

Syntax

```
CTerm(n1, n2, n3)
```

Parameters

n1

is the interest rate per period.

n2

is the future value of the investment.

n3

is the amount of the initial investment.

Returns

The number of periods or null if any of its parameters are null.

If any of n1, n2, or n3 are non-positive, the function generates an error exception.

Example 25.59 Expressions using CTerm()

```
CTerm(.02, 200, 100)
```

returns 35.00*, which is the required period for $100 invested at 2% to grow to $200.

FV()

This function returns the future value of periodic constant payments at a constant interest rate.

Syntax

```
FV(n1, n2, n3)
```

Parameters

n1

is the amount of each equal payment.

n2

is the interest rate per period.

n3

is the total number of periods.

Returns

The future value or null if any of its parameters are null.

If n1 or n3 are non-positive, or if n2 is negative, the function generates an error exception.

If n2 is 0, the function returns the product of n1 and n3, i.e., the payment amount multiplied by the number of payments.

Example 25.60 Expressions using FV()

```
FV(100, .075 / 12, 10 * 12)
```

returns 17793.03*, which is the amount present after paying $100 a month for 10 years in an account bearing an annual interest of 7.5%.

```
FV(1000, 0.01, 12)
```

returns 12682.50*.

IPmt()

This function returns the amount of interest paid on a loan over a period of time.

Syntax

```
IPmt(n1, n2, n3, n4, n5)
```

Parameters

n1

is the principal amount of the loan.

n2

is the annual interest rate.

n3

is the monthly payment.

n4

is the first month of the computation.

n5

is the number of months to be computed.

Returns

The interest amount or null if any of its parameters are null.

If any of n1, n2, or n3 are non-positive, the function generates an error exception.

If n4 or n5 are negative, the function generates an error exception.

If the payment is less than the monthly interest load, the function returns 0.

Example 25.61 Expressions using IPmt()

```
IPmt(30000, .085, 295.50, 7, 3)
```

returns 624.88* which is the amount of interest paid starting in July (month 7) for 3 months on a loan of $30,000.00 at an annual interest rate of 8.5% being repaid at a rate of $295.50 per month.

NPV()

This function returns the net present value of an investment based on a discount rate, and a series of periodic future cash flows.

Syntax

```
NPV(n1, n2 [, ...])
```

Parameters

n1

is the discount rate over one period.

n2, ...

are the cash flow values which must be equally spaced in time and occur at the end of each period.

Returns

The net present value rate or null if any of its parameters are null.

The function uses the order of the values n2, ... to interpret the order of the cash flows. Ensure payments and incomes are specified in the correct sequence.

If n1 is non-positive, the function generates an error exception.

Example 25.62 Expressions using NPV()

```
NPV(0.15, 100000, 120000, 130000, 140000, 50000)
```

returns 368075.16* which is the net present value of an investment projected to generate $100,000, $120,000, $130,000, $140,000 and $50,000 over each of the next five years and the rate is 15% per annum.

```
NPV(0.10, -10000, 3000, 4200, 6800)
```

returns 1188.44*.

```
NPV(0.08, 8000, 9200, 10000, 12000, 14500)
```

returns 41922.06*.

Pmt()

This function returns the payment for a loan based on constant payments and a constant interest rate.

Syntax

```
Pmt(n1, n2, n3)
```

Parameters

n1

is the principal amount of the loan.

n2

is the interest rate per period.

n3

is the number of payment periods.

Returns

The loan payment or null if any of its parameters are null.

If any of n1, n2, or n3 are non-positive, the function generates an error exception.

Example 25.63 Expressions using Pmt()

```
Pmt(30000.00, .085 / 12, 12 * 12)
```

returns 333.01*, which is the monthly payment for a loan of a $30,000, borrowed at a yearly interest rate of 8.5%, repayable over 12 years (144 months).

```
Pmt(10000, .08 / 12, 10)
```

returns 1037.03*, which is the monthly payment for a loan of a $10,000 loan, borrowed at a yearly interest rate of 8.0%, repayable over 10 months.

PPmt()

This function returns the amount of principal paid on a loan over a period of time.

Syntax

```
PPmt(n1, n2, n3, n4, n5)
```

Parameters

n1

is the principal amount of the loan.

n2

is the annual interest rate.

n3

is the monthly payment.

n4

is the first month of the computation.

n5

is the number of months to be computed

Returns

The principal paid or null if any of its parameters are null.

If any of n1 , n2 , or n3 are non-positive, the function generates an error exception.

If n4 or n5 are negative, the function generates an error exception.

If payment is less than the monthly interest load, the function generates an error exception.

Example 25.64 Expressions using PPmt()

```
PPmt(30000, .085, 295.50, 7, 3)
```

returns 261.62*, which is the amount of principal paid starting in July (month 7) for 3 months on a loan of $30,000 at an annual interest rate of 8.5%, being repaid at $295.50 per month. The annual interest rate is used in the function because of the need to calculate a range within the entire year.

PV()

This function returns the present value of an investment of periodic constant payments at a constant interest rate.

Syntax

```
PV(n1, n2, n3)
```

Parameters

n1

is the amount of each equal payment.

n2

is the interest rate per period.

n3

is the total number of periods.

Returns

The present value or null if any of its parameters are null.

If any of n1 and n3 are non-positive, the function generates an error exception.

Example 25.65 Expressions using PV()

```
PV(1000, .08 / 12, 5 * 12)
```

returns 49318.43* which is the present value of $1000.00 invested at 8%per annum for 5 years.

```
PV(500, .08 / 12, 20 * 12)
```

returns 59777.15*.

Rate()

This function returns the compound interest rate per period required for an investment to grow from present to future value in a given period.

Syntax

```
Rate(n1, n2, n3)
```

Parameters

n1

is the future value.

n2

is the present value.

n3

is the total number of periods.

Returns

The compound rate or null if any of its parameters are null.

If any of n1, n2 , or n3 are non-positive, the function generates an error exception.

Example 25.66 Expressions using Rate()

```
Rate(110, 100, 1)
```

returns 0.10 which is what the rate of interest must be for and investment of $100 to grow to $110 if invested for 1 term.

Term()

This function returns the number of periods needed to reach a given future value from periodic constant payments into an interest bearing account.

Syntax

```
Term(n1, n2, n3)
```

Parameters

n1

is the payment amount made at the end of each period.

n2

is the interest rate per period.

n3

is the future value.

Returns

The number of periods or null if any of its parameters are null.

If any of n1, n2, or n3 are non-positive, the function generates an error exception.

Example 25.67 Expressions using Term()

```
Term(475, .05, 1500)
```

returns 3.00* which is the number of periods for an investment of $475, deposited at the end of each period into an account bearing 5% compound interest, to grow to $1500.00.

Logical Built-in Functions

Most of these logical functions return the boolean results true or false, represented by the numeric values of 1 and 0, respectively.

Some of the following built-in function examples make use of the identifier $ to mean a reference to the value of the object to which the form calculation is bound; this object is typically called the referencing object.

Choose()

This function selects a value from a given set of parameters.

Syntax

```
Choose(n1, s1 [, s2...])
```

Parameters

n1

is the n'th value to select from the set.

s1

is the first value of the set.

s2, ...

are optional additional value of the set.

Returns

The selected argument or null if its first parameter is null.

If n1 is less than 1 or greater than the number of arguments in the set, the function returns an empty string.

Example 25.68 Expressions using Choose()

```
Choose(3, "Accounting", "Administration", "Personnel", "Purchasing")
```

returns "Personnel".

```
Choose(Quantity, "A", "B", "C")
```

returns B if the value in Quantity is 2.

Exists()

Determines if the given parameter is an accessor to an existing object.

Syntax

```
Exists(v1)
```

Parameters

v1

is the accessor.

Returns

True (1) if the given parameter is an accessor to (a property of) an object that exists, and false (0), if it does not.

If the given parameter is not an accessor, the function returns false (0).

Example 25.69 Expressions using Exists()

```
Exists(Item)
```

returns true (1) if the object `Item` exists, false (0) otherwise.

```
Exists("hello world")
```

returns false (0) — the string is not an accessor.

```
Exists(Invoice.Border.Edge[1].Color)
```

returns true (1) if the object `Invoice` exists and has a `Border` property, which in turn, has at least one `Edge` property, which in turn, has a `Color` property. Otherwise, it returns false (0).

HasValue()

Determines if the given parameter is an accessor with a non-null, non-empty, non-blank value.

Syntax

```
HasValue(v1)
```

Parameters

v1

is the accessor.

Returns

True (1) if the given parameter is an accessor with a non-null, non-empty, non-blank value. A non-blank value will contain characters other than white spaces.

If the given parameter is not an accessor, the function returns true (1), if its a non-null, non-empty, non-blank value.

Example 25.70 Expressions using HasValue()

```
HasValue(Item)
```

returns true (1), if the object `Item` exists, and has a non-null, non-empty, non-blank value. Otherwise, it returns false (0).

```
HasValue(" ")
```

returns false (0).

```
HasValue(0)
```

returns true (1).

Oneof()

This logical function returns true if a value is in a given set.

Syntax

```
Oneof(s1, s2 [, s3...])
```

Parameters

s1

is the value to match.

s2

is the first value in the set.

s3, ...

are optional additional values in the set.

Returns

True (1) if the first parameter is in the set, false (0) if it is not in the set.

Example 25.71 Expressions using Oneof()

```
Oneof($, 4, 13, 24)
```

returns true (1) if the current object has a value of 4, 13 or 24; otherwise it returns false (0).

```
Oneof(Item, null, "A", "B", "C")
```

returns true (1) if the value in the object Item is null, "A", "B" or "C"; otherwise it returns false (0).

Within()

This logical function returns true if a value is within a given range.

Syntax

```
Within(s1, s2, s3)
```

Parameters

s1

is the value to test.

s2

is the lower bound of the range.

s3

is the upper bound of the range.

Returns

True (1) if the first parameter is within range, false (0) if it is not in range, or null if the first parameter is null.

If the first value is numeric then the ordering comparison is numeric.

If the first value is non-numeric then the ordering comparison uses the collating sequence for the current locale.

Example 25.72 Expressions using Within()

```
Within("C", "A", "D")
```

returns true (1).

```
Within(1.5, 0, 2)
```

returns true (1).

```
Within(-1, 0, 2)
```

returns false (0).

```
Within($, 1, 10)
```

returns true (1) if the value of the current object is between 1 and 10.

String Built-in Functions

FormCalc provides a large number of functions to operate on the content of strings, including the ability to:

- retrieve parts of a string
- insert parts of a string
- delete parts of a string

Many of these functions require a numeric position argument. All strings are indexed starting at character position one; i.e., character position 1 is the first character of the array. The last character position coincides with the length of the string.

Any character position less than one refers to the first character string, and any character position greater than the length of the string refers to the last character of the string.

At()

This function locates the starting character position of string s2 within string s1.

Syntax

```
At(s1, s2)
```

Parameters

s1

is the source string.

s2

is the string to search for.

Returns

The character position of the start of s2 within s1 or null if any of its parameters are null.

If string s2 is not in s1, the function returns 0.

If string s2 is empty, the function returns 1.

Example 25.73 Expressions using At()

```
At("ABC", "AB")
```

returns 1.

```
At("ABCDE", "DE")
```

returns 4.

```
At("WXYZ", "YZ")
```

returns 3.

```
At("123999456", "999")
```

returns 4.

Concat()

This function returns the string concatenation of a given set of strings.

Syntax

```
Concat(s1 [, s2...])
```

Parameters

s1

is the first string in the set.

s2,...

are additional strings to append from the set.

Returns

The concatenated string or null if all of its parameters are null.

Example 25.74 Expressions using Concat()

```
Concat("ABC", "CDE")
```

returns "ABCCDE".

```
Concat("XX", Item, "-01")
```

returns "XXABC-01" if the value of `Item` is "ABC".

Decode()

This function returns the decoded version of a given string.

Syntax

```
Decode(s1 [, s2])
```

Parameters

s1

is the string to be decoded.

s2

is a string identifying the type of decoding to perform:

- if the value is "url", the string will be URL decoded.

- if the value is "html", the string will be HTML decoded.

- if the value is "xml", the string will be XML decoded.

If s2 is omitted, the string will be URL decoded.

Returns

The decoded string.

Example 25.75 Expressions using Decode()

```
Decode("%ABhello,%20world!%BB", "url")
```

returns "«hello, world!»".

```
Decode("&AElig;&Aacute;&Acirc;&Aacute;&Acirc;", "html")
```

returns "ÆÁÂÁÂ".

```
Decode("~!@#$%^&amp;*()_+|`{&quot;}[]&lt;&gt;?,./;&apos;:", "xml")
```

returns "~!@#$%^&*()_+|`{""}[]<>?,./;':".

See Also

`Encode()`.

Encode()

This function returns the encoded version of a given string.

Syntax

```
Encode(s1 [, s2])
```

Parameters

s1

is the string to be encoded.

s2

is a string identifying the type of encoding to perform:

- if the value is "url", the string will be URL encoded.

- if the value is "html", the string will be HTML encoded.

- if the value is "xml", the string will be XML encoded.

If s2 is omitted, the string will be URL encoded.

Returns

The encoded string.

Example 25.76 Expressions using Encode()

```
Encode("""hello, world!""", "url")
```

returns "%22hello,%20world!%22".

```
Encode("ÁÂÃÄÅÆ", "html")
```

returns the HTML encoding "ÁÂÃÄÅÆ".

See Also

`Decode()`.

Format()

This function formats the given data according to the given picture clause.

Syntax

```
Format(s1, s2[, s3...])
```

Parameters

s1

is the picture clause, which may be a locale-sensitive picture clause. See "Picture Clause Specification" on page 991..

s2

is the source data being formatted.

s3, ...

is any additional source data being formatted.

For date picture clauses, the source data must be an ISO date string in one of two formats:

YYYY[MM[DD]]

YYYY[-MM[-DD]]

or, be an ISO date-time string.

For time picture clauses, the source data must be an ISO time string in one of the following formats:

```
HH[MM[SS[.FFF][z]]]
HH[MM[SS[.FFF][+HH[MM]]]]
HH[MM[SS[.FFF][-HH[MM]]]]
HH[:MM[:SS[.FFF][z]]]
HH[:MM[:SS[.FFF][-HH[:MM]]]]
HH[:MM[:SS[.FFF][+HH[:MM]]]]
```

or, be an ISO date-time string.

For date-time picture clauses, the source data must be an ISO date-time string.

For numeric picture clauses, the source data must be numeric.

For text picture clauses, the source data must be textual.

For compound picture clauses, the number of source data arguments must match the number of sub elements in the picture.

Returns

The formatted data as a string, or an empty string if unable to format the data.

Example 25.77 Expressions using Format()

```
Format("MMM D, YYYY", "20020901")
```

returns Sep 1, 2002.

```
Format("$Z,ZZZ,ZZ9.99", 1234567.89)
```

returns "$1,234,567.89" in the US and "€1 234 567,89" in France.

See Also

`IsoDate2Num()`,

`IsoTime2Num()`, and

`Parse()`.

Left()

This function extracts a number of characters from a given string, starting with the first character on the left.

Syntax

```
Left(s1, n1)
```

Parameters

s1

is the string to extract from.

n1

is the number of characters to extract.

Returns

The extracted string or null if any of its parameters are null.

If the number of characters to extract is greater than the length of the string, the function returns the whole string.

If the number of characters to extract is 0 or less, the function returns the empty string.

Example 25.78 Expressions using Left()

```
Left("ABCD", 2)
```

returns "AB".

```
Left("ABCD", 10)
```

returns "ABCD".

```
Left("XYZ-3031", 3)
```

returns "XYZ".

Len()

This function returns the number of characters in a given string.

Syntax

```
Len(s1)
```

Parameters

s1

is the string to be evaluated.

Returns

The length or null if its parameter is null.

Example 25.79 Expressions using Len()

```
Len("ABC")
```

returns 3.

```
Len("ABCDEFG")
```

returns 7.

Lower()

This function returns a string where all given uppercase characters are converted to lowercase.

Syntax

```
Lower(s1[, k1])
```

Parameters

s1

is the string to be converted.

k1

is a locale identifier string, as described in "Specifying a Locale (Locale Identifier String)" on page 919.

If k1 is omitted, the prevailing locale is used.

Returns

The lowercased string or null if any of its mandatory parameters are null.

In some locales, there are alphabetic characters that do not have an lowercase equivalent.

Bugs

The current Acrobat implementation limits the operation of this function to ASCII, Latin1, and full-width subranges of the Unicode 2.1 character set. Characters outside these subranges are never converted.

Example 25.80 Expressions using Lower()

```
Lower("Abc123X")
```

returns "abc123x".

```
Lower("ÀBÇDÉ")
```

returns "àbçdé".

Ltrim()

This function returns a string with all leading white space characters removed.

Syntax

```
Ltrim(s1)
```

Parameters

s1

is the string to be trimmed.

Returns

The trimmed string or null if its parameter is null.

White space characters includes the ASCII space, horizontal tab, line feed, vertical tab, form feed and carriage return, as well as, the Unicode space characters (Unicode category Zs).

Example 25.81 Expressions using Ltrim()

```
Ltrim(" ABC")
```

returns "ABC".

```
Ltrim(" XY ABC")
```

returns "XY ABC".

Parse()

This function parses the given data according to the given picture clause.

Syntax

```
Parse(s1, s2)
```

Parameters

s1

is a picture clause. See "Picture Clause Specification" on page 991..

s2

is the string data being parsed.

Returns

The parsed data as a string, or the empty string if unable to parse the data. The data is formatted in canonical format, as described in the chapter "Canonical Format Reference" on page 887.

A successfully parsed date is returned as an ISO date string of the form YYYY-MM-DD.

A successfully parsed time is returned as an ISO time string of the form: HH:MM:SS.

A successfully parsed date-time is returned as an ISO date-time string of the form: YYYY-MM-DDTHH:MM:SS.

A successfully parsed numeric picture clause is returned as a number.

A successfully parsed text pictures is format returned as text.

Example 25.82 Expressions using Parse()

```
Parse("MMM D, YYYY", "Sep 1, 2002")
```

returns 2002-09-01.

```
Parse("$Z,ZZZ,ZZ9.99", "$1,234,567.89")
```

returns 1234567.89 in the US.

See Also

`Format()`.

Replace()

This function replaces all occurrences of one string with another within a given string.

Syntax

```
Replace(s1, s2[, s3])
```

Parameters

s1

is the source string.

s2

is the string to be replaced.

s3

is the replacement string.

If s3 is omitted or null, the empty string is used.

Returns

The replaced string or null if any of its mandatory parameters are null.

Example 25.83 Expressions using Replace()

```
Replace("it's a dog's life", "dog", "cat")
```

returns the string "it's a cat's life".

```
Replace("it's a dog's life", "dog's ")
```

returns the string "it's a life".

Right()

This function extracts a number of characters from a given string, beginning with the last character on the right.

Syntax

```
Right(s1, n1)
```

Parameters

s1

is the string to be extract from.

n1

is the number of characters to extract.

Returns

The extracted string or null if any of its parameters are null.

If the number of characters to extract is greater than the length of the string, the function returns the whole string.

If the number of characters to extract is 0 or less, the function returns the empty string.

Example 25.84 Expressions using Right()

```
Right("ABC", 2)
```

returns "BC".

```
Right("ABC", 10)
```

returns "ABC".

```
Right("XYZ-3031", 4)
```

returns "3031".

Rtrim()

This function returns a string with all trailing white space characters removed.

Syntax

```
Rtrim(s1)
```

Parameters

s1

is the string to be trimmed.

Returns

The trimmed string or null if any of its parameters are null.

White space characters includes the ASCII space, horizontal tab, line feed, vertical tab, form feed, and carriage return, as well as, the Unicode space characters (Unicode category Zs).

Example 25.85 Expressions using Rtrim()

```
Rtrim("ABC ")
```

returns "ABC".

```
Rtrim("XYZ ABC ")
```

returns "XYZ ABC".

Space()

This function returns a string consisting of a given number of blank spaces.

Syntax

```
Space(n1)
```

Parameters

n1

is the number of spaces to generate.

Returns

The blank string or null if its parameter is null.

Example 25.86 Expressions using Concat()

```
Concat("Hello ", null, "world.")
```

returns "Hello world.".

```
Concat(FIRST, Space(1), LAST)
```

returns "Gerry Pearl" when the value of the object FIRST is "Gerry", and the value of the object LAST is "Pearl".

Str()

This function converts a number to a character string.

Syntax

```
Str(n1 [, n2 [, n3]])
```

Parameters

n1

is the number to convert.

n2

is the maximal width of the string; if omitted, a value of 10 is used as the default width.

n3

is the precision — the number of digits to appear after the decimal point; if omitted, or negative, 0 is used as the default precision.

Returns

The formatted number or null if any of its mandatory parameters are null.

The number is formatted to the specified width and rounded to the specified precision; the number may have been zero-padded on the left of the decimal to the specified precision. The decimal radix character used is the dot (.) character; it is always independent of the prevailing locale .

If the resulting string is longer than the maximal width of the string, as defined by n2, then the function returns a string of '*' (asterisk) characters of the specified width.

Example 25.87 Expressions using Str()

```
Str(2.456)
```

returns " 2".

```
Str(4.532, 6, 4)
```

returns "4.5320".

```
Str(31.2345, 4, 2)
```

returns "****".

See Also

`Format()`.

Stuff()

This function inserts a string into another string.

Syntax

```
Stuff(s1, n1, n2[, s2])
```

Parameters

s1

is the source string.

n1

is the character position in string s1 to start stuffing.

If n1 is less than one, the first character position is assumed.

If n1 is greater than then length of s1, the last character position is assumed

n2

is the number of characters to delete from string s1, starting at character position n1.

If n2 is less than or equal to 0, 0 characters are assumed.

s2

is the string to insert into s1.

If s2 is omitted or null, the empty string is used.

Returns

The stuffed string or null if any of its mandatory parameters are null.

Example 25.88 Expressions using Stuff()

```
Stuff("ABCDE", 3, 2, "XYZ")
```

returns "ABXYZE".

```
Stuff("abcde", 4, 1, "wxyz")
```

returns "abcwxyze".

```
Stuff("ABCDE", 2, 0, "XYZ")
```

returns "AXYZBCDE".

```
Stuff("ABCDE", 2, 3)
```

returns "AE".

Substr()

This function extracts a portion of a given string.

Syntax

```
Substr(s1, n1, n2)
```

Parameters

s1

is the string to be evaluated.

n1

is the character position in string s1 to start extracting.

If n1 is less than one, the first character position is assumed.

If n1 is greater than then length of s1, the last character position is assumed

n2

is the number of characters to extract.

If n2 is less than or equal to 0, 0 characters are assumed.

Returns

The sub string or null if any of its parameters are null.

If n1 + n2 is greater than the length of s1 then the function returns the sub string starting a position n1 to the end of s1 .

Example 25.89 Expressions using Substr()

```
Substr("ABCDEFG", 3, 4)
```

returns "CDEF".

```
Substr("abcdefghi", 5, 3)
```

returns "efg".

Uuid()

This function returns a Universally Unique Identifier (UUID) string which is guaranteed (or at least extremely likely) to be different from all other UUIDs generated until the year 3400 A.D.

Syntax

```
Uuid([n1])
```

Parameters

n1

identifies the format of UUID string requested:

- if the value is 0, the returned UUID string will only contain hex octets.

- if the value is 1, the returned UUID string will contain dash characters separating the sequences of hex octets, at fixed positions.

If n1 is omitted, the default value of 0 will be used.

Returns

The string representation of a UUID, which is an optionally dash-separated sequence of 16 hex octets.

Bugs

When used in in the XML Forms Plugin environment, the current implementation of this function does not return anything useful.

Example 25.90 Expressions using Uuid()

```
Uuid()
```

returns "3c3400001037be8996c400a0c9c86dd5" on some system at some point in time.

```
Uuid(1)
```

returns "1a3ac000-3dde-f352-96c4-00a0c9c86dd5" on that same system at some other point in time.

Upper()

This function returns a string with all given lowercase characters converted to uppercase.

Syntax

```
Upper(s1[, k1])
```

Parameters

s1

is the string to convert.

k1

is a locale identifier string, as described in "Specifying a Locale (Locale Identifier String)" on page 919.

If k1 is omitted, the prevailing locale is used.

Returns

The uppercased string or null if any of its mandatory parameters are null.

In some locales, there are alphabetic characters that do not have a lowercase equivalent.

Bugs

The current Acrobat implementation limits the operation of this function to the ASCII, Latin1, and full-width subranges of the Unicode 2.1 character set. Characters outside these subranges are never converted.

Example 25.91 Expressions using Upper()

```
Upper("abc")
```

returns "ABC".

```
Upper("àbCdé")
```

returns "ÀBCDÉ".

WordNum()

This function returns the English text equivalent of a given number.

Syntax

```
WordNum(n1 [, n2 [, k1]])
```

Parameters

n1

is the number to be converted.

n2

identifies the format option as one of the following:

- if the value is 0, the number is converted into text representing the simple number.

- if the value is 1, the number is converted into text representing the monetary value with no fractional digits.

- if the value is 2, the number is converted into text representing the monetary value with fractional digits.

If n2 is omitted, the default value of 0 will be used.

k1

is a locale identifier string, as described in "Specifying a Locale (Locale Identifier String)" on page 919.

If k1 is omitted, the default en_US locale is used.

Note: This argument is currently ignored in Acrobat. `WordNum()` can be used only in English-speaking locales.

Returns

The English text, or null if any of its parameters are null.

If n1 is not numeric or the integral value of n1 is negative or greater than 922,337,203,685,477,550 the function returns "*" (asterisk) characters to indicate an error condition.

Bugs

By specifying a locale identifier other than the default, it should be possible to have this function return something other than English text. However the language rules used to implement this function are inherently English. Thus, for now, the locale identifier is ignored.

Example 25.92 Expressions using WordNum()

```
WordNum(123.54)
```

returns "One Hundred Twenty-three".

```
WordNum(1011.54, 1)
```

returns "One Thousand Eleven Dollars".

```
WordNum(73.54, 2)
```

returns "Seventy-three Dollars And Fifty-four Cents".

URL Built-in Functions

FormCalc provides a number of functions to manipulate the content of URLs, including the ability to:

- download data from a URL,
- upload data to a URL, and
- post data to a URL

These functions are only operational when a protocol host has been provided to the **FormCalc** engine. The list of supported URL protocols (http, https, ftp, file) may thus vary with each protocol hosting environment.

Get()

This function downloads the contents of the given URL.

Syntax

```
Get(s1)
```

Parameters

s1

is the URL being downloaded.

Returns

The downloaded data as a string, or an error exception if unable to download the URL's contents.

Example 25.93 Expressions using Get()

```
Get("http://www.w3.org/TR/REC-xml-names/")
```

returns the Namespaces in XML standard from the World Wide Web Consortium.

```
Get("ftp://ftp.gnu.org/gnu/GPL")
```

returns a document our Legal Department studies carefully.

```
Get("http://example.com?sql=SELECT+*+FROM+projects+FOR+XML+AUTO,+ELEMENTS")
```

returns the result of an SQL query as an XML document.

See Also

Post() and Put()

Post()

This function posts the given data to the given URL.

Syntax

```
Post(s1, s2[, s3[, s4[, s5]]])
```

Parameters

s1

is the URL being posted.

s2

is the data being posted.

s3

is an optional string containing the name of the content type of the data being posted. Valid content types include:

text/html
text/xml
text/plain
multipart/form-data
application/x-www-form-urlencoded
application/octet-stream
any valid MIME type

If s3 is omitted, the content type defaults to "application/octet-stream". Note that the application is responsible for ensuring that the posted data is formatted according to the given content type.

s4

is an optional string containing the name of the code page that was used to encode the data being posted. Valid code page names include:

UTF-8
UTF-16
ISO8859-1
Any recognized [IANA] character encoding

If s4 is omitted, the code page defaults to "UTF-8". Note that the application is responsible for ensuring that the posted data is encoded according to the given code page.

s5

is an optional string containing any additional HTTP headers to be included in the post. If s5 is omitted, no additional HTTP header is included in the post. Note that when posting to SOAP servers, a "SOAPAction" header is usually required.

Returns

The post response as a string, or an error exception if unable to post the data. The response string will be decoded according to the response's content type. For example, if the server indicates the response is UTF-8 encoded, then this function will UTF-8 decode the response data before returning to the application.

Example 25.94 Expressions using Post()

```
Post("http://tools_build/scripts/jfecho.cgi",
"user=joe&passwd=xxxxx&date=27/08/2002",
"application/x-www-form-urlencoded")
```

posts some urlencoded login data to a server and returns that server's acknowledgement page.

```
Req = "<?xml version='1.0' encoding='UTF-8'?>"
Req = concat(Req, "<soap:Envelope>")
Req = concat(Req, " <soap:Body>")
Req = concat(Req, "  <getLocalTime/>")
Req = concat(Req, " </soap:Body>")
Req = concat(Req, "</soap:Envelope>")
Head = "SOAPAction: ""http://www.Nanonull.com/TimeService/getLocalTime"""
Url = "http://www.nanonull.com/TimeService/TimeService.asmx/getLocalTime"
Resp = post(Url, Req, "text/xml", "utf-8", Head)
```

posts a SOAP request for the local time to some server, expecting an XML response back.

See Also

Get() and Put()

Put()

This function uploads the given data into the given URL.

Syntax

```
Put(s1, s2[, s3])
```

Parameters

s1

is the URL being uploaded.

s2

is the data being uploaded.

s3

is an optional string containing the name of the code page that is to be used to encode the data before uploading it. Valid code page names include:

- UTF-8,

- UTF-16,

- ISO8859-1, or

- any recognized [IANA] character encoding.

If s3 is omitted, the code page defaults to "UTF-8".

Returns

The empty string, or an error exception if unable to upload the data.

Example 25.95 Expressions using Put()

```
Put("ftp://www.example.com/pub/fubu.xml",
   "<?xml version='1.0' encoding='UTF-8'?><msg>hello world!</msg>")
```

returns nothing if the ftp server permits the user to upload some xml data to the file `pub/fubu.xml`.

`Get()` and `Post()`

Miscellaneous Built-in Functions

Ref()

Returns a reference to an existing object.

Syntax

```
Ref(v1)
```

Parameters

v1

is an accessor, reference, method, function or value.

Returns

A reference (or "handle") to an existing object if the given parameter is an accessor referring to an existing object, or an existing reference, or a method that returns an object, or, a function that evaluates to an object.

If the given parameter is null, the function returns the null reference. For all other given parameters, the function returns the value given.

Example 25.96 Expressions using Ref()

```
Ref(Invoice.Border.Edge[4].Color)
```

might return a handle to a color object.

```
Ref("hello")
```

returns "hello".

See Also

Exists()

UnitValue()

Returns the value of a unitspan after an optional unit conversion. A unitspan string consist of a number immediately followed by a unit name. Recognized unit names include:

Unit Name	Meaning
`in` `inches`	inches (2.54 cm)
`mm` `millimeters`	millimeters
`cm` `centimeters`	centimeters
`pt` `points`	points (1/72 inch)
`mp` `millipoints`	millipoints (1/72000 inch)[a]

a.Millipoints are retained for backwards compatability with very early versions of XFA, however the use of millipoints is deprecated.

Syntax

```
UnitValue(s1 [, s2])
```

Parameters

`s1`

is a unitspan string.

`s2`

is an optional string containing a unit name. The unitspan's value will be converted to the given units. If s2 is omitted, the unitspan's units are used.

Returns

The unitspan's value.

Example 25.97 Expressions using UnitValue()

```
UnitValue("1in", "cm")
```

returns 2.54.

```
UnitValue("72pt", "in")
```

returns 1.

UnitType()

Returns the units of a unitspan.

Syntax

```
UnitType(s1)
```

Parameters

s1

is a unitspan string.

Returns

The unitspan's units. Unit names are canonized as follows.

Input unit name	Canonical unit name
in inches	in
mm millimeters	mm
cm centimeters	cm
pt points	pt
mp millipoints	mp

Note: Millipoints are retained for backwards compatability with very early versions of XFA. However the use of millipoints is deprecated.

Example 25.98 Expressions using UnitType()

```
UnitType("36in")
```

returns "in".

```
UnitValue("2.54centimeters")
```

returns "cm".

26 Picture Clause Specification

This reference describes picture clauses and the syntax used to express them. It provides guidance to template designers who wish to create picture clauses that specify the format of dates, times, numbers and text. Such picture clauses are used in the following contexts:

- Data output formatting and input parsing, as described in "Localization and Canonicalization" on page 138
- Data conversion performed by some FormCalc functions, as described in "FormCalc Specification" on page 891

About

Picture clauses are a sequence of symbols (characters) that specify the rules for formatting and parsing textual data, such as dates, times, numbers and text. Each *symbol* is a place-holder that typically represents one or more characters that occur in the data. FormCalc supports the BMP characters described in [Unicode-2.1].Often the terms *pattern* and *picture format* are used as synonyms for the term *picture clause*.

A picture clause can be used equally for output formatting of data and input parsing of data. *Output formatting* is the process of transforming a raw value into a formatted value, under the direction of a picture clause. *Input parsing* is the process of transforming a formatted input value into a raw elemental value, again under the direction of a picture clause. The raw elemental values that result from input parsing are represented in the canonical formats described in "Canonical Format Reference" on page 887.

Picture clauses can passively use the template-declared locale or the ambient locale, or they can specify a particular locale (language alone or a country and language). Additionally, date and time picture clauses can specify certain characteristics used in East Asian locales, including ideographs and full-width characters, Asian numeric system, era years, and era styles. ("Asian Date, Time and Number Considerations" on page 1001)

How Picture Clauses Are Used

Picture clauses are used as properties in XFA templates and as arguments in FormCalc, as described in the "FormCalc Specification" on page 891. An XFA processing application uses the picture clause properties to determine how to perform localized output formatting and input parsing, as described in "Localization and Canonicalization" on page 138.

The following table summarizes the role of picture clauses in FormCalc functions.

Picture clause parent element (Alternate name)	**Output formatting**	**Input parsing**	**Role of picture clause**
`format()`	✔		Specifies the formatting to be applied to a date, which is supplied in canonical format
`parse()`		✔	Specifies the formatting expected in a date

How Picture Clauses Evolved

Java's text API includes a `DateFormat` class which has evolved along similar principles; its meta-symbol specifications is more intuitive than its Unix predecessors. Java's designers extended these concepts to format and parse numeric data, again, in a locale-sensitive matter.

Java has substantially influenced picture clauses. There's almost a one-to-one correspondence between our date and time picture clauses and the Java `SimpleDateFormat` class pattern strings. The need to better accommodate legacy applications accounts for some of the differences — picture clauses were initially described as part of FormCalc's date and time functions — functions that were modeled from another legacy script language.

Picture-Clause Building Blocks

Picture clauses are expressed using a combination of symbols and literals. The symbols are either context-specific (apply only in certain contexts, such as dates or times) or global.

Context-Specific Picture-Clause Symbols

The following example illustrates the result of applying a picture clause to two sample data values; the picture clause is designed to format the numeric data values into a result with two fractional digits, suppressing leading zeros, and adding a grouping separator.

Example 26.1 Numeric output formatted using a picture clause

Picture clause	Input value	Formatted result
`zz,zz9.99`	`2157.5`	`2,157.5`
	`50.6`	`50.60`

The next example illustrates the result of applying the same picture clause to two input data values. The picture clause is designed to parse numeric values having up to seven significant digits, the three least significant digits being mandatory, with two of them being fractional digits; six and seven digit numbers must include a grouping separator.

Example 26.2 Numeric input parsed using the same picture clause

Picture clause	Input value	Parsed result
`zz,zz9.99`	`2,157.50`	`2157.5`
	`50.60`	`50.60`

Due to the varying types of data that can benefit from the application of picture clauses, it is useful to divide the picture clause symbols into categories that correspond to a type of data. This permits us to reuse individual picture clause symbols across categories. For instance, the date picture clause `D/M/YYYY` makes use of the symbol **`M`** to represent the month portion of a date; however, it is equally useful to permit the symbol **`M`** to represent the minute portion of a time in the picture clause `H:M:S`.

Note: Picture clause symbols are case-sensitive and must correspond exactly to this specification.

Global Picture-Clause Symbols

There are some symbols that are used in all categories of picture clauses. These global picture cause symbols are described in the following table:

Symbol	Affect on output formatting and input parsing
?	• Input parsing: Match any one character, as in a wild-card • Output formatting: Format as a space
*	• Input parsing: Match *zero* or more whitespace characters[a] • Output formatting: Format as a space
+	• Input parsing. Match *one* or more whitespace characters[a] • Output formatting: Format as a space

a.The term whitespace characters means any [UNICODE] character classified as a break space. When input parsing, any whitespace character is accepted, and when output formatting, a single space character is emitted.

Picture Clause Literals

A picture clause may contain any combination of picture symbols and literal text, as illustrated in the following examples. The outputs have been quoted, so that one can see the significant spaces — the quotes are not actually part of the output.

Picture clause literals may be standard separators or text enclosed in matched single quotes. The separators include the standard and full-width comma (`,`), dash (`-`), colon (`:`), slash (`/`), period (`.`) and space ( ).

To embed a quote within a literal, specify two quote characters.

Output Formatting

When output formatting, literals are formatted verbatim into the output text. the following numeric picture clause examples include several literals (single-quoted and default). The following examples use double-underline to identify literals. The double underlines are not part of the picture clause expression.

Example 26.3 Output formatted using literals

Picture clause	Input value	Formatted result
'You owe' zz,zz9.99'!'	2157.5	"You owe 2,157.50!"
	50.6	"You owe 50.60!"

Input Parsing

When input parsing, literals must match the input data verbatim, but never contribute to the resulting text, as illustrated in the following examples.

Example 26.4 Input parsed using literals

Picture clause	Input value	Formatted result
'You owe' zz,zz9.99'!'	You owe 2,157.50!	2157.50
	You owe 50.60!	50.60

Note: Any alphabetic or punctuation character appearing within a picture clause that is not specified by this document as a valid picture symbol and is not enclosed within quotes as a literal, is reserved for future use as a potential picture symbol and should be considered a user error irrespective of the current implementation behavior.

Locale Identifier Strings

Locale influences date, time, and number picture clause processing. For example, the format used for full date presentations differs between English-language (October 25, 2002) and French-language locales (25 octobre, 2002). This section describes what a locale is, how the locale is determined and how locales are identified in picture clauses.

What a Locale Is

When developing internationalized applications, a locale is the standard term used to identify a particular cultural context (language and/or country). A locale defines (but is not limited to) the format of dates, times, numeric and currency punctuation that are culturally relevant to a specific cultural context. A properly internationalized application will always rely on the locale to supply it with the format of dates and times. This way, users operating in their locale will always be presented with the date and time formats they are accustomed to.

A locale is identified by a language code and/or a country code. Usually, both elements of a locale are important. For example, the names of weekdays and months in English Canada and in the United Kingdom are formatted identically, but dates are formatted differently. So, specifying an English language locale would not suffice. Conversely, specifying only a country as the locale may not suffice either — for example, Canada, has different date formats for English and French.

Determining the Prevailing Locale

There are several sources for locale information. For example, the hosting operating system may provide an XFA processing application with a locale to use and the picture clause may provide a locale. The *prevailing locale* is the locale that should be used for input parsing or output formatting.

An XFA processing application determines prevailing locale by examining the following, in order:

1. Explicit declaration in the picture clause ("Convention for Explicitly Naming Locale").
2. Template field or subform declarations, using the `locale` property.
3. Ambient locale. *Ambient locale* is the system locale declared by the application or in effect at the time the XFA processing application is started. In the event the application is operating on a system or within an environment where a locale is not present, the ambient locale defaults to English United States (`en_US`).

"Localization and Canonicalization" on page 138 in the chapter "Exchanging Data Between an External Application and a Basic XFA Form" provides additional information about localization.

Convention for Explicitly Naming Locale

Picture clauses may include locale identifier strings. Such designators conform to the following syntax, where the square brackets show optional parts:

```
language[_country[_modifier]]
```

The syntax includes codes for the following:

- `language`. Language codes use the 2-character representation for languages specified in [ISO 639-1].
- `country`. Country codes use the 2-character country representations specified in [ISO-3166-1].
- `modifier`. Modifier codes can be used to mean a variety of things. For example, en_GB_EURO uses the modifer to select an alternate default currency. By contrast th_TH_TH and ko_KR_Hani use the modifier to select an alternate default script.

"Locale-Specific Picture Clauses" on page 998 explains how to explicitly declare locale in a picture clause.

The following table presents examples of locale identifier strings. Such designators can change, reflecting the dynamic geopolitical world.

Supported locale identifiers

locale identifier string	Description
ar_SA	Arabic specific for Saudi Arabia
en	English
en_CA	English specific for Canada
en_GB	English specific for the United Kingdom.
en_GB_EURO	English specific for the United Kingdom, using the Euro as the default currency.
fr	French
fr_CA	French specific for Canada
ko_KR	Korean specific for the Republic of Korea. The default ideograph script for this designator is Hangul.
ko_KR_Hani	Korean specific for the Republic of Korea. The ideograph script for this designator is Hanja.
th_TH_TH	Traditional Thai with Thai digits
zh_CN	Chinese specific for China
zh_HK	Chinese specific for Hong Kong

Note: The full locale identifier strings (language_country) should be used for currency numeric values because currencies differ from country-to-country. For example, the currency representations for the en_GB (English for the United Kingdom) and en_CA (English for Canada) locales are quite different, even though some of their date representations are identical.

Complex Picture-Clause Expressions

Picture clauses are structured to support increasing complex string sequences. A simple picture clause can format a date, time, number, or text. These simple picture clauses may be assembled into more complex structures that reflect locale and that provide picture clause choices. The following table summarizes the levels of structure that can be applied to picture clauses. The square brackets enclose optional, repeating items.

Structure	Expression
Simple picture clauses (described earlier in this chapter)	`picture-clause`
"Predefined Picture Clauses"	`category.subcategory{}`
"Compound Picture Clauses"	`category{picture-clause}category{picture-clause} [category{picture-clause}]`[a]
"Locale-Specific Picture Clauses"	`(locale){picture-clause}`
"Locale-Specific, Compound Picture Clauses"	`category(locale){picture-clause}category(locale) {picture-clause}`
"Alternate Picture Clauses"	`picture-clause|picture-clause[|picture-clause...]`[a]
"Alternate Locale-Specific Picture Clauses"	`category(locale){picture-clause}| category(locale){picture-clause} [|category(locale){picture-clause}]`[a]

a.Square brackets represent optional, repeating parts of a picture clause expression. They are not part of the picture clause.

Predefined Picture Clauses

There is a set of predefined picture clauses that can be used in picture processing. The advantage of using one of these predefined picture clauses is that it is already defined across all locales. The template can invoke a predefined picture clause and automatically work across locales.

Within a template a predefined picture clause is invoked with the following syntax:

```
category-name.subcategory-name{}
```

where category-name is one of the keywords `date`, `time`, `num`, or `text`, and subcategory-name depends on category-name as shown in the following table.

Category name	Subcategory name	*Example 26.5 Example data in `en_us` locale*	*Example 26.6 Same data in `fr_fr` locale*
`date`	`short`	`7/1/06`	`01/07/06`
	`medium`	`Jul 1, 2006`	`1 juil. 2006`
	`long`	`July 1, 2006`	`1 juillet 2006`
	`full`	`Saturday, July 1, 2006`	`samedi 1 juillet 2006`
	`default`	`Jul 1, 2006`	`1 juil. 2006`

Category name	Subcategory name	*Example 26.5 Example data in* `en_us` *locale*	*Example 26.6 Same data in* `fr_fr` *locale*
time	short	5:23 PM	17:23
	medium	5:23:52 PM	17:23:52
	long	5:23:52 PM EDT	17:23:52 CEST
	full	5:23:52 PM EDT	17 h 23 CEST
	default	5:23:52 PM	17:23:52
datetime	short	7/1/06 5:23 PM	01/07/06 17:23
	medium	Jul 1, 2006 5:23:52 PM	1 juil. 2006 17:23:52
	long	July 1, 2006 5:23:52 PM EDT	1 juillet 2006 17:23:52 CEST
	full	Saturday, July 1, 2006 5:23:52 PM EDT	samedi 1 juillet 2006 17 h 23 CEST
	default	Jul 1, 2006 5:23:52 PM	1 juil. 2006 17:23:52
num	integer	1,234	1.235
	decimal	1,234.56	1.234,56
	currency	$1,234.56	€1.234,56
	percent	1,234%	1.234%

The picture name `default` delegates to the XFA processor the selection of a predefined picture clause. Which one it picks is implementation defined. For example, Acrobat selects `medium` when `default` is specified. There is no `default` option for the `num` category.

The predefined picture clauses for `date`, `time` and `num` are carried in the `localeSet` packet of the XDP and can be redefined by the form creator. In contrast the `datetime` predefined picture is not exposed in the `localeSet` packet and cannot be redefined. For most locales it is the corresponding `date` picture, followed by a space (U+0020) character, followed by the corresponding `time` picture, however for some locales the order of date and time is reversed and for some locales the separator is not a space or there is no separator.

Within the `localeSet` packet there are only three numeric picture clauses. The picture clause named `numeric` does double-duty. The integer part of it is used for `num.integer` formatting and the whole picture clause is used for `num.decimal` formatting.

Compound Picture Clauses

In some circumstances it is necessary to construct a picture clause which comprises other picture clauses from more than one category. Consider the following example of two data items formatted with a compound picture clause. Results have been quoted, so that one can see where spaces would appear in the formatted value. The quotes are not actually part of the result.

Example 26.7 Output formatted using compound picture clauses

Picture clause	Input value	Formatted result
`'Balance on' date{MMM YY}: num{z,zz9.99}`	`1999-12 2157.50`	`"Balance on Dec 99: 2,157.50"`
	`2003-12 50.60`	`"Balance on Dec 03: 50.60"`

The first data item is a value representing the date Dec 1999, and the second data item is a numeric value. The picture clause uses a bracketing syntax to partition the date picture sub-clause from the numeric picture sub-clause. The left and right brace characters are reserved for this purpose, and therefore must always be quoted within a picture clause to obtain a brace literal character.

The syntax for compound picture clauses is:

```
category-name{picture-symbols}
```

where category-name is one of the keywords `date`, `time`, `datetime`, `num`, `text`, `zero`, or `null`, and picture-symbols corresponds to one or more picture symbols from a particular picture category. The characters enclosed within the curly braces are interpreted as part of the picture clause.

Note that the quoted literals in the previous example could have appeared inside or outside of the braces with equal results. Therefore the following compound picture clauses are all equivalent.

Example 26.8 Equivalent compound picture clauses

```
date{'Balance as of' MMM, YY}: num{z,zz9.99}
date{'Balance as of' MMM, YY: }num{z,zz9.99}
date{'Balance as of' MMM, YY}num{': 'z,zz9.99}
```

Explicitly stating the category-name is not required for picture clauses that contain picture symbols from only one category; the processing application must attempt to infer the category based upon the symbols found in the picture clause. If the symbols are ambiguous or too complex to automatically identify to a category, then it is an error. Explicitly stating the category within the compound picture clause shall always take precedence over any other interpretation of the picture clause by the processing application.

Locale-Specific Picture Clauses

Picture clauses can specify the prevailing locale to use for picture processing. This ability is useful when formatting or parsing locale-specific data (such as dates, times or currencies) for a locale that differs from any template-declared locales or from the ambient locale. (See also "Determining the Prevailing Locale" on page 994)

Note: Starting with XFA 2.4 this standard fully supports locales where the ordinary flow of text is right to left, as well as the left to right locales previously supported. However, regardless of the locale, picture clauses are always processed from left to right (i.e. document order). In circumstances where the flow of text is also left to right this means the picture clause more or less resembles the data to be put out or taken in. However when the flow of text is right to left the picture clause resembles a mirror image of the data. This is not an error. The picture clause itself **must** be parsed in a locale-independent way.

The syntax for specifying a locale-specific picture clause is:

```
category-name(locale-name){picture-symbols}
```

or

```
category-name.subcategory-name(locale-name){}
```

where category-name and picture-symbols are as before, and locale-name is the name of a locale conformant to the locale naming standards defined above.

Example 26.9 Output formatted using locale-specific picture clauses

Picture clause	Input value	Formatted result
`date(fr){DD MMMM, YYYY}`	`2002-10-25`	`25 octobre, 2002`
`date(es){EEEE, D 'de' MMMM 'de' YYYY}`	`2002-10-25`	`viernes, 25 de octubre de 2002`
`date.long(fr)()`	`2002-20-25`	`25 octobre, 2002`

Locale-Specific, Compound Picture Clauses

Picture clauses may be assembled into a series of locale-specific picture clauses, using the following syntax:

```
category(locale){picture-clause} category(locale){picture-clause}
```

Locale-specific compound picture clauses are supported; however, their usefulness is limited since most individual data items pertain to a single locale.

Alternate Picture Clauses

There are circumstances when raw data may be in one of many formats, yet the formatting capabilities of picture clauses are still desired. For example, if inputting phone numbers, the user might prefer omitting the area code of local numbers, so the processing application might be required to parse ten digit numbers and seven digit numbers, both being equally valid. To that end, we provide alternative picture clauses. These are simply a series of picture clauses separated by a vertical bar (`|`) character. Everything to the right of the vertical bar character up to another vertical bar or the end of the picture clause is one picture clause alternative.

The syntax for alternate picture clauses is:

```
picture-clause|picture-clause[|picture-clause...]
```

where picture-clause is as defined above, and the square brackets and ellipses denote optional, repeated, alternate picture clauses. Thus, the vertical bar character is reserved for delimiting alternate picture clauses, and therefore, must always be quoted within a picture clause to obtain the vertical bar literal character.

During input parsing and output formatting against a set of alternate picture clauses, the XFA processing application chooses the picture clause to use, by sequentially matching the data against each picture clause in the expression, stopping when a match is found. The picture clauses are examined in order from left to right.

The following table presents examples of input parsing with alternate picture clauses. The parsed result is the canonical format of the data. Picture clauses used for input parsing are relevant only when the picture clause appears in a `ui`, `bind`, or `connect` element.

Example 26.10 Input parsed using alternate picture clauses

Picture clause	Input value	Parsed result

`null{'No data'}`
`  | null{}`
`  | text{999*9999}`
`  | text{999*999*9999}`
	`"555 1212"`	`5551212`

	`"613 555 1212"`	`6135551212`
	`""` (Zero Length string)	`Null`
	`"No data"`	`Null`
	`"Hello"`	`Hello`[a]

a.The input value does not match any of the picture clauses, so is left as-is. That is, additional processing or validation checks may be required before the data can be assumed to be in canonical format.

The following table presents examples of output formatting with alternate picture clauses.

Example 26.11 Output formatted using alternate picture clauses

Picture clause	Input data	Formatted result

`null{'No data'}`
`  | null{}`
`  | text{999*9999}`
`  | text{999*999*9999}`
	`5551212`	`"555 1212"`

	`6135551212`	`"613 555 1212"`
	`Null`	`"No data"`
	`Hello`	`"Hello"`[a]

a.The data does not match any of the picture clauses, so it is passed through unchanged.

Alternate Locale-Specific Picture Clauses

Picture clauses may be constructed as a set of alternate picture clauses, with each part containing a locale identifier string. As with "Alternate Picture Clauses", the alternate picture clauses are used only for input parsing.

```
category(locale){picture-clause}|category(locale){picture-clause}
  [|category(locale){picture-clause}]
```

The following table presents examples of input parsing with alternate picture clauses.

Example 26.12 Input parsed using alternate picture clauses

Picture clause	Input value	Parsed result
`num(ar_SA){$z,zz9.zzs}\|num(en_GB){$z,zz9.99}`	€100.00	100
	١٠٠٫٠٠ ر.س.	100
`text(th_TH){999*9999}|text(th_TH_TH){999*9999}`	555-1212	5551212
	๕๕๕๑๒๑๒	5551212

Calendars and Locale

The standard Western calendar is the Gregorian calendar. Although the months and days of the week have different names in different locales, all users of the Gregorian calendar agree upon the numbering of the day, of the month, and of the year.

The architecture and grammar of XFA support the entry and display of dates in other calendars. However so far the only non-Gregorian calendars that have been specified are calendars that use Gregorian months and days but calculate the year differently. For example, the Gregorian year 2007 is the Korean year 4340. For more information about the Korean calendar see "Korean Date Time Rules" on page 1004.

In addition calendars that calculate the year differently split up historic time into different eras. The Gregorian calendar splits up time into the BC and AD eras. By contrast Chinese calendars split up historical time into eras corresponding to ruling dynasties. For more information about Chinese calendars see "Chinese (Taiwan) Date Time Rules" on page 1004 and "Chinese (China) Date Time Rules" on page 1005.

When the particular locale in effect has an calendar which uses Gregorian months and days but an alternate year and era calculation, the XFA processor uses the alternate year and era calculation by default. Otherwise it defaults to the Gregorian year and era.

A future version of this specification will deal with additional calendars that do not use Gregorian months and days, such as lunar calendars.

Asian Date, Time and Number Considerations

This section describes special considerations for describing East Asian eras in date and time picture clauses.

Note: Henceforth, this section uses the term *Asian* to denote East Asian locales.

Asian date representations may differ from Western ones in several respects:

- *Characters/ideographs*. Asian dates may use full-width characters or ideographs rather than Latin numbers. ("Using Full-Width Characters and Ideographs in Date and Time Data" on page 1001)
- *Numeric systems*. Asian dates may use either the standard Arabic numeric system or another system, described in this document as the tens rule. ("Tens Rule Numeric System" on page 1002)
- *Eras*. Date picture clauses allow years to be represented in terms of the Gregorian calendar or in terms of imperial eras. ("Imperial (Alternate) Eras and Alternate Era Styles" on page 1003)
- *Era name symbol styles*. Some Asian locales use multiple character and ideographic styles for an era name. ("Imperial (Alternate) Eras and Alternate Era Styles" on page 1003)

Using Full-Width Characters and Ideographs in Date and Time Data

In Asian prevailing locales, date and time picture clauses can specify the appearance of the data as any one of the following:

- ASCII digits 0-9 (U+30 to U+39)
- Unicode full-width digits 0-9 (U+FF10 - U+FF19)
- Ideographic numbers specific for the locale.

The following table illustrates such ideographic numbers.

Universal		Examples of Asian Ideographic Digits		
Latin digits	**Full-width digits**	**Kanji**	**Hangul**	**Hanja**
0	０ (U+FF10)	〇 (U+3007)	영 (U+C601)	零 (U+96F6)
1	１ (U+FF11)	一 (U+4E00)	일 (U+C77C)	一 (U+4E00)
2	２ (U+FF12)	二 (U+4E8C)	이 (U+C774)	二 (U+4E8C)
3	３ (U+FF13)	三 (U+4E09)	삼 (U+C0BC)	三 (U+4E09)
4	４ (U+FF14)	四 (U+56DB)	사 (U+C0AC)	四 (U+56DB)
5	５ (U+FF15)	五 (U+4E94)	오 (U+C624)	五 (U+4E94)
6	６ (U+FF16)	六 (U+516D)	육 (U+C721)	六 (U+516D)
7	７ (U+FF17)	七 (U+4E03)	칠 (U+CE60)	七 (U+4E03)
8	８ (U+FF18)	八 (U+516B)	팔 (U+D314)	八 (U+516B)
9	９ (U+FF19)	九 (U+4E5D)	구 (U+AD6C)	九 (U+4E5D)
10	１０	十 (U+5341)	십 (U+C2ED)	十 (U+5341)
100	１００	百 (U+767E)	백 (U+BC31)	百 (U+767E)
1000	１０００	千 (U+5343)	천 (U+CC9C)	千 (U+5343)

Using Full-Width Characters in Number Data

In Asian prevailing locales, number picture clauses can specify the appearance of data as standard ASCII characters or full-width characters.

Tens Rule Numeric System

Asian numbers may be assembled using either Arabic number format or Tens Rule.

- *Arabic numeral system*. In this system, numbers are, representing increasing orders of magnitude for every digit to the left of the (imaginary) decimal point. In this convention, the ideographic characters

are simply concatenated together. For example, using Kanji digits, 10 (一〇) is character 1 (一) and 0 (〇) together, 11 (一一) is two occurrences of the character 1 (一), while 32 (三二) is character 3 (三) and 2 (二) together.

- *Tens rule*. When using the tens rule to symbolically display a numeric value, the number of tens's (十) and singletons are combined together. Thus, again using Kanji digits, 20 (二十) is 2 (二) tens (十), and 32 (三十二) is 3 tens (三十) plus 2 (二). As with all rules, there's an exception: 10 is represented using one ideograph (十) and not (一十).

The tens rule naturally extends to values in the hundreds (百) and in the thousands (千). Only Korean years have values in the thousands. Future Taiwanese eras may have year values in the hundreds. Most other CKJ numeric values are in the tens.

Imperial (Alternate) Eras and Alternate Era Styles

This section describes the representation of imperial eras in Asian locales.

Asian locales identify the start and end of a year according to the Gregorian calendar; however, such locales may use multiple eras, where an *era* is a convention for assigning an origin to the number of years:

- Gregorian calendar era. The origin of a year is relative to the birth and death of Christ. This convention uses the era names BC and AD. In date picture clauses, the Gregorian calendar era is the *primary or default era*.
- Imperial era. The origin of a year is the beginning of an emperor's reign. In this convention, an era identifier precedes the year. The era identifier does not necessarily include the emperor's name. In date picture clauses, the imperial era is called the *alternate era*.

  An Asian locale may use multiple *era styles* for imperial era identifiers, with each style using a different method for representing an imperial era. The following section describe era styles for the supported Asian locales.

Japanese Date Time Rules

The following rules apply exclusively to the locales `ja` (Japanese) and `ja_JP` (Japanese - Japan).

The last century spanned the reign of four Japanese emperors:

Imperial era	Dates relative to the Gregorian calendar
Meiji	1868/09/08 to 1912/07/29
Taisho	1912/07/30 to 1926/12/24
Showa	1926/12/25 to 1989/01/07
Heisei	1989/01/08 to present

An alternate era may be represented in several styles. For example, the following table shows the different era styles for the Heisei era.

Style number	Date picture symbol	*Example 26.13 Representation of the Heisei era*	
		Character/Ideograph	Unicode
1	g	H	U+48
2	gg	平	U+5E73
3	ggg	平成	U+5E73 U+6210
4	g[a]	Ｈ	U+FF28
5	gg[a]	㍻	U+337B

a.This picture symbol is expressed as full-width characters.

Korean Date Time Rules

The following rules apply exclusively to the locales `ko` (Korean language) and `ko_KR` (Korean language for the Republic of Korea).

The Tangun era began 2333 BC. To convert the current year (2004) into its Tangun era counterpart, the value 2333 must be added.

Korean date and times values use a single alternate era style; however, Korea uses two different sets of ideographs, depending on the script of the prevailing locale ("Determining the Prevailing Locale" on page 994). The following table shows the Tangun era ideographs represented using the supported scripts.

Korean Script	Ideograph	Unicode
Hangul	단기	U+B2E8 U+AE30
Hanja	檀紀	U+6A80 U+7D00

Note: For the Tangun era, the value 2333 must be added to the current year. For example, the year 2004 is represented as 4337.

Note: Korean numbers always use tens rule.

Chinese (Taiwan) Date Time Rules

These rules apply exclusively to the locale `zh_TW` (Chinese - Taiwan).

The last century spanned three eras, as described in the following table.

Imperial era	Dates relative to the Gregorian calendar
GuangXu	1875/01/01 to 1908/12/31
XuanTong	1909/01/01 to 1911/12/31
MinGuo	912/01/01

Chinese (Taiwan) dates and times always use the tens rule, with the following exception. When represented *without* the Chinese era, the year is represented using Arabic number format. For example, 2004 is represented as 二〇〇四.

Chinese (China) Date Time Rules

These rules apply equally to the locales `zh_CN` (Chinese - China), `zh_HK` (Chinese - Hong Kong), and `zh_MO` (Chinese - Macau).

The last century spanned 4 eras:

Imperial era	Dates relative to the Gregorian calendar
GuangXu	1875/01/01 to 1908/12/31
XuanTong	1909/01/01 to 1911/12/31
MinGuo	1912/01/01 to 1949/09/30
unnamed	1949/10/01 to present

For dates from October 1, 1949, to present, there is no symbol for the Chinese imperial era.

Numeric date and time values are always represented symbolically, using the tens rule, with the following exception. When represented without the Chinese era, the year is represented using the Arabic numeral system. For example, when unaccompanied with the era, the 2004 is represented 二〇〇四.

The symbols used to represent Chinese eras vary with locale. China represents eras using Simplified Chinese characters; whereas, Hong Kong and Macau use Traditional Chinese characters.

Thai Date Time Rules

A future version of this specification will discuss the rules for formatting dates in Thai language locales.

Picture Clause Reference

The following sections describe the categories of picture clauses:

- "Date Picture Clauses"
- "Time Pictures"
- "Numeric Pictures"
- "Text Pictures"

Conventions

This reference uses the following font faces to differentiate picture clause symbols:

`sss` and `SSS`	Regular symbols
ｈｈｈ and ＨＨＨ	Full-width symbols

Date Picture Clauses

Symbols used for date picture clauses apply to all locales ("Standard Date Picture Symbols") or apply primarily to Asian environments ("Asian Date Symbols"). The category of such picture clauses is identified as `date`.

Standard Date Picture Symbols

The standard date picture clause symbols are described in the following table.

Symbol	Is the picture symbol for …
`D`	1- or 2-digit (1-31) day of the month.
`DD`	Zero-padded 2 digit (01-31) day of the month.
`J`	1-, 2- or 3-digit (1-366) day of the year.
`JJJ`	Zero-padded 3 digit (001-366) day of the year.
`M`	1- or 2-digit (1-12) month of the year.
`MM`	Zero-padded 2 digit (01-12) month of the year.
`MMM`	Abbreviated month name of the prevailing locale.
`MMMM`	Full month name of the prevailing locale.
`E`	1-digit (1-7) day of the week, where 1 = Sunday.
`EEE`	Abbreviated weekday name of the prevailing locale.
`EEEE`	Full weekday name of the prevailing locale.

<table>
<tr><th>Symbol</th><th>Is the picture symbol for ...</th></tr>
<tr><td>e</td><td>1 digit (1-7) day of the week, where 1 = Monday. This symbol is used in the context of the ISO Week Date format, where weeks start on Mondays rather than Sundays.

Note: Expressions of the form eee and eeee are not supported because they duplicate the capability of the date picture symbols EEE and EEEE.</td></tr>
<tr><td>g</td><td>Alternate-era name of the prevailing locale, represented using alternate-era style 1. This symbol is used only in Asian locales. This symbol is more fully described on page 1008</td></tr>
<tr><td>gg</td><td>Alternate-era name of the prevailing locale, represented using alternate-era style 2. This symbol is used only in Asian locales. This symbol is more fully described on page 1008</td></tr>
<tr><td>ggg</td><td>Alternate-era name of the prevailing locale, represented using alternate-era style 3. This symbol is used only in Asian locales. This symbol is more fully described on page 1008</td></tr>
<tr><td>G</td><td>Christian era name (BC or AD).</td></tr>
<tr><td>Y</td><td>1- or 2-digit year, used in the context of Asian alternate eras. This symbol is more fully described on page 1009.</td></tr>
<tr><td>YY</td><td>2-digit year, where 00 = 2000, 29 = 2029, 30 = 1930, and 99 = 1999.</td></tr>
<tr><td>YYYY</td><td>4-digit year.</td></tr>
<tr><td>w</td><td>1-digit (0-5) week of the month. Week 1 of a month is the earliest set of four contiguous days in that month that ends on a Saturday.
<table>
<tr><th colspan="7">July 2004</th><th>Week number</th></tr>
<tr><td>Su</td><td>Mo</td><td>Tu</td><td>We</td><td>Th</td><td>Fr</td><td>Sa</td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td>1</td><td>2</td><td>3</td><td>0</td></tr>
<tr><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>1</td></tr>
<tr><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>2</td></tr>
<tr><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>3</td></tr>
<tr><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td><td>4</td></tr>
</table></td></tr>
<tr><td>WW</td><td>2-digit (01-53) ISO-8601 week of the year. Week 01 of a year is the week containing January 4.</td></tr>
<tr><td></td><td>See also "Global Picture-Clause Symbols" on page 993, which describes the symbols "?", "*", and "+".</td></tr>
</table>

Asian Date Symbols

The following table describes date symbols used primarily in Asian locales.

Symbol	Full-width (FW) or ideographic (I)	Is the picture symbol for …
D (U+FF24)	FW	1- or 2-digit full-width numeric value for the day of the month, (1-31)
DD	FW	Zero-padded 2-digit full-width numeric value for the day of the month (01-31)
DDD	I	Prevailing-locale ideographic numeric value for the day of the month.
DDDD	I, tens rule	Tens rule prevailing-locale ideographic numeric value for the day of the month.
J (U+FF2A)	FW	1, 2 or 3-digit full-width numeric value for the day of the year (1-366)
JJJ	FW	Zero-padded 3-digit full-width numeric value for the day of the year (001-366)
M (U+FF2D)	FW	1- or 2-digit full-width numeric value for the month of the year (1-12)
MM	FW	Zero-padded 2-digit full-width numeric value for the month of the year (01-12)
MMM	I	Prevailing-locale ideographic numeric- value for the month of the year
MMMM	I, tens rule	Tens rule prevailing-locale ideographic numeric value for the month of the year.
E (U+FF25)	I	1-digit (1-7) prevailing-locale's ideographic numeric value for the day of the week, where 1 = Sunday.
e (U+FF45)	I	1 digit (1-7) prevailing-locale's ideographic numeric value for the day of the week, where 1 = Monday. This symbol is used in the context of the ISO Week Date format, where weeks start on Mondays rather than Sundays.
g	Depends on locale	Alternate-era name of the prevailing locale, represented using alternate-era style 1. This symbol is meaningful only in Asian locales. In all other locales, the symbol specifies Christian era (BC/AD). See "Imperial (Alternate) Eras and Alternate Era Styles" on page 1003. See also the description for the full-width symbols "g (U+FF47)" and "gg" on page 1009.
gg	Depends on locale	Alternate-era name of the prevailing locale, represented using alternate-era style 2. This symbol is meaningful only in Asian locales. In all other locales, the symbol specifies Christian era (BC/AD).See "Imperial (Alternate) Eras and Alternate Era Styles" on page 1003. See also the description for the full-width symbols"g (U+FF47)" and "gg" on page 1009.

<table>
<tr><th>Symbol</th><th>Full-width (FW) or ideographic (I)</th><th>Is the picture symbol for …</th></tr>
<tr><td>ggg</td><td>Depends on locale</td><td>Alternate-era name of the prevailing locale represented using alternate-era style 3. This symbol is meaningful only in Asian locales. In all other locales, the symbol specifies Christian era (BC/AD). See “Imperial (Alternate) Eras and Alternate Era Styles” on page 1003. See also the description for the full-width symbols “ｇ (U+FF47)” and “ｇｇ” on page 1009.</td></tr>
<tr><td>ｇ (U+FF47)</td><td>Depends on locale</td><td>Alternate-era name of the prevailing locale represented using alternate-era style 4. This symbol is meaningful only in Asian locales. In all other locales, the symbol specifies Christian era (BC/AD). See “Imperial (Alternate) Eras and Alternate Era Styles” on page 1003.</td></tr>
<tr><td>gg</td><td>Depends on locale</td><td>Alternate-era name of the prevailing locale represented using alternate-era style 5. This symbol is meaningful only in Asian locales. In all other locales, the symbol specifies Christian era (BC/AD). See “Imperial (Alternate) Eras and Alternate Era Styles” on page 1003.</td></tr>
<tr><td>ｗ (U+FF57)</td><td>FW</td><td>1-digit (0-5) full-width numeric value for the week of the month. Week 1 of a month is the earliest set of four contiguous days in that month that ends on a Saturday.
<table>
<tr><th colspan="7">July 2004</th><th>Week number</th></tr>
<tr><td>Su</td><td>Mo</td><td>Tu</td><td>We</td><td>Th</td><td>Fr</td><td>Sa</td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td>1</td><td>2</td><td>3</td><td>0</td></tr>
<tr><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>1</td></tr>
<tr><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>2</td></tr>
<tr><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>3</td></tr>
<tr><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td><td>4</td></tr>
</table></td></tr>
<tr><td>ｗｗ (U+FF37)</td><td>FW</td><td>2-digit full-width numeric value for the ISO-8601 week of the year (01-53). Week 01 of a year is the week containing January 4.</td></tr>
<tr><td>Y</td><td>Standard Latin digits</td><td>1- or 2-digit year.

If any of the year symbols (Y, Y, YY, YYY, YYYY, or YYYYY) are preceded by an imperial era symbol (g, gg, etc), the year is given in terms of that era. Further, the first year of any era is always represented using the ideograph 元 (U+5143) rather than the full-width 1 or the full-width 01.

If this symbol is not preceded by an alternate-era symbol, the year is given according to the Gregorian calendar.

Note: The symbols Y and Y should be used only in the context of an imperial era, where the year has a reasonable single-digit representation.</td></tr>
</table>

Symbol	Full-width (FW) or ideographic (I)	Is the picture symbol for ...
Ｙ (U+FF39)	I	1- or 2- digit full-width numeric value for the year value. (See description for "Y".)
ＹＹ	I	2-digit full-width numeric value for the year value. (See description for "Y".)
ＹＹＹ	I	Prevailing-locale ideographic numeric value for year. (See description for "Y".)
ＹＹＹＹ	FW	Full-width 4 digit year value. (See description for "Y".)
ＹＹＹＹＹ	I, tens rule	Tens rule prevailing-locale ideographic numeric value for year. (See description for "Y".)
		See also "Global Picture-Clause Symbols" on page 993, which describes the symbols "**?**", "*****", and "**+**".

The standard and full-width (**,**), dash (**-**), colon (**:**), slash (**/**), period (**.**) and space ( ) are treated as literals.

Requirements for Acceptable Date Picture Clauses

This section provides guidelines for writing acceptable date picture clauses. Please note that a picture clauses used for input parsing have stricter guidelines than those used for output formatting.

Avoid Ambiguity in Date Picture Clauses Used for Input Parsing

Date picture clauses must be reasonably unambiguous; however, there are certain ambiguous date inputs that can reasonably resolved into unambiguous expressions.

Example 26.14 Un-ambiguous date picture clauses

```
MM/DD/YY
MM-DD-YY
DD.MM.YYYY
DD MMM YYYY
MMMM DD, YYYY
EEEE,' le 'D MMMM, YYYY
```

Example 26.15 Unacceptably ambiguous date picture clauses

Unacceptably ambiguous date picture clause	Explanation
`YYY`	The year cannot be reasonably deduced with the information provided.
`YYMD`	Date picture clauses with adjacent one letter picture symbols are ambiguous. With a picture clause of `YYMD`, an input of "`99121`" can be evaluated as either "`Jan 21, 1999`" or "`Dec 1, 1999`".

Example 26.16 Acceptably ambiguous date picture clauses

Acceptably ambiguous date picture clause	Explanation
MMDDYY	As a concession to present day realities, the two-digit years 00 to 29 are interpreted as the years 2000 to 2029, while the two-digit years 30 to 99 are interpreted as the years 1930 to 1999. This is known as the century split option, and the century split year is set by default to 30; it is expected that applications using picture clauses would be able to reconfigure the century split year. **Important**: It is strongly recommended that applications interchange data with fully specified years.

Avoid Multiple Occurrences of the Same Types of Symbols in Input Parsing

Date picture clauses used for input parsing must avoid multiple occurrences of symbols used for the same type of data. The following table provides examples of such unacceptable multiple occurrences.

Example 26.17 Picture clauses with multiple symbols that are unacceptable for parsing input

Unacceptable multiple sets of symbols	Explanation
DD/MM/DD	The DD symbol appears twice.
JJJ-DD-MMM-YY	The JJJ and DD symbols both format days.

When output formatting, date picture clauses with multiple instances of the same symbols are acceptable, as are date formats with conflicting symbols.

Examples of Output Formatting

As examples of output formatting date values, consider the following: results have been quoted, so that one can see where spaces would appear in the formatted value. The quotes are not actually part of the result.

Example 26.18 Output formatted using date picture clauses

Picture clause	Input value	Formatted result
MMMM DD, YYYY	2002-10-25	"October 25, 2002"
'Week of the month is' w	20040722	"Week of the month is 5"
e 'days after Sunday'	20040722	4 days after Sunday
YYYY-'W'WW-e	20040722	2004-W30-4 **Note:** This is the ISO Week Date format.
E 'days after Saturday'	20040722	5 days after Saturday
EEEE, 'the' D 'of' MMMM, YYYY	2000-01-01	"Saturday, the 1 of January, 2000"

Examples of Input Parsing

Example 26.19 Input parsed using date picture clauses

Picture clause	Input value	Parsed result
MM/D/YY	12/2/99	1999-12-02
MMM D, YYYY	Jan 10, 1999	1999-01-10

Examples of Asian Output Formatting and Input Parsing

Japanese Locale

The following table shows formatted results when the prevailing locale is Japanese. The same result is achieved by enclosing the picture close with a localization designator in the format `date(ja){picture clause}`. ("Locale-Specific Picture Clauses" on page 998)

Example 26.20 Input and output using picture clauses in the Japanese locale

Picture clause	Input value	Formatted result	Explanation
gY/M/D	2003-11-03	H15/11/3	Alternate-era style #1
ggY-M-D	1989-01-08	平1-1-8	Alternate-era style #2
ggYY/MM/DD	1989-11-03	昭01/11/03	Alternate-era style #5
ggY'年'M'月'D'日'	2003-11-03	平１５年１１月３日	Alternate-era style
gggYYY'年'MMM '月'DDD'日'	1989-01-08	平成元年一月八日	Alternate-era style 3, ideographic year. The 元 pictograph represents the first year in the Heisei impereal era.
gYY'年'MM'月'DD'日'	1989-12-23	H０１年１２月２３日	Alternate-era style 1
YYY'年'MMM'月'DDD'日'	1998-12-10	一九九八年一二月一〇日	Gregorian calendar year

Time Pictures

Symbols used for time picture clauses apply to all locales ("Standard Symbols") or apply primarily to Asian locales ("Asian Time Picture Symbols"). The category of such picture clauses is identified as `time`.

Standard Symbols

The standard picture clause symbols for time are:

Symbol	Is the picture symbol for ...
h	1 or 2 digit (1-12) hour of the meridiem (AM/PM), expressed as a 12-hour clock.
hh	2 digit (01-12) hour of the meridiem (AM/PM)), expressed as a 12-hour clock.
h	1- or 2-digit (1-12) hour of the meridiem (AM/PM).
hh	2-digit (01-12) hour of the meridiem (AM/PM).
k	1- or 2-digit (0-11) hour of the meridiem (AM/PM).
kk	2-digit (00-11) hour of the meridiem (AM/PM).
H	1- or 2-digit (0-23) hour of the day, expressed as a 24-hour clock.
HH	Zero-padded 2 digit (00-23) hour of the day, expressed as a 24-hour clock.
K	1- or 2-digit (1-24) hour of the day.
KK	Zero-padded 2 digit (01-24) hour of the day.
M	1- or 2-digit (0-59) minute of the hour.
MM	2-digit (00-59) minute of the hour.
S	1- or 2-digit (0-59) second of the minute.
SS	2-digit (00-59) second of the minute.
FFF	3-digit (000-999) thousandth of the second.
A	Meridiem name (AM or PM) of the prevailing locale.
Z	Abbreviated time-zone name (GMT, EST, GMT-00:30) of the prevailing locale.
z	ISO-8601 time-zone format: Z, +HH[MM], or -HH[MM]. In the examples at left, HH is a placeholder for a zero-padded 2-digit hour of the day, and the MM is a placeholder for a zero-padded 2-digit minute of the hour. The acceptable values for z are further described below: • Z. A time zone of 'Z' (Unicode character U+005A) indicates the time zone is 'zero meridian', or 'Zulu Time'. The [ISO-8601] section titled *Universal Time Coordinated* describes a method of defining time absolutely. Another helpful document is *A Few Facts Concerning GMT, UT, and the RGO*, by R. Langley, 20 January 1999, which is available at http://www.apparent-wind.com/gmt-explained.html. • +HH[MM] or -HH[MM]. A time zone expressed as an offset of plus or minus states that the offset can be added to the time to indicate that the local time zone is HH hours and MM minutes ahead or behind. The plus or minus sign must be included.
zz	Alternate ISO-8601 time-zone format: Z, +HH[:MM], or -HH[:MM]. The z and zz formats differ only in their use of the colon as a separator.
	See also "Global Picture-Clause Symbols" on page 993, which describes the symbols "**?**", "*****", and "**+**".

The standard and full-width comma (**,**), dash (**-**), colon (**:**), slash (**/**), period (**.**) and space ( ) are treated as literals.

Asian Time Picture Symbols

The following table lists the Asian picture clause symbols for time.

Symbol	Full-width (FW) or ideographic (I)	Is the picture symbol for …
h (U+FF48)	FW	1- or 2-digit full-width numeric value for the hour of the meridiem (AM/PM), expressed as a 12-hour clock. (1-12)
hh	FW	2-digit full-width numeric value for the hour of the meridiem (AM/PM)), expressed as a 12-hour clock. (01-12)
hhh	I	Prevailing locale's ideographic numeric value (1-12) for the hour of the meridiem.
hhhh	I, tens rule	Prevailing locale's tens rule ideographic numeric value (1-12) for the hour of the meridiem.
k (U+FF4B)	FW	1- or 2-digit full-width numeric value for the hour of the meridiem (AM/PM) (0-11)
kk	FW	2-digit (00-11) full-width numeric value for the hour of the meridiem (AM/PM).
kkk	I	Prevailing locale's ideographic numeric value (0-11) for the hour of the meridiem.
kkkk	I, tens rule	Prevailing locale's tens rule ideographic numeric value (0-11) for the hour of the meridiem.
H (U+FF28)	FW	1 or 2 digit (0-23) full-width numeric value for the hour of the day, expressed as a 24-hour clock.
HH	FW	Zero-padded 2-digit (00-23) full-width numeric value for the hour of the day, expressed as a 24-hour clock.
HHH	I	Prevailing locale's ideographic numeric value (0-23) for the hour of the day.
HHHH	I, tens rule	Prevailing locale's tens rule ideographic numeric value (0-23) for the hour of the day.
K (U+FF2B)	FW	1- or 2-digit (1-24) full-width numeric value for the hour of the day.
KK	FW	Zero-padded 2-digit (01-24) full-width numeric value for the hour of the day.
KKK	I	Prevailing locale's ideographic numeric value for the (1-24) hour of the day.
KKKK	I, tens rule	Prevailing locale's tens rule ideographic numeric value for the (1-24) hour of the day
M (U+FF2D)	FW	1- or 2-digit (0-59) full-width numeric value for the minute of the hour.
MM	FW	2-digit (00-59) full-width numeric value for the minute of the hour.

Symbol	Full-width (FW) or ideographic (I)	Is the picture symbol for …
MMM	FW	Prevailing locale's ideographic numeric value (0-59) for the minute of the hour.
MMMM	I, tens rule	Prevailing locale's tens rule ideographic numeric value (0-59) for the minute of the hour
S (U+FF33)	FW	1- or 2-digit (0-59) full-width numeric value for the second of the minute.
SS	FW	2-digit (00-59) full-width numeric value for the second of the minute.
SSS	I	Prevailing locale's ideographic numeric value (0-59) for the second of the minute.
SSSS	I, tens rule	Prevailing locale's tens rule ideographic numeric value (0-59) for the second of the minute.
FFF (U+FF26)	FW	3-digit (000-999) full-width numeric value for the thousandth of the second.
z (U+FF5A)	FW	Full-width ISO-8601 time-zone format: Z, +HH[MM], or -HH[MM]. All characters are full-width. In the above examples, H is a placeholder for an hour digit, and the M is a placeholder for a minute digit.
zz	FW	Alternate full-width ISO-8601 time-zone format: Z, +HH[:MM], or -HH[:MM].
		See also "Global Picture-Clause Symbols" on page 993, which describes the symbols "?", "*", and "+".

The standard and full-width comma (,), dash (-), colon (:), slash (/), period (.) and space ( ) are treated as literals.

Requirements for Acceptable Time Picture Clauses

When input parsing, time picture clauses containing hour of the meridiem symbols (h or k) without the meridiem symbol are invalid.

Examples of Output Formatting

The following examples illustrate the process of output formatting with time picture clauses. Results have been quoted, so that one can see where spaces would appear in the formatted value. The quotes are not actually part of the result.

Example 26.21 Output formatted using time picture clauses

Picture clause	Input value	Formatted result
h:MM A	11:11:11	"11:11 AM"
HH:MM:SS 'o''clock' A Z	11:11:11	"11:11:11 o'clock AM EDT"
h:MM A	14:30:59	"2:30 PM"
HH:MM:SS A Z	14:30:59	"14:30:59 PM EDT"

When input parsing with time picture clauses, a successfully parsed input value is returned as an ISO local time string of the form

```
HH[MM[SS[.FFF][z]]]
HH[MM[SS[.FFF][+HH[MM]]]]
HH[MM[SS[.FFF][-HH[MM]]]]
HH[:MM[:SS[.FFF][z]]]
HH[:MM[:SS[.FFF][-HH[:MM]]]]
HH[:MM[:SS[.FFF][+HH[:MM]]]]
```

Square brackets denote optional elements.

Examples of Input Parsing

As examples of input parsing time values, consider the following examples.

Example 26.22 Input parsed using time picture clauses

Picture clause	Input value	Parsed result
HH:MM	18:00	18
H.MM 'Uhr'	12.59 Uhr	12:59
h:MM:SS A Z	1:05:10 PM PST	17:05:10

Note: The return value in the last example will vary with the platform's timezone at the time the platform was parsing the input; the displayed result comes from a platform running on EDT.

Examples of Asian Output Formatting and Input Parsing

The following examples illustrate the process of input parsing with time picture clauses in a Asian locale.

Example 26.23 Input parsed using time picture clauses in Asian locales

Picture clause	Input value	Parsed result
time(ja){Ahh'時'MM'分'DD'秒'}	午後05時02分03秒	17:02:03
time(ko){KKK'時'MMM'分'SSS'秒'} This example specifies ideographic hours (24-hour clock), minutes and seconds. Although the picture symbols do not specify tens-rule, the numbers use it because all Korean ideographic numbers use tens rule.	십이時삼십오分사십	12:35:46
time(ja){Ahhhh'時'MMMM'分'SSSS'秒'} This example specifies tens-rule ideographic hours, minutes and seconds. The hours are meridiem.	午後十二時一分二秒	12:01:02
HH:MM zz	13:30 +01	13:30:00+01[a]

a.The parsed result reflects the current time zone. This result indicates the user resides in the Central European Time timezone. If the user resided elsewhere, the parsed result would differ.

Numeric Pictures

The following table describes the standard ASCII and full-width numeric picture clause symbols. The category of such picture clauses is identified as num.

Full-width symbols can be used in Asian locales to specify full-width data.

Symbol		
Standard ASCII	**Full-width**	**Is the picture symbol for …**
9	９ (U+FF19)	Output formatting: a single digit, or for the zero digit if the input data is empty or a space in the corresponding position. Input parsing: a single digit.
8	８ (U+FF18)	Output formatting: a single digit, or for nothing if the input data has nothing in the corresponding position. For more information see "Effect of the 8 Picture Symbol" on page 1020. Input parsing: a single digit or nothing.
z	ｚ (U+FF5A)	Output formatting: a single digit, or for nothing if the input data is empty, a space, or the zero digit in the corresponding position. For more information see "Uppercase Picture Symbols versus Lowercase Picture Symbols" on page 1019. Input parsing: a single digit or nothing.

Symbol		
Standard ASCII	**Full-width**	**Is the picture symbol for …**
Z	Z (U+FF3A)	Output formatting: a single digit, or for a space if the input data is empty, a space, or the zero digit in the corresponding position. For more information see "Uppercase Picture Symbols versus Lowercase Picture Symbols" on page 1019. Input parsing: a single digit or a space.
S	S (U+FF33)	Output formatting: a minus sign if the number is negative, and a space otherwise. Input parsing: a minus sign if the number is negative and a plus sign if the number is positive.
s	s (U+FF53)	Output formatting: a minus sign if the number is negative, and nothing otherwise. Input parsing, it is also the picture symbol for a plus sign if the number is positive.
E	E (U+FF25)	Output formatting: the exponent part of a floating point number, consisting of the exponential symbol (E), followed by an optional plus or minus sign, followed by the exponent value. Input parsing: Same as for output formatting.
$	$ (U+FF04)	Currency symbol of the prevailing locale. In cases where the symbol may be ambiguous, please use the **$$** symbol. For example, the dollar symbol applies equally to the Canadian dollar and the U.S. dollar.
$$	$$	International currency name of the prevailing locale, as described in [ISO-4217].
CR		Credit symbol (CR) if the number is negative, or spaces otherwise. **Note:** CR and DB are English-language accounting practices and may not be meaningful in other locales.
cr		Credit symbol (CR) if the number is negative, or nothing otherwise. **Note:** CR and DB are English-language accounting practices and may not be meaningful in other locales.
DB		Debit symbol (DB) if the number is negative, or spaces otherwise. **Note:** CR and DB are English-language accounting practices and may not be meaningful in other locales.
db		Debit symbol (DB) if the number is negative, or nothing otherwise. **Note:** CR and DB are English-language accounting practices and may not be meaningful in other locales.
(	((U+FF08)	Left parenthesis if the number is negative, or a space otherwise.

Symbol		
Standard ASCII	**Full-width**	**Is the picture symbol for ...**
)	) (U+FF09)	Right parenthesis if the number is negative, or a space otherwise.
. (period)	. (U+FF0E)	Decimal radix of the prevailing locale. See also "Uppercase Picture Symbols versus Lowercase Picture Symbols" on page 1019.
V	V (U+FF36)	Decimal radix of the prevailing locale, allowing the decimal radix to be implied when input parsing.
v	v (U+FF56)	Decimal radix of the prevailing locale, allowing the decimal radix to be implied when input parsing and output formatting.
, (comma)	, (U+FF0C)	Grouping separator of the prevailing locale
%		Percent symbol of the prevailing locale. For more information see "Effect of the % Picture Symbol" on page 1020.
		See also "Global Picture-Clause Symbols" on page 993, which describes the symbols "**?**", "*****", and "**+**".

The standard and full-width dash (**-**), colon (**:**), slash (**/**), and space ( ) are treated as literals. Unlike the other categories of picture clauses, the comma is omitted as a literal because it is used as the symbol for grouping separators.

Uppercase Picture Symbols versus Lowercase Picture Symbols

There are two differences between the uppercase **Z** picture symbol and the lowercase **z** picture symbol. The differences do not affect the parsing of input, only the formatting of output.

1. Lowercase **z** to the left of the **.** (period) picture symbol omits leading zeros, whereas uppercase **Z** in this position displays leading zeros as space characters. This also applies if the picture clause does not contain a **.** (period) picture symbol.

2. Lowercase **z** to the right of the **.** (period) picture symbol omits the radix point when the input number does not have it, whereas uppercase **Z** in this position always inserts the radix point.

Similar differences apply to uppercase Ｚ (U+FF3A) and lowercase ｚ (U+FF5A).

1. Lowercase ｚ (U+FF5A) to the left of the ． (U+FF0E) picture symbol omits leading zeros, whereas uppercase Ｚ (U+FF3A) in this position displays leading zeros as space characters. This also applies if the picture clause does not contain a ． (U+FF0E) picture symbol.

2. Lowercase ｚ (U+FF5A) to the right of the ． (U+FF0E) picture symbol omits the radix point when the input number does not have it, whereas uppercase Ｚ (U+FF3A) in this position always inserts the radix point.

The following table shows differences in the formatting of output strings when **Z** is used in place of **z** in a picture clause. The output strings are shown within quotation marks to make the string boundaries apparent, however the quotation marks would not be included in the formatted output.

Example 26.24 Differences between z and Z picture symbols in output

Input value	Output formatted by zz9.zzz	Output formatted by ZZ9.ZZZ
1.234	"1.234"	" 1.234"
12.345	"12.345"	" 12.345"
123.456	"123.456"	"123.456"
123	"123"	"123."
123.	"123."	"123."
123.0	"123.0"	"123.0"
123.000	"123.000"	"123.000"

Effect of the 8 Picture Symbol

The **8** picture symbol is used to retain the expressed precision of supplied data after the decimal radix. For example, suppose the picture clause is being used for output. The following table shows the output for various picture clauses and input strings:

Example 26.25 Input parsed using numeric pictures containing the '8' symbol

Picture clause	Input value	Formatted result
zzz,zz9.8888	123456.000	123,456.000
	123456.0	123,456.0
	123456	123,456
zzz,zz9.88	123456	123,456

Note that, as for the lower-case **z** picture symbol, when the input number is an exact integer the decimal radix is removed from the formatted result.

Effect of the % Picture Symbol

The % picture symbol indicates that the input data is a percentage and ends in the percentage symbol of the prevailing locale. On input the percentage symbol is stripped out and the numeric value is converted to canonical form by multiplying it by 100. On output the canonical value is divided by 100 and the percentage symbol of the prevailing locale is appended.

Example 26.26 Input and output using numeric picture clauses containing the '%' symbol

Picture clause	Input value	Internal (canonical) value	Output value
zz9%	12%	0.12	12%
zzz,zz9.99%	1,234.5%	12.345	1,234.50%

Requirements for Acceptable Number Picture Clauses

When the parentheses picture symbols are used in a numeric picture clause, they must be paired, left with right, and must enclose all occurrences of **9**, **z**, and **Z** picture symbols.

When the **E** picture symbol is used in a numeric picture clause, it must follow all occurrences of the **9**, **z**, and **Z** picture symbols.

When the **8** picture symbol is used in a numeric picture clause, it must not be intermixed to the right of the **.** picture symbol with either **9** or **Z** picture symbols.

When the **%** picture symbol is used in a numeric picture clause, it must follow all occurrences of the **9**, **z**, and **Z** picture symbols.

Also, the meaning of an **8** picture symbol to the left of a **.** picture symbol is not defined in this specification.

Example of Output Formatting

The application of picture clauses to numeric data can produce results such as numeric or monetary values. Consider the following examples. Results have been quoted, so that one can see where spaces would appear in the formatted value. The quotes are not actually part of the result.

Example 26.27 Output formatted using numeric picture clauses

Picture clause	Input value	Formatted results
S999v99	-1.23	"-00123"
S999V99	1.23	" 001.23"
	123	" 123.00"
SZZ9.99	12.3	" 12.30"
	-12.3	"- 12.30"
szz9.99	123	"123.00"
	-123	"-123.00"
$ZZ,ZZ9.99CR	1234	"$ 1,234.00 "
	-1234	"$ 1,234.00CR"
$z,zz9.99DB	1234	"$1,234.00"
	-1234	"$1,234.00DB"

Picture clause	Input value	Formatted results
99.999E	12345	12.345E+3
	.12345	12.345E-3

Examples of Input Parsing

Conversely, the application of picture clauses to numeric or monetary values can be used to produce numeric data results, as in the following example.

Example 26.28 Input parsed using numeric picture clauses

Picture clause	Input value	Parsed results
99V99	1050	10.50
	3125	31.25
99.999E	12.345E3	12345
	12.345E-2	.12345
z999	150	150
	0150	150
z,zz9.99	10.50	10.50
	3,125.00	3125.00
$z,zz9.99DB	$1,234.00	1234.00
	$1,234.00DB	-1234.00

If the distinction between various numeric picture symbols appears subtle, it's to provide the flexibility normally required when strict parsing and formatting rules are in place. For instance, a number value of 150 should be accepted given the picture clause `z999`; the addition of a leading zero to the value does not change the value from 150, i.e., the values 0150 and 150 are equivalent.

Similarly the picture clause `S9999` would accept the value -5000 or +5000 or 5000.

Examples of Picture Clauses Using Full-Width Digits

Full-width digits are normally used in Asian locales so that the digits fit into the same column width as ideographic characters. However picture clauses specifying full-width digits can be used in any locale.

Example 26.29 Output formatted using numeric picture clauses with full-width characters

In the examples below results have been quoted, so that one can see where spaces would appear in the formatted value. The quotes are not actually part of the result.

Picture clause	Input value (English language locale)	Formatted results
S999v99	-1.23	"-00123"
S999V99	1.23	" 001.23"
	123	" 123.00"
SZZ9.99	12.3	" 12.30"
	-12.3	"- 12.30"
szz9.99	123	"123.00"
	-123	"-123.00"
$ZZ,ZZ9.99s	1234	"$ 1,234.00 "
	-1234	"-$ 1,234.00"
99.999E	12345	12.345E+3
	.12345	12.345E-3

Text Pictures

The following table describes the text picture clause symbols. The category of such picture clauses is identified as `text`.

Symbol	Is the picture symbol for ...
A	Single alphabetic character.
X	Single character.
O (upper-case alphabetic character)	Single alphanumeric character.
Ø (zero)	Single alphanumeric character.
9	Single digit.
	See also "Global Picture-Clause Symbols" on page 993, which describes the symbols "**?**", "*****", and "**+**".

The standard and full-width comma (**,**), dash (**-**), colon (**:**), slash (**/**), period (**.**) and space ( ) are treated as literals.

Examples of Output Formatting

Example 26.30 Output formatted using text picture clauses

Results have been quoted, so that one can see where spaces would appear in the formatted value. The quotes are not actually part of the result.

Picture clause	Input value	Formatted result
A9A 9A9	K1S5K2	"K1S 5K2"
'+1 ('999') '999-9999	6135551212	"+1 (613) 555-1212"
999.999.9999	6135551212	"613.555.1212"

Examples of Input Parsing

Example 26.31 Input parsed using text picture clauses

In this example we have serial number data that requires the input to be of the form: three alphabetic characters, followed by four digits, followed by a single character of any type. A suitable text picture clause would therefore be:

```
AAA-9999-X
```

The following table shows the result of applying this picture clause as an input mask against various input strings. Results have been quoted, so that one can see where spaces would appear in the formatted value. The quotes are not actually part of the result.

Picture clause	Input value	Formatted result
AAA-9999-X	ABC-1234-5	"ABC12345"
	ABC-1234-D	"ABC1234D"
	123-4567-8[a]	""[a]

a.Note that the input data 123-4567-8 did not satisfy the input mask, and the resulting parsed value was an empty string.

Null-Category Picture Clauses

Null-category picture clauses specify output formatting for null values associated with non-image content types. They are always expressed with the "null" category designator and may use any of the symbols defined for numeric picture clauses.

```
null{numeric picture clause symbols}
```

See also "Compound Picture Clauses" on page 997 and "Numeric Pictures" on page 1017.

Null-category picture clauses are typically used in alternate picture clause expressions, such as the following.

Example 26.32 Null-category picture clause used as an alternate picture clause

```
null{'n/a'} | num{$z,zz9.99}
```

See also "Alternate Picture Clauses" on page 999.

Example 26.33 Null-category picture clause used to differentiate null and non-null data

Null-category picture clauses are especially useful when there is a difference between null-data and non-null data. In the example at right, the field labeled "Tax withheld" remains blank until the person filling out the form supplies a value.

The illustrated behavior is specified by the picture clause `null{}|num{$z,zz9.99}`.

Zero-Category Picture Clauses

Zero-category picture clauses specify output formatting for zero values associated with non-image content types. They are always expressed with the "`zero`" category designator and may use any of the symbols defined for numeric picture clauses.

```
zero{numeric picture clause symbols}
```

See also "Compound Picture Clauses" on page 997 and "Numeric Pictures" on page 1017.

As with null-category picture clauses, zero-category picture clauses are typically used in alternate picture clause expressions, such as the following.

Example 26.34 Zero-category picture clause used as an alternate picture clause

```
zero{9} | num{$z,zz9.99}
```

See also "Alternate Picture Clauses" on page 999.

Examples of Input Parsing Against Null- and Zero-Category Picture Clauses

The following table illustrates the affect of input parsing against a set of alternate picture clauses that include null-category and zero-category picture clauses. Results have been quoted, so that one can see where spaces would appear in the formatted value. The quotes are not actually part of the result. The word *null* without quotation marks represents null data.

Example 26.35 Input parsed against null- and zero-category alternate picture clauses

Picture clause	Input value	Parsed result
`null{'n/a'} | zero {9} | num{z,zz9.9}`	`""`	*null*
	"n/a"	*null*
	"0"	"0"
	"0.0"	"0"
	`"1,234.5"`	"1234.5"

Note: Order is important in expressions that use alternative picture clauses. If the picture clause `num{z,zz9.9}` is the first picture clause in the expression, the `zero{9}` picture clause is never considered.

Examples of Output Formatting Against Null- and Zero-Category Picture Clauses

The following table illustrates the affect of output formatting against a set of alternate picture clauses that include null-category and zero-category picture clauses.

Example 26.36 Output formatted by null- and zero-category alternate picture clauses

Picture clause	Input value	Formatted output
`null{'n/a'} | zero {9} | num{z,zz9.9}`	""	`"n/a"`
	"0"	"0"
	`"1234.5"`	"1,234.5"

Note: Order is important in expressions that use alternative picture clauses. If the picture clause `num{z,zz9.9}` is the first picture clause in the expression, the `zero{9}` picture clause is never considered.

27 Rich Text Reference

Rich Text is expressed in XFA via the use of [XHTML] markup augmented with a restricted set of Cascading Style Sheet attributes [CSS2], and a number of style attributes and values that are currently not part of the Cascading Style Sheet standard, but are proposed extensions. In addition there are several XFA-specific attributes, but these use an XFA namespace so that the content incorporating them is still valid XHTML/CSS.

The following sections describe the specific elements and style attributes of [XHTML] and [CSS2] that are supported. The accompanying descriptions and examples, however, are meant to be informative only. Please consult the [XHTML] and [CSS2] specifications as the normative reference for more detail on the capabilities and permissible values for the elements and style attributes.

XFA processors may implement a larger subset, or the entirety, of XHTML and CSS2 if desired, barring only markup that is deprecated in [XHTML] or [CSS2]. This specification sets out a minimum set that must be supported.

XFA processors ignore any markup in rich text that they do not understand. For example, XFA processors do not support the `head` element of [XHTML]. When an XFA processor encounters an unrecognized element such as this, it ignores the entire content of the element.

Summary of Supported XHTML and CSS Attributes

XFA processors support the following [XHTML] elements:

Element name	Where described
`b`	"Bold" on page 1035
`br`	"Line Break" on page 1030
`body`	"Body Element" on page 1029
`html`	"HTML Element" on page 1028
`i`	"Italic" on page 1038
`p`	"Paragraph" on page 1031
`span`	"Span" on page 1038
`sub`	"Subscript" on page 1038
`sup`	"Superscript" on page 1039

XFA processors support the following [CSS2] style attributes on the above-listed [XHTML] elements:

Attribute name	Where described
`color`	"Color" on page 1035

Attribute name	Where described
font	"Font" on page 1036
font-family	
font-size	
font-stretch	
font-style	
font-weight	
margin	"Set Margins" on page 1032
margin-bottom	"Space After Paragraph" on page 1032
margin-left	"Left Margin" on page 1030
margin-right	"Right Margin" on page 1031
margin-top	"Space Before Paragraph" on page 1033
line-height	"Line Spacing" on page 1030
tab-interval	"Tab Stops" on page 1039
tab-stop	
text-decoration	"Underline and Strikethrough" on page 1041
text-indent	"First Line Indent" on page 1029
vertical-align	"Vertical Alignment" on page 1033

This specification supports the following subset of allowable [CSS2] measurement units:

- cm -- centimeters
- in -- inches
- mm -- millimeters
- pt -- points

Supported Container Elements

XFA supports the optional use of XHTML container elements to enclose rich text. In addition, because unrecognized elements are suppressed, rich text used in XFA can take the form of a complete XHTML document including a `head` element, albeit a document restricted to a subset of XHTML and CSS2.

HTML Element

The outer element for HTML documents is an `html` element, as specified in [XHTML]. In XFA rich text may be enclosed in an `html` element, although this is not required. The `html` element is merely a container and does not appear in the output.

Body Element

The displayable content of an HTML document is contained in a `body` element, as specified in [XHTML]. In XFA rich text may be enclosed in a `body` element, although this is not required. It may also be enclosed in a `body` element that is itself within an `html` element. The `body` element is merely a container and does not appear in the output.

Supported Paragraph Formatting

First Line Indent

A paragraph of text may be indented on the first line (temporarily adjusting the left margin) via the use of the [CSS2] `text-indent` style attribute with a measurement.

Example 27.1 Paragraph with first-line indent

```
<p style="text-indent:0.5in">
   The first line of this paragraph is indented a half-inch.<br />
   Successive lines are not indented.<br />
   This is the last line of the paragraph.<br />
</p>
```

Produces:

```
      The first line of this paragraph is indented a half-inch.
Successive lines are not indented.
This is the last line of the paragraph.
```

Horizontal Alignment

A paragraph of text may be aligned horizontally via the use of the [CSS2] `text-align` style attribute.

The supported alignments are limited to left, center, right, justify, and justify-all. justify-all is a non-standard extension to [CSS2] expressing that all lines of the paragraph including the last line shall be justified.

Example 27.2 Paragraph with right text alignment

```
<p style="text-align:right">
  This is the first line of the paragraph.<br />
  This is the second line of the paragraph.<br />
  This is the last line of the paragraph.<br />
</p>
```

Produces:

```
                                  This is the first line of the paragraph.
                                 This is the second line of the paragraph.
                                   This is the last line of the paragraph.
```

Left Margin

A paragraph of text may be indented on the left (adjusting the left margin) via the use of the [CSS2] `margin-left` style attribute with a measurement.

Example 27.3 Paragraph with left text alignment

```
<p style="margin-left:0.5in">
  This text is left-indented a half-inch.
</p>
```

Produces:

```
      This text is left-indented a half-inch.
```

Line Break

Within a paragraph, `br` elements may be used to force line breaks as defined in [XHTML].

Example:

```
<p>This is a paragraph of text.<br/>This is some more text.</p>
```

Produces:

```
This is a paragraph of text.
This is some more text.
```

Line Spacing

A paragraph of text may have the line-spacing of its text set via the use of the [CSS2] `line-height` style attribute with a measurement. By default, line-spacing is derived from the tallest object on any given line.

Example 27.4 Paragraph with line spacing

```
<body>
   <p style="line-height:0.5in">
     This is the first line of the paragraph.<br />
     This is the second line of the paragraph.<br />
     This is the last line of the paragraph.<br />
   </p>
</body>
```

Produces:

```
This is the first line of the paragraph.

This is the second line of the paragraph.

This is the last line of the paragraph.
```

Paragraph

A paragraph of text is expressed using the [XHTML] paragraph `p` element. By default text contained within the `p` element flows from left-to-right, word-wrapping as necessary to fit within the left and right margins. By default consecutive white space characters are compressed to a single space character. `p` elements cannot nest, nor can they hold `html` or `body` elements.

Example 27.5 Paragraph markup

```
<p>This is a paragraph of text. This is some more text.</p>
```

Produces:

```
This is a paragraph of text. This is some more text.
```

The paragraph element may have attributes that format the paragraph, as described in the following subsections. The left-to-right flow may be interrupted by a `br` element, as described in "Line Break" on page 1030.

Right Margin

A paragraph of text may be indented on the right (adjusting the right margin) via the use of the [CSS2] `margin-right` style attribute with a measurement. This is most commonly used in concert with a right text alignment, as described in Horizontal Alignment.

Example 27.6 Paragraph with a right margin

```
<p style="margin-right:0.5in;
         text-align:right">
  This text is right-aligned and right-indented a half-inch.
</p>
```

Produces:

```
              This text is right-aligned and right-indented a half-inch.
```

Set Margins

One or more margins may be adjusted, affecting paragraph spacing and indenting, through the use of the [CSS2] `margin` style attribute which accepts a variable number of arguments affecting the top, bottom, left, and right margins. Use of this attribute provides no additional features beyond the individually addressable features described in sections "Space Before Paragraph", "Space After Paragraph", "Line Spacing", "Left Margin", "Right Margin", and "First Line Indent". The syntax of the `margin` style attribute is explained in the following excerpt from the [CSS2] specification:

> *If there is only one value, it applies to all sides. If there are two values, the top and bottom margins are set to the first value and the right and left margins are set to the second. If there are three values, the top is set to the first value, the left and right are set to the second, and the bottom is set to the third. If there are four values, they apply to the top, right, bottom, and left, respectively.*

Example 27.7 Paragraph setting a margin all around

```
<body>
   <p>
      This is the first paragraph.
   </p>
   <p style="margin:0.5in">
      This second paragraph has a half-inch margin on all sides.
   </p>
   <p>
      This is the third paragraph.
   </p>
</body>
```

Produces:

```
This is the first paragraph.



      This second paragraph has a half-inch margin on all sides.



This is the third paragraph.
```

Space After Paragraph

A paragraph of text may have additional succeeding vertical space via the use of the [CSS2] `margin-bottom` style attribute with a measurement.

Example 27.8 Paragraph setting a bottom margin

```
<body>
   <p style="margin-bottom:0.5in">
      This paragraph is spaced a half-inch away from the next paragraph.
   </p>
   <p>This is a paragraph of text.</p>
</body>
```

Produces:

```
This paragraph is spaced a half-inch away from the next paragraph.



This is a paragraph of text.
```

Space Before Paragraph

A paragraph of text may have additional preceding vertical space via the use of the [CSS2] `margin-top` style attribute with a measurement.

Example 27.9 Paragraph setting a top margin

```
<body>
   <p>This is a paragraph of text.</p>
   <p style="margin-top:0.5in">
      This paragraph is spaced a half-inch away from the previous
      paragraph.
   </p>
</body>
```

Produces:

```
This is a paragraph of text.



This paragraph is spaced a half-inch away from the previous paragraph.
```

Vertical Alignment

A paragraph of text may be aligned vertically via the use of a `text-valign` style attribute which is a non-standard extension to [CSS2].

The supported alignments are limited to top, middle, and bottom.

Example 27.10 Paragraph setting a vertical alignment

```
<p style="text-valign:middle">
   This is the first line of the paragraph.<br />
   This is the second line of the paragraph.<br />
   This is the last line of the paragraph.<br />
</p>
```

Produces:

```


This is the first line of the paragraph.
This is the second line of the paragraph.
This is the last line of the paragraph.


```

Supported Character Formatting

The following sections describe the various formatting elements that may be applied to paragraph content. In addition it describes attributes that may be applied to modify the effect of the elements. Each character formatting element encloses a section of text and supplies formatting attributes to that text.

Character formatting elements are low-level objects. They must not enclose higher-level elements that are recognized by XFA. Hence they must not enclose `html`, `body`, or `p` elements.

Character formatting attributes may be applied to `p` elements, in which case they affect all of the text in the paragraph. Character formatting attributes may also be applied to `span` elements, in which case they apply to the text enclosed by the `span` element. Enclosed markup must not conflict with enclosing markup. For example, it is not permissible to set a baseline on a paragraph and a different baseline on a span within the paragraph. Instead the markup must be flattened so that each section is enclosed in its own `span` element and marked up separately.

Baseline Adjustment

A region of text may have its baseline position raised or lowered within a line of text via the use of the [CSS2] `vertical-align` style attribute. The font size of the text is not affected.

This specification supports a restricted set of [CSS2] vertical-align formats, hence, this attribute is defined as:

`vertical-align:`*`vAlginValue`*

where *`vAlignValue`* is either the keyword baseline or a measurement.

If *`vAlignValue`* is baseline, the region of text will have its baseline situated on the calculated baseline for the surrounding line of text. Note that this behavior differs from [CSS2] where the region of text has its baseline situated on the calculated baseline for the surround parent `span` element rather than the surrounding line.

If *`vAlignValue`* is a measurement, the region of text will have its baseline raised for a positive measurement or lowered for a negative measurement in relation to the calculated baseline for the surrounding line of text. A measurement of zero (0) produces the same result as `vertical-align:baseline`.

Example 27.11 Paragraph including raised and lowered spans of text

```
<p>This sentence contains
   <span style="vertical-align:-3pt">lowered text</span>
   on a line. <br />
   <br /> Most <span style="vertical-align:-4pt">
   of this sentence is lowered but this
   </span>word<span style="vertical-align:+4pt">
   appears on the </span> line's baseline. <br />
   <br /> This sentence contains
   <span style="vertical-align:3pt">
   raised text</span> on a line.
</p>
```

Produces:

```
This sentence contains lowered text on a line.

 Most of this sentence is lowered but this word appears on the line's baseline.

This sentence contains raised text on a line.
```

Superscripts and subscripts can be implemented using a combination of a baseline adjustment and font size adjustment. However, vertical alignment can also be adjusted using `sub` (subscript) and `sup` (superscript) elements.

Bold

A region of text may be in bold type via the use of the [CSS2] `font-weight` style attribute or the `b` [XHTML] element.

Example 27.12 Paragraph including bolded spans of text

```
<p>The<b> second </b>and
<span style="font-weight:bold"> fourth </span>
words are bold.
</p>
```

Produces:

The **second** and **fourth** words are bold.

Color

A color may be specified for a region of text via the [CSS2] `color` style attribute. Color rendition is device- and implementation-specific so the text may be rendered in a color other than the specified color. Conforming implementations are merely required to make their best effort to render the requested color. For example, when printing with a monochrome printer, colors other than black and white are usually rendered as grey.

This specification supports a restricted set of [CSS2] color-value formats, defined as:

`color:`*`colorValue`*

`colorValue` values are described in the following table.

"colorValue" value	Specifies color's RGB value as ...
`#rrggbb`	Hexadecimal notation with a two-digit non-negative hexadecimal value each for red, green, and blue. A value of `00` means the color is absent (zero-intensity) whereas `ff` means it is at full intensity. Digits a through f may be upper or lower case.
`rgb(r,g,b)`	Separate non-negative integer decimal values for red, green, and blue. A value of 0 means the color is absent (zero-intensity) whereas 255 means it is at full intensity.

The value is an RGB value specified in the sRGB color space [SRGB].

Example 27.13 Colored paragraph including a differently-colored span

```
<p style="color:#0000ff">All of this text is blue except for
this <span style="color:rgb(0,255,0)"> green </span>
word.</p>
```

Produces:

All of this text is blue except for this green word.

Font

A region of text may be in a specific font via the use of one or more of the [CSS2] `font` style attributes; the deprecated `font` HTML element is not supported.

The [CSS2] specification provides several style attributes that affect the current font.

Font style attribute	Description
`line-height:`*`lineHeight`*	*`lineHeight`* is a measurement giving the distance between baselines of adjacent lines
`font-family:`*`fontFamilyName`*	*`fontFamilyName`* is a list of one or more typeface names. The list constitutes a search path such that the glyph for any particular character is taken from the first font on the list which contains a glyph for that character. If a typeface name contains white space, it must be enclosed either within single quote (') characters or within double quote (") characters. Such punctuation allows the font name to be parsed as a single parameter. **Note:** Acrobat does not support multiple typeface names.
`font-size:`*`characterHeight`*	*`characterHeight`* is a measurement giving the height of a full-height capital letter
`font-stretch:`*`stretchName`*	*`stretchName`* is one of the following, in order from narrowest to widest: • ultra-condensed • extra-condensed • condensed • semi-condensed • normal, the default • semi-expanded • expanded • extra-expanded • ultra-expanded **Note:** Acrobat does not support this attribute.

Font style attribute	Description
`font-style:`*`styleName`*	*`styleName`* is one of: • `normal`, the text appearance will be normal (Roman) • `italic`, the text appearance will be oblique or slanted
`font-weight:`*`weightName`*	*`weightName`* is one of the following: • normal, the text appearance will be at the font's normal (default) weight • bold, the text appearance will be in a bold type • 100, 200, 300, 400, 500, 600, 700, 800, or 900, the text appearance will approximate as closely as possible a weight corresponding to the number, using a scale upon which normal is equivalent to 400 and bold is equivalent to 700 **Note:** Acrobat does not support numeric weights.

Alternatively, multiple font properties may be defined at once using the following syntax:

```
font:[ styleName ] [ weightName ] characterHeight [ / lineHeight ] fontFamilyName
```

or

```
font:[ weightName ] [ styleName ] characterHeight [ / lineHeight ] fontFamilyName
```

where *`styleName`*, *`weightName`*, *`characterHeight`*, *`lineHeight`*, and *`fontFamilyName`* are as defined above. The character "/" (U002F) preceding *`lineHeight`* is a literal. The *`styleName`* and *`weightName`* parameters, if present, can be specified in either order. When an optional parameter is omitted, the result is to set the corresponding property to its default. This syntax is a subset of the corresponding [CSS2] syntax.

Example 27.14 Paragraph declaring a font with spans in different fonts

```
<p style='font:italic bold 16pt/0.5in "Minion Pro"'>
 The base font for this text is Minion Pro italic bold 16pt
 with a line-spacing of a half-inch.
 <span style="font-family:'Courier Std'">
   The second sentence switches to a Courier Std typeface.
 </span>
 <span style="font-size:12pt">
   The last sentence switches to a 12-point font.
 </span>
</p>
```

Produces:

The base font for this text is Minion Pro italic bold 16pt with a line-spacing of a half-inch. The second sentence switches to a Courier Std typeface. The last sentence switches to a 12-point font.

Italic

A region of text may be italicized via the use of the [CSS2] `font-style` style attribute or the `i` [XHTML] element.

Example 27.15 Paragraph including italicized text

```
<p>The<i> second </i>and
<span style="font-style:italic"> fourth </span>
words are italicized.
</p>
```

Produces:

The *second* and *fourth* words are italicized.

Span

The `span` element has no formatting effect of its own, but it accepts formatting attributes and applies those attributes to whatever it encloses, as defined by the [CSS2] and [XHTML] specifications. However XFA imposes an additional restriction, namely that `span` elements can not nest.

Subscript

A region of text may have its baseline position lowered within a line of text via the use of the [XHTML] `sub` element. The effect is to lower the baseline by 15% of the current font height, and to set a new font height which is 66% of the current font height.

Example 27.16 Paragraph including a subscripted span of text

```
<p>This sentence contains
   <sub>lowered text</sub>
   on a line.
</p>
```

Produces:

This sentence contains _{lowered text} on a line.

Subscripts with other baseline and height parameters can be achieved using a combination of the `vertical-align` and `line-height` style attributes.

`sub` elements must not contain `sub` or `sup` elements. In addition they must not contain `span` elements that assert `vertical-align` or `font-size`.

Superscript

A region of text may have its baseline position raised within a line of text via the use of the [XHTML] `sup` element. The effect is to raise the baseline by 31% the current font height, and to set a new font height which is 66% of the current font height.

Example 27.17 Paragraph including a superscripted span of text

```
<p>This sentence contains
   <sup>raised text</sup>
   on a line.
</p>
```

Produces:

This sentence contains $^{\text{raised text}}$ on a line.

Superscripts with other baseline and height parameters can be achieved using a combination of the `vertical-align` and `line-height` style attributes.

`sup` elements must not contain `sub` or `sup` elements. In addition they must not contain `span` elements that assert `vertical-align` or `font-size`.

Tab Stops

Tab stops are not a feature provided by [CSS2] or the [XHTML] specification, however, the following extensions are provided for setting tab-stops at either a repeating interval or at specific locations. For compatibility with [XHTML], tabs are invoked using an element with a style attribute. The ASCII tab character (U0009) is ordinary white space that does not advance to the next tab-stop.

Default tab stops may be set at a repeating interval via the use of a nonstandard [CSS2] style attribute tab-interval. The default tab stops occur at every multiple of the specified measurement value. However, these tab stops are in effect only beyond the positions specified by the nonstandard [CSS2] style attribute tab-stops described below. As a consequence, if no tab stops were defined with the tab stops attribute then all of the default tab stop positions will be in effect.

Default tab stops are always left tabs, that is, after advancing to the tab stop subsequent text is left-aligned with the tab stop.

This attribute is defined as:

```
tab-interval:size
```

where size is a non-zero measurement

A `span` element with a style attribute of `xfa-tab-count` may be used to advance by a specific number of tab-stops relative to the current position. The `span` element is recommended to be empty, as any text or markup that is contained within the span may be discarded by an XFA processing application.

```
xfa-tab-count:count
```

where count is a non-negative integer representing the number of tabs-stops to advance

When count is zero the xfa-tab-count attribute must have no effect.

Example 27.18 Paragraph using regular tab stops

```
<p style="tab-interval:0.5in">A
<span style="xfa-tab-count:2" />B
<span style="xfa-tab-count:1" />C</p>
```

Produces:

```
A           B     C
```

Tab-stops may be set at specific locations via the use of a nonstandard [CSS2] style attribute `tab-stops`.

This attribute is defined as:

```
tab-stops:align measurement[ align measurement]...
```

Note: This document uses the *`monospaced italics`* type face to indicate placeholders. In actual use, meaningful values replace such placeholders.

`align` values are described in the following table.

"align" value	Effect
`center`	Center-aligned tab stop
`left`	Left-aligned tab stop
`right`	Right-aligned tab stop
`decimal`	Tab-stop that aligns content around a radix point

A center-aligned tab stop causes the text following the tab to be centered on the tab stop position. A left-aligned tab stop causes the text following the tab to be left-aligned with the tab stop position. A right-aligned tab stop causes the text following the tab to be right-aligned with the tab stop position. A radix-aligned tab stop is used with numeric data. It causes the text following the tab to be aligned with its radix character (for example, in English-speaking locales) left-aligned with the tab stop position. If the text has no radix character the text is right-aligned with the tab stop position. The determination of the appropriate radix character for the locale is implementation-defined.

The cursor position starts at the left margin and the tab index starts at zero upon entry to the element that declares the tab-stops. Within a `p` element that declares tab-stops, a `br` element restarts the cursor position at the left margin and the tab index at zero.

Example 27.19 Paragraph using left and right tab stops

```
<p style="tab-stops:left 0.5in left 2in">
<span style="xfa-tab-count:1" />Charles
<span style="xfa-tab-count:1" />Porter<br />
<span style="xfa-tab-count:1" />Alice
<span style="xfa-tab-count:1" />Crawford
</p>
```

Produces:

```
      Charles           Porter
      Alice             Crawford
```

Example 27.20 Paragraph using decimal tab stops

```
<p style="font-family:courier;tab-stops:decimal 0.5in">
<span style="xfa-tab-count:1" />1.2345<br />
<span style="xfa-tab-count:1" />99<br />
<span style="xfa-tab-count:1" />-.033<br />
<span style="xfa-tab-count:1" />$ 17.60 plus tax
</p>
```

Produces:

```
     1.2345
    99
     -.033
  $ 17.70 plus tax
```

Underline and Strikethrough

A region of text may be underlined or struck through using the [CSS2] `text-decoration` style attribute. The deprecated `u` and `s` HTML elements must not be supported by XFA applications.

The [CSS2] `text-decoration` style attribute only provides for a single-continuous underline. This specification allows for several nonstandard extensions to the [CSS2] `text-decoration` style attribute, to allow for combinations of single and double continuous and word-broken underlines.

Underlining is rendered on the baseline in the same color as the associated text.

This attribute is defined as:

`text-decoration:`*`decorationStyle`* `[` *`line-through`* `]`

or

`text-decoration:` *`line-through`* `[` *`decorationStyle`* `]`

The following table describes the *`decorationStyle`* values.

"decorationStyle" value	Produces …
`underline`	Single continuous underline
`word`	Single underline that breaks at word boundaries
`double`	Double continuous underline
`double word`	Double underline that breaks at word boundaries

Note that `double word` has whitespace between the words, not a hyphen.

Example 27.21 Paragraph using underline

```
<p>The
```

```
<span style="text-decoration:underline">second</span> and
<span style="text-decoration:underline">fourth</span>
words are underlined.
</p>
```

Produces:

The second and fourth words are underlined.

Example 27.22 Paragraph using line-through and underline

```
<p>The
<span style="text-decoration:line-through underline">second</span>
and <span style="text-decoration:line-through">fourth</span>
words appear with strikethrough.
</p>
```

Produces:

The ~~second~~ and ~~fourth~~ words appear with strikethrough.

The above example also demonstrates that a region of text may have both an underline and strikethrough by applying the [CSS2] `text-decoration` style attribute with multiple values.

Retaining Consecutive Spaces (xfa-spacerun:yes)

Normally, an XFA processing application removes consecutive spaces, as described in "White Space Handling" on page 131; however, this behavior can be overridden in rich text.

A run of consecutive normal spaces may be represented via the use of a `span` element in conjunction with a nonstandard [CSS2] style attribute `xfa-spacerun`. An XFA processing application that supports this feature must interpret each non-breaking-space character within the span as representing a normal space character (character 32).

In order to allow this `span` element to produce an approximate visual rendering, when processed by a browser or other [XHTML] processing application, the `span` element may contain a mixture of non-breaking-space and normal space characters. While the application that supports this `xfa-spacerun` style must interpret each non-breaking-space character within the span as representing a normal space character (character 32), the browser or other [XHTML] processing application will likely ignore the `xfa-spacerun` style and provide a rendering that may be visually similar.

Example 27.23 Paragraph using space runs

```
<p style="font-family:Courier">
  Two spaces here: <span style="xfa-spacerun:yes">  </span>
  and three spaces here: <span style="xfa-spacerun:yes">
     </span>before the next word.</p>
```

Note: The entity ` ` is a non-breaking space.

Produces:

```
Two spaces here:  ; and three spaces here:   before the next word.
```

The same result as above is produced by an application that supports this feature regardless of whether the `span` element contains non-breaking-space or normal-space characters.

Example 27.24 Same result as preceding using normal spaces

```
<p style="font-family:Courier">Two spaces here:
   <span style="xfa-spacerun:yes"> </span>
   and three spaces here:
   <span style="xfa-spacerun:yes">  </span>before the next word.</p>
```

Produces:

```
Two spaces here:  and three spaces here:   before the next word.
```

An application that supports this feature must process any character other than a non-breaking-space or normal-space character as normal text content.

Example 27.25 xfa:spacerun is ignored for non-space characters

```
<p style="font-family:Courier">All of
   <span style="xfa-spacerun:yes">this is</span> ordinary text.</p>
```

Produces:

```
All of this is ordinary text.
```

Embedded Object Specifications

The span elements within rich text may contain attributes that reference external plain-text or rich-text objects. Such external references are resolved during the rendering process and the referenced data is inserted at the point where the external reference appears. See "Rich Text That Contains External Objects" on page 193.

```
<span

   xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
   xfa:embedType="uri | som"
   xfa:embed="<SOM expr> | <uri>"
   xfa:embedMode="raw | formatted"
/>
```

xfa:embedType

An attribute that specifies the type of reference in xfa:embed.

som

The value of xfa:embed is a SOM expression.

uri

The value of xfa:embed is a URI.

xfa:embed

An attribute that provides a SOM or URI reference to the text being embedded.

xfa:embedMode

An attribute that specifes whether styling markup specification in the imported text should be respected.

formatted

Inserts the object with text styling preserved.

raw

Inserts the object, ignoring text styling.

Version Specification

In the HTML body element, the XFA grammar defines several attributes that specify the version of rich text supported and the version of the API used to produce the rich text.

Example 27.26 Rich text using version declarations

```
<body xfa:APIVersion="1.3.1253.0"
   xfa:spec="2.0.2"
   xmlns="http://www.w3.org/1999/xhtml"
   xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
   <p>This is a paragraph of text.</p>
```

```
</body>
```

The intent of these attributes is to provide a way for:

- Correlating updates to this specification with the software that processes the rich text
- Allowing future revisions of the software to apply special processing to the rich text that was produced by earlier releases of the software

Both attributes take version identifiers as values.

xfa:APIVersion

A concatenation of numeric fields that specify the version of the software used to produce the enclosing rich text. Numeric fields are separated from one another with periods, and each numeric field contains a whole number. The larger the number, the more recent the version. The further left the field, the more significant it is.

When comparing two version identifiers, if one identifier has fewer fields than the other, the one with fewer fields must be extended by padding it on the right with fields containing “0”.

The syntax of `xfa:APIVersion` is

```
xfa:APIVersion=releaseID
```

where *releaseID* is a version identifier representing the version of the software that produced the document.

If the `xfa:APIVersion` attribute is omitted from rich text, it should be assumed that the application producing the rich text has used markup attributes that correspond to the latest revision of this specification (*XFA Specification*).

The features described by this specification are supported by Adobe XFA rich text processing software bearing an `xfa:APIVersion` attribute of "2.5.6290.0" or later.

xfa:spec

The version of the XFA specification to which the rich text string complies.

```
spec=revisionID
```

Part 4: Appendices, Bibliography, Glossary and Index

This part contains appendices that provide adjunct information referenced by the narrative chapters in Part 1. It also contains a bibliography, a glossary and an index.

A Algorithms for Determining Coordinates Relative to the Page

This appendix provides algorithms that can be used to determine an object's position relative to the page coordinates. For brevity, within this section a container is referred to as a *parent* and the contained object as a *child*. A single container object may be parent to one object and child to another.

Suppose, for example, a child object places its internal origin at the top-left corner of its nominal content region (the typical case). If we have a point (Cx,Cy) in child coordinates, we can generate common coordinates (CCx,CCY) for the parent with the following simple equations:

CCx = Cx + Mx

CCY = Cy + My

where Mx and My are the child's left and top margin insets, respectively.

In order to convert these common coordinates into its own space, the parent must first determine the origin (Ox,Oy) of the child's top-left corner in its (the parent's) own coordinate space. These would be computed from the child's anchor point (Ax,Ay), using the child's nominal extent's width and height (W,H) as follows:

Ox = Ax (TopLeft, MiddleLeft, BottomLeft)

Ox = Ax - W/2 (TopCenter, MiddleCenter, BottomCenter)

Ox = Ax - W (TopRight, MiddleRight, BottomRight)

Oy = Ay (TopLeft, TopCenter, TopRight)

Oy = Ay - H/2 (MiddleLeft, MiddleCenter, MiddleRight)

Oy = Ay - H (BottomLeft, BottomCenter, BottomRight)

Now, it's a very simple transformation to generate parent coordinates (Px,Py) from common coordinates:

Px = CCx + Ox

Py = CCy + Oy

Or,

Px = Cx + Mx + Ox

Py = Cx + My + Oy

A slight optimization could be to avoid recalculating the invariants (Mx+Ox, My+Oy) through a little cooperation between the parent and the child.

B Layout Objects

The following table lists the characteristics of all the different layout objects.

Characteristics of layout objects

Layout object	Used as …	
arc	**boilerplate geometric figure**	
	layout strategy	Not a container
	break control?	No
	natural size?	Inherited, except line thickness
	growable?	All except line thickness. No limit
	splittable?	No
	multiply-occurring?	No
	container for	N/A
area	**physical and logical grouping of objects**	
	layout strategy	Positioned
	break control?	No
	natural size?	Defined by contained objects
	growable?	Yes (but area boundaries are not visible so this has no concrete effect)
	splittable?	Yes
	multiply-occurring?	No
	container for	`area`, `draw`, `exclGroup`, `field`, `subform`, `subformSet` Association at merge time.

Characteristics of layout objects (Continued)

Layout object	Used as …	
barcode	**machine-readable data**	
	layout strategy	Not a container
	break control?	No
	natural size?	Fixed or variable, depending on barcode type. When barcode type is variable, natural size depends upon symbol width.
	growable?	Depends upon barcode type; when yes, controlled by length of data
	splittable?	No
	multiply-occurring?	No
	container for	N/A
button	**clickable region**	
	layout strategy	Not a container
	break control?	No
	natural size?	Zero (Visually represented by container's caption and/or borders.)
	growable?	No
	splittable?	No
	multiply-occurring?	No
	container for	N/A
checkButton	**check box or radio button**	
	layout strategy	Not a container
	break control?	No
	natural size?	Size property (defaults to 10pt)+margins
	growable?	Yes
	splittable?	In margins
	multiply-occurring?	No
	container for	N/A

Characteristics of layout objects (Continued)

Layout object	Used as …	
contentArea	**physical region of a display surface**	
	layout strategy	Positioned, flowing
	break control?	No
	natural size?	Properties `w` and `h` required. Imposes splitting on contents.
	growable?	No
	splittable?	No
	multiply-occurring?	No
	container for	`area`, `draw`, `exclGroup`, `field`, `subform`, `subformSet` Association at layout time
draw containing image	**static displayable (boilerplate) image**	
	layout strategy	Positioned
	break control?	No
	natural size?	May be supplied by the properties `w` and `h`; otherwise, content + caption + margins. Does not cause splitting
	growable?	Yes. Optional limits supplied by properties, such as `minH`
	splittable?	In margins
	multiply-occurring?	No
	container for	Image
draw containing geometric figure	**static displayable (boilerplate) geometric figure**	
	layout strategy	Positioned
	break control?	No
	natural size?	Properties `w` and `h` required. Does not cause splitting
	growable?	No
	splittable?	In margins
	multiply-occurring?	No
	container for	`arc`, `line`, `rectangle`

Characteristics of layout objects (Continued)

Layout object	Used as …	
draw containing text	**static displayable (boilerplate) text**	
	layout strategy	
	break control?	
	natural size?	May be supplied by the properties `w` and `h`; otherwise, supplied by content + caption + margins. Does not cause splitting
	growable?	Yes. Optional limits supplied by properties such as `minH`.
	splittable?	In margins and between lines. Text within rotated containers cannot be split.
	multiply-occurring?	No
	container for	Text
embedded object	**non-text displayable entity embedded in text**	
	layout strategy	Not a container
	break control?	No
	natural size?	Implementation-defined
	growable?	No
	splittable?	No
	multiply-occurring?	No
	container for	N/A
exclGroup	**logical grouping of fields (one-of-many)**	
	layout strategy	Inherited
	break control?	No
	natural size?	content + caption = margins
	growable?	No
	splittable?	In margins or within contained object
	multiply-occurring?	No
	container for	`field`

Characteristics of layout objects (Continued)

Layout object	Used as …	
field containing button	**clickable widget**	
	layout strategy	Positioned
	break control?	No
	natural size?	Caption + margins
	growable?	Yes. Optional limits supplied by properties such as `maxH`.
	splittable?	In margins
	multiply-occurring?	No
	container for	`button`
field containing check-box	**clickable widget**	
	layout strategy	Positioned
	break control?	No
	natural size?	Content + caption + margins
	growable?	Yes. Optional limits supplied by properties such as `maxH`.
	splittable?	In margins
	multiply-occurring?	No
	container for	`checkButton`
field containing date, time, or date-time	**one item of variable data**	
	layout strategy	Flowing
	break control?	No
	natural size?	Content + caption + margins
	growable?	Yes. Optional limits supplied by properties such as `maxH`.
	splittable?	In margins and between lines
	multiply-occurring?	No
	container for	`text`

Characteristics of layout objects (Continued)

Layout object	Used as …	
field containing image	**variable image**	
	layout strategy	Positioned
	break control?	No
	natural size?	Content + caption + margins
	growable?	No
	splittable?	In margins
	multiply-occurring?	No
	container for	`image`
field containing number	**variable numeric data**	
	layout strategy	Flowing
	break control?	No
	natural size?	Content + caption + margins
	growable?	No
	splittable?	In margins
	multiply-occurring?	No
	container for	`text` (single-line)
field containing password	**variable character data**	
	layout strategy	Flowing
	break control?	No
	natural size?	Implementation-defined
	growable?	No
	splittable?	In margins
	multiply-occurring?	No
	container for	`text` (single-line)

Characteristics of layout objects (Continued)

Layout object	Used as …	
field containing radio button	**clickable widget**	
	layout strategy	Positioned
	break control?	No
	natural size?	Content + caption + margins
	growable?	Yes. Optional limits supplied by properties such as `maxH`.
	splittable?	In margins
	multiply-occurring?	No
	container for	`checkButton`
field containing signature	**clickable widget**	
	layout strategy	Implementation-defined
	break control?	No
	natural size?	Content + caption + margins
	growable?	No
	splittable?	In margins
	multiply-occurring?	No
	container for	Implementation-defined object
field containing text	**variable character data**	
	layout strategy	Flowing
	break control?	No
	natural size?	Content + caption + margins
	growable?	No
	splittable?	In margins and between lines. Text within rotated containers cannot be split.
	multiply-occurring?	No
	container for	`text`

Characteristics of layout objects (Continued)

Layout object	Used as …	
glyph	**printable symbol**	
	layout strategy	Not a container
	break control?	No
	natural size?	Determined by character code, type face, and type size
	growable?	No
	splittable?	No
	multiply-occurring?	No
	container for	N/A
image	**bitmapped image**	
	layout strategy	Not a container
	break control?	No
	natural size?	Determined by the image data
	growable?	Yes, unless `aspect` is `actual`. No limit.
	splittable?	No
	multiply-occurring?	No
	container for	N/A
line	**boilerplate geometric figure**	
	layout strategy	Not a container
	break control?	No
	natural size?	Inherited, except line thickness
	growable?	All, except thickness. No limit.
	splittable?	No
	multiply-occurring?	No
	container for	N/A

Characteristics of layout objects (Continued)

Layout object	Used as …	
pageArea	**display surface, such as one side of a sheet of paper**	
	layout strategy	Positioned (all content), flowing (all content except `contentArea`)
	break control?	No
	natural size?	Assumed to be infinite
	growable?	Not applicable; already infinite
	splittable?	No
	multiply-occurring?	Yes; controlled by `occur` property
	container for	`area`, `contentArea`, `draw`, `exclGroup`, `field`, `subform`, `subformSet`
pageSet	**collection of display surfaces**	
	layout strategy	Not applicable
	break control?	No
	natural size?	Assumed to be infinite
	growable?	Not applicable; already infinite
	splittable?	No
	multiply-occurring?	No
	container for	`pageArea`
rectangle	**boilerplate geometric figure**	
	layout strategy	Not a container
	break control?	No
	natural size?	Inherited, except line thickness
	growable?	All except line thickness; no limit
	splittable?	No
	multiply-occurring?	No
	container for	N/A

Characteristics of layout objects (Continued)

Layout object	Used as …	
subform	**logical grouping of fields and boilerplate**	
	layout strategy	Positioned, flowing
	break control?	Yes, except when used as a leader or trailer
	natural size?	May be supplied by the properties `w` and `h`; otherwise, determined by content. Does not cause splitting.
	growable?	yes. Optional limited supplied by properties such as `maxH`.
	splittable?	In margins and where consensus exists among contained objects
	multiply-occurring?	No. `occur` property not used in layout
	container for	`area`, `draw`, `exclGroup`, `field`, `subform`, `subformSet`
subformSet	**logical grouping of subforms**	
	layout strategy	Not applicable (transparent)
	break control?	
	natural size?	
	growable?	
	splittable?	
	multiply-occurring?	No. `occur` property not used in layout
	container for	`subform`, `subformSet`
text	**sequence of glyphs and/or embedded objects**	
	layout strategy	Flowing
	break control?	No
	natural size?	Determined by contents
	growable?	No
	splittable?	Between lines. Text within rotated containers cannot be split.
	multiply-occurring?	No
	layout container for	glyph or embedded object

C AXTE Line Positioning

Introduction

The process of text layout is indeed complex. Even English—which is perhaps the simplest language to lay out—is not without its challenges.

A number of Adobe products, including Acrobat, use the Adobe XFA Text Engine (AXTE) to effect text layout. Description of a complete layout process might occupy volumes. This document attempts to describe one key aspect of the AXTE layout process: line positioning. This document describes the behavior of AXTE when used for XFA-related processing in Acrobat 7 and Acrobat 8. Acrobat 6 used a different text engine which is not discussed here. Some other (non-XFA) subsystems of Acrobat use a different mode of AXTE which behaves slightly differently.

Note: This appendix is non-normative, that is, it is not a requirement that XFA processors operate exactly as described here. Rather this appendix is provided to help users of XFA understand the way in which a popular implementation handles certain aspects of text layout.

Line Positioning

An extreme over-simplification might be to describe the text layout process as follows:

- Divide the text amongst one or more lines
- Position each line vertically
- Within each line, position its text horizontally

This document describes the middle step. One aspect of the first and third steps that makes them difficult to document is that their processing is very language-dependent. For example, the rules for determining how much Arabic text fits in a line, and then positioning that text horizontally within the line are quite different from the rules for Thai text.

With one exception, the variables that control line positioning are used in the same way across languages. The exception is overall text orientation: horizontal (e.g., English) of vertical (e.g., Chinese). At this time of writing, vertically-oriented text is very much a work-in-progress in AXTE. Consequently, this document limits itself to horizontally-oriented text. In other words, describing line position is “simply” describing the vertical position of lines of horizontal text.

Scope

AXTE is only the text engine. It does not stand alone. Instead, there is an application that invokes AXTE services as needed to lay out and render text. AXTE deals with each text object in isolation. It is up to the invoking application to determine where text objects go on the page, screen or output/display medium.

For example, any XFA-based application deals with the concept of the XFA box model. This model describes object positioning and display embellishments such as margins and borders. In turn, box model objects are typically positioned in a subform—itself a box model object. Eventually, the entire hierarchy gets placed on a page. AXTE is oblivious to the box model constructs and higher-level positioning operations. Instead, it works in its own relative co-ordinate space that XFA applications place in the content region of the box model text object. In other words, AXTE deals only with the text itself and all box

model application is outside the scope of this document. Note that box model captions are separate text objects from AXTE's perspective.

This document makes no attempt to describe the behaviours of higher-level applications. For more information on those behaviours, please consult the applications' documentation.

AXTE makes extensive use of font metrics. These are measurements of various aspects of glyphs stored in font files. However, there are inconsistencies and ambiguities in the way such metrics are created and used. AXTE relies on Adobe's font access library, Adobe® CoolType, to access font information and resolve such issues. Explanation of CoolType algorithms is beyond the scope of this document.

Definitions

This document uses the term *text block* to describe any text object. In an XFA form each text object (including separate caption and field value) is a separate text block.

In the XFA world, higher-level XFA processing applies many XFA box model variables to get each AXTE text block positioned properly. These include borders and margins at the object, caption and widget level. As mentioned already, such processing is outside the scope of this document. Note that AXTE has its own margins, applied at the paragraph level.

The following diagram demonstrates some of the key concepts in vertical text positioning. Figure 1 shows three lines of text, middle-aligned in a block, with paragraph top and bottom margins. The black box indicates the extent of the text block. Dark grey lines denote the vertical boundaries between lines. Light grey lines indicate significant vertical offsets within lines.

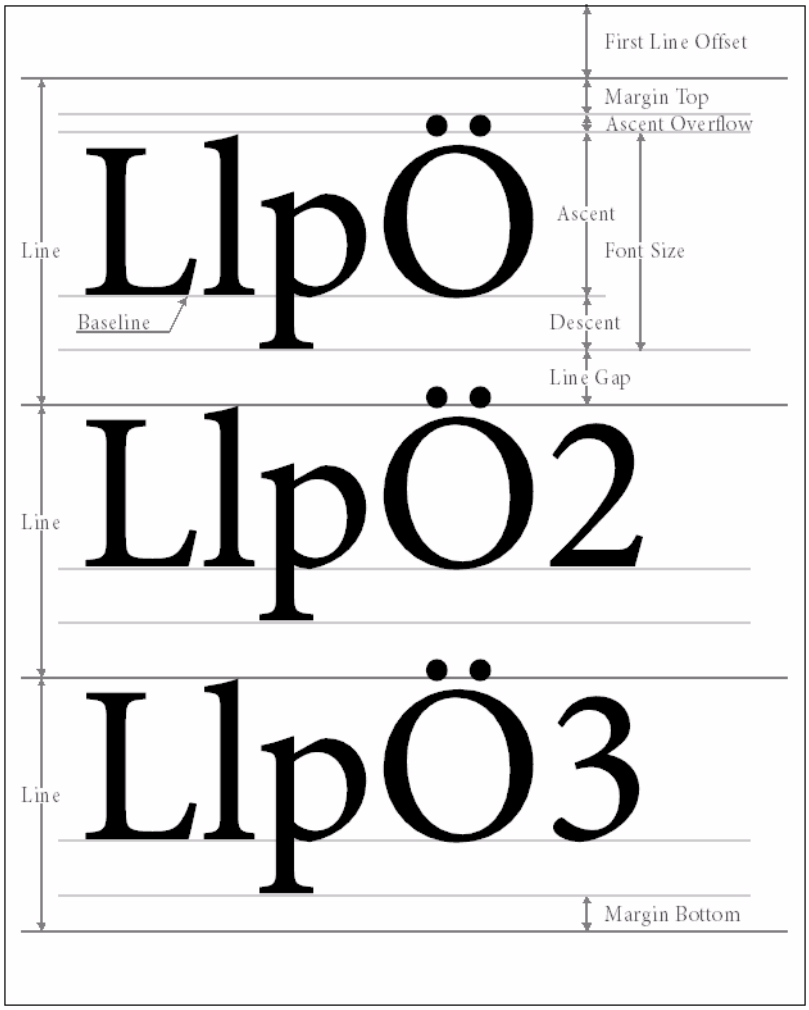


Figure 1 — Vertical layout definitions

Vertical text positioning is a function of a number of variables. Variables originate from font metrics, rich text content, or can be computed from other variables. The following table lists detailed definitions of the terms and variables used.

Term	Variable	Source	Definition
Ascender			The portion of any letter that extends above the baseline (see definition below). For example, the entire letter "L" is an ascender in figure 1.
Ascent	**A**	Font	The nominal height of the font above the baseline (see definition below). This is nominal only in that different characters have different ascents. For example, the characters "L" and "2" in figure 1 above do not extend quite as high as the lower-case "l". More important, it is possible for some characters to extend beyond the declared ascent, for example, accented upper-case letters such as the letter "Ö" in the example. Ascent is based on the Roman baseline of a font.

Term	Variable	Source	Definition
Ascent Overflow	**AO**	Computed	Additional vertical ascent applied to the first line to accommodate accents and other ascenders that extend beyond the declared ascent of the font. See the special handling of the letter "Ö" in the first line of figure 1. Ascent overflow is determined by comparing the bounding boxes of all glyphs in the line against the line's ascent.
Baseline	**B**	Computed	The line on which text characters are placed. Note that some characters extend below the baseline, for example, the letter "p" in figure 1. As a variable, the value of the baseline is the positive distance from the top of the line's extent to the baseline position.
Baseline Shift	**BS**	Text	Amount to shift the baseline of a span of text, either up or down.
Block Height	**BH**	Text	Height of the text block into which the lines are to be placed.
Derived Line Spacing	**DS**	Computed	Vertical line spacing computed for a line after all variables have been taken into account except for paragraph's top and bottom margins.
Descender			The portion of a letter that extends below the baseline, for example, the bottom part of the letter "p" in figure 1 above.
Descent	**D**	Font	The nominal height of the font below the baseline, for example, to accommodate the portion of the letter "p" in figure 1 that extends below the baseline. Descent is based on the Roman baseline of a font.
First Line Offset	**FO**	Computed	The distance from the top of the text block to the top of the first line.
Font Size	**FS**	Text	The declared size of the font. In an ideal world, this would be equal to the sum of the ascent and descent. For example, suppose a text object asserts a font of 12pt Myriad Pro. Then, the characters will normally be scaled such that the sum of the ascent and descent is 12pt. However, for many fonts, the sum of ascent and descent may be more than or less than the font height. Also known as *font height*.
Full Height	**FH**	Computed	Final height of a line, after all variables and computations have been taken into account. In figure 1, the full height is the distance between the dark grey lines that denote its vertical limits.
Line			The fundamental unit of vertical placement of horizontal text. In figure 1, each line is denoted by the text "line" at the left-hand side.

Term	Variable	Source	Definition
Line Gap	**LG**	Font	Space inserted between lines to ensure there is a visually pleasing separation between those lines. The line gap may also allow for descenders to clear ascenders or accented capitalized letters on the next line. Note that the line gap is not included in the font size. Note also that the line gap appears between lines. In other words, the number of gaps in a block of text is one less than the number of lines. In AXTE, the line gap is associated with the line that precedes it. Also sometimes referred to as *leading*, although leading can also mean the distance from one baseline to the next.
Line Height			The height of a line, including ascent, descent and line gap across all fonts in a line.
Line Spacing	**SP**	Text	Override of line height, supplied by the text. It is quite common for the line spacing override to be zero, indicating there is no override in effect. In other words, the line height would be determined from font metrics.
Margin Bottom	**MB**	Text	Margin applied after the last line in a paragraph. Also referred to as *bottom margin*. This is independent of margins applied by the XFA box model.
Margin Top	**MT**	Text	Margin applied before the first line in a paragraph. Also referred to as *top margin*. This is independent of margins applied by the XFA box model.
Text Height	**TH**	Computed	Text height, generally computed as the sum of maximum ascent and descent across a line.

Discussion

This section provides a verbal description of positioning issues.

Vertical Positioning Algorithm

Simply stated, the text layout algorithm can be thought of as following three steps.

- Flatten field hierarchy
- Break individual lines
- Measure and place lines

These are described in the following subsections.

Flatten Field Hierarchy

AXTE supports the concept of fields nested in text objects, and indeed fields within fields. This is necessary for form letters and other applications where flowed text consists of a combination of form-supplied content (draws) and user-supplied content (fields).

Though it has no direct effect on vertical positioning, flattening is nonetheless an important step in the process. This determines the order of the content to be laid out and therefore indirectly influences line positioning.

Break Individual Lines

The second layout step is to separate the flattened content into one or more lines, performing word-wrapping to ensure that no line exceeds the width available. AXTE uses the standard Unicode line breaking algorithm [UAX-14] during this stage.

As the rest of this document demonstrates, the process of vertical line positioning is all about positioning these lines.

Measure and Place Lines

This is the stage that is of most relevance to vertical line positioning.

Before individual lines can be placed, they must be measured as a group, in order that vertical alignment may be applied. Measuring a line means determining its Full Height. This step also determines the overall text block height for growable text blocks.

If the text block height is greater than the accumulated lines' height, the extra space appears at the bottom for top-aligned text, at the top for bottom-aligned text and split above and below for middle-aligned text.

Once AXTE has determined how much space to reserve above the lines, it has its `Y` offset—the first line offset—for the first line. This is the distance from the top of the text block to the top of the first line. The bottom of the first line is determined by measuring down from its top by its full height. The top of the second line is then positioned at the bottom of the first, and so on. Even though text is eventually placed on a line's baseline, many operations treat the line as a block, extending in `Y` from its top to its bottom.

Coordinate Systems

In any large graphic application, there are often multiple coordinate systems in use, with transformations between each. For this discussion, there are at least four:

- Application coordinates
- Text block coordinates
- Text line coordinates
- Font (glyph) coordinates

This discussion concentrates on the middle two. The rendering API used by AXTE supports multiple coordinate systems and stacking of coordinate transformations. It also supports rotation, so that AXTE need only worry about rendering glyphs horizontally and lines vertically.

Text Block Coordinates

The top-left corner of a text block is at `(0,0)` in its coordinate system. `X` coordinates increase to the right while `Y` coordinates increase downward. The top of each line is determined in this coordinate space.

Text Line Coordinates

Most of the calculations described below occur in text line coordinates. Each line has its own coordinate system, with `(0,0)` positioned at its top-left. Thus, the baseline's `Y` coordinate has a value greater than zero, the distance from the top of the line to the baseline.

Lines, Spans and Accumulation

Some attributes apply at the paragraph level. For example, it doesn't make sense to change horizontal alignment part way through a paragraph. The AXTE API ensures that paragraph-level attributes change only at paragraph breaks. Because an individual line can be part of only one paragraph, it can have only a single value for such an attribute, irrespective of how many spans are in the line. One example of this kind of attribute is the line spacing override.

While the line is the fundamental unit of interest in this discussion, a number of operations apply on a finer level of granularity—the *span*. Also known as a *run*, a span is a set of contiguous input characters[1] with consistent text attributes (e.g., font, baseline shift). In some of the algorithm descriptions below it is necessary to distinguish between the variable value for an individual span and that for the entire line. In such a case, the span variable is prefixed with a lower case `s`, while the line's variable is not. For example, a span's descent would be denoted by `sD`, while the line's descent is designated as `D`.

Because different spans may have different attributes, AXTE needs to reconcile them. For example, two spans may have different font sizes, which means different ascent, descent and line gap values. Or they may have the same size, but different font families which apportion the font height into ascent and descent differently. These variables directly affect line height calculation and therefore influence line positioning.

AXTE generally reconciles conflicting metrics by taking the larger of the two. This is referred to as *accumulation*. AXTE tends to perform accumulation on a variable-by-variable basis before calculating

1. The actual glyphs rendered for a line may not map 1:1 to the original characters and they may be re-ordered as part of the layout process. However this discussion focuses on the attributes themselves and not the actual glyphs rendered with those attributes. Indeed, AXTE performs all measurement for vertical text positioning on spans (of input characters, not glyphs).

derived variables. For example, suppose a line has two spans with the same font size, but that height is apportioned differently into ascent and descent between the two spans. If AXTE were to compute text height on each span first, both spans would yield the same height and AXTE would end up with a text height that was too small. Instead, AXTE accumulates ascent and descent separately and computes text height only when all spans have been accumulated. In this way, the computed text height is larger and has room for both the maximum ascent and maximum descent. When examining line content for vertical positioning, AXTE accumulates all of the following variables across the entire line, before performing calculations based on them:

- Ascent (A)
- Descent (**D**)
- Line Gap (**LG**)
- Spacing Override (**SP**), though this should not change value over the line

Note that Descent and Line Gap are accumulated independently. Some text processing systems accumulate the sum of Descent and Line Gap.

Special Lines

AXTE is aware of four special line types and applies special vertical positioning processing:

Type	Processing
First line in block	If there is a line spacing override and it is larger than the default line spacing, the extra space is ignored on the first line in a text block, for consistency with other text processing applications.
Last line in block	Any line gap value does not apply to the last line in a text block.
First line in paragraph	The top margin value applies only if the line is the first in its paragraph.
Last line in paragraph	The bottom margin value applies only if the line is the last in its paragraph.

Note that the first line in a text block is also the first line in its paragraph and that the last line in a text block is also the last line in its paragraph. In addition, the first line in a paragraph may be also the last line in its paragraph (one-line paragraph), and the first line in a block may be also the last line in the block (one-line text block).

Baseline Shift

Baseline shifts introduce a fair degree of complexity to the vertical positioning of text.

Relative Shifts

A baseline shift can be specified as a percentage of font size. This means that font metrics must be available to the code processing baseline shifts, and it must compute absolute shifts on-the-fly. Resolving relative shifts into absolute values is done a span-by-span basis, relative to the font size of the span containing the shift. In the algorithm descriptions below, any reference to a span's baseline shift (**sBS**) assumes that it has been resolved to an absolute value.

The Real Baseline

Earlier, this document discussed a line's baseline, as if there was only one and it was absolute for the line. Baseline shifts introduce multiple baselines in a single text line. In order to provide a reference point in this document and to remain consistent with the implementation, the existing definition of baseline stands. Baseline shifts simply cause temporary adjustments to the rendering baseline, but do not influence the line's baseline, other than through line height adjustments (see below) for ascent and descent extension.

Line Height Adjustments

The application of a baseline shift may or may not cause the shifted text to move outside the vertical space the line would otherwise occupy. An outright shift with no font size change will always require extra space. However, shifts are often accompanied by font size reductions (e.g., subscripts and superscripts). Shifted and resized text may fit without extending the space. A baseline shift's extensions may occur in either the ascent or descent area, and therefore may contribute to the accumulation of either ascent or descent.

Detailed Algorithms

The following two subsections describe the vertical placement algorithms with a minimum of prose. Note that all span-level algorithms are applied before line-level ones, which in turn are applied before any block-level ones.

Span Level

Basic Metric Adjustments

AXTE accommodates a combined ascent and descent larger than the font size. If the sum of the two is less than the font size, it pads the ascent, so that the requested font size is always consumed.

In addition, some fonts report unusual values for line gap, ranging from zero to 100% of the font size. Such values could lead to text lines that were too close together or too far apart. AXTE adopts the convention embraced by other Adobe applications that line gap is always determined to be 20% of font size.

```
if (sA + sD) < sFS then
   sA = sFS – sD
sLG = sFS * 0.2
```

General Metric Accumulation

This is where the accumulation of the (possibly modified) basic metrics occurs. AXTE simply accumulates separate ascent, descent and line gap for later calculation.

```
accumulate sA in A
accumulate sD in D
accumulate sLG in LG
```

Line Spacing

Internally AXTE has the convention that a line spacing override value of zero means that there is no override in effect (line spacing to be determined from font metrics). Convention has it that increasing line spacing does not push down the first line of a text block—it affects only the remaining lines.

```
if first line in block then
   if sSP > sFS then
      sSP = 0
accumulate sSP in SP
```

Baseline Shift

Note that most spans have no shift. This discussion applies only to those that do. The span's absolute shift value, **sBS**, is negative for up-shifts and positive for down-shifts.

AXTE treats up-shifts and down-shifts consistently. An up-shift will alter the accumulated ascent only if more ascent space is required after taking both the shift and font size into account. A down-shift will alter the accumulated descent only if more descent space is required after taking both the shift and font size into account.

Note that a baseline shift is often accompanied by a change to a smaller font size. Therefore, span variables **sA**, **sD**, **sLG** and **sFS** are often smaller than those accumulated elsewhere in the line.

Consequently, steps that accumulate data based on these variables often don't change the underlying accumulated values.

```
if sBS < 0 then
   accumulate sA + |sBS| in A
else
   accumulate sD + |sBS| in D
```

Line Level

Text Height

This is simply a matter of computing the overall text height based on the independently accumulated ascent and descent values.

```
TH = A + D
```

Derived Spacing

This is essentially the calculation of the line height required for all the text in the line, before considerations that apply to special lines. If any line spacing override is in effect, it is used. Otherwise, AXTE uses the sum of accumulated ascent, descent and line gap.

```
if SP > 0 then
   DS = SP
else
   DS = TH + LG
```

Margin Adjustments

Appropriate margins are dropped if this line doesn't meet special line criteria. If there is no bottom margin or it is too small, it is increased.

```
if not first line in paragraph then
   MT = 0
if not last line in paragraph then
   MB = 0
```

Full Height

The full height is the total amount of vertical space occupied by the line. AXTE removes the line gap on the last line in a block so that bottom-aligned text doesn't appear shifted up.

```
FH = MT + DS + MB
if last line in block then
   FH = FH - LG
```

Adjustment for First Line Accents

Some fonts have glyphs whose ascent overflowed the font's declared ascent. This happens most often to accented capital letters. Without accounting for this situation, it could lead to the complete truncation of accents from a displayed line of text is some applications. So AXTE adjusts for ascent overflow on the first line only, provided there is no line spacing override in effect. On subsequent lines, it is expected that any overflow will spread into the line gap of the previous line.

```
if first line in block and AO > 0 and SP == 0 then
   A = A + AO
   TH = TH + AO
   FH = FH + AO
```

Baseline

Finally the position of the baseline—relative to the top of the line's space—can be determined. All non-shifted glyphs are drawn on this baseline (**B**). For any span that has a baseline shift, its glyphs are position on a baseline computed as **B+sBS**.

When no line spacing override was in effect, version 6 effectively included the line gap above every line (including the first) by computing a baseline above the bottom margin by the line's descent amount.

If there is a spacing override, AXTE uses that value to position the descent, provided the spacing override exceeds the text height. In other words, increasing the line spacing override over the text height makes the text move down, but decreasing it below the text height does not make it move up. This is intentional behaviour to obtain the most sensible line positioning in both cases.

AXTE has already accounted for the spurious line gap (after the last line) in the calculation of full height above.

```
if (SP == 0) or (SP - LG < TH) then
   B = MT + TH - D
else
   B = MT + SP - LG - D
```

Block Level

First Line Offset

Once the full height of all lines in the block has been determined, the first line offset can be set. Under certain circumstances, AXTE may store lines whose total height is greater than the block height. In such a case, the block is treated as being top-aligned. Depending on the application, lines spilling out the bottom may or may not be rendered.

```
if sum(FH) > BH then
   FO = 0
else
   if top alignment then
      FO = 0
   else if middle alignment then
      FO = (BH - sum(FH)) / 2
   else
      FO = BH - sum(FH)
```

D History of Changes in This Specification

This chapter describes the changes made to the XFA syntax and XFA processing rules since version 2.0.

New Object Models

The following Data Object Models (DOMs) were added after version 2.0.

Data Object Models

Connection Set DOM added, version 2.1

The Connection Set DOM holds information concerning web services. This information is required in order to use web services. The Connection Set DOM is serialized as XML in a new section of the XDP.

Connection Data DOM added, version 2.1.

The Connection Data DOM is a temporary buffer used to hold data that is about to be sent to a host or has just been received from a host. While there the data can be inspected and modified by scripts. The Connection Data DOM is only serialized in free-standing messages to the host, never as part of an XDP.

Data Description DOM added, version 2.1.

The Data Description DOM holds a description of the structure (schema) for the data. This information is optional. Furthermore, scripts and supplied data are not constrained by the data description. However if the data description is supplied, and the supplied data conforms to it, then the XFA processor will ensure that any changes it makes to the Data DOM during data binding conform to the supplied schema.This DOM is serialized as XML in a new section of the XDP.

Layout DOM added, version 2.1

The Layout DOM holds the mapping of logical form features - blocks of text, images, and so on - to pages and regions of pages. This DOM is never serialized to XDP.

Special Object Models

Special Object Models, version 2.1

Several special objects are mentioned in this document. These objects play a role in Scripting Object Model expressions, which may in turn appear as the values of XFA template attributes. These special objects include $event, $host, $layout, $log, and $vars. These objects and their properties and methods are described in *Adobe XML Form Object Model Reference* [FOM].

New XFA Template Features

Element Descriptions

Availability, version 2.5

Template elements are now formally divided into subsets according to the type of form in which they can appear. Each element description now indicates the element's subset membership. See "How to Read an Element Specification" on page 482.

Container Properties

Form fragments, version 2.4

Almost any object in the template can now be used as a prototype for other objects. In addition prototypes may be located in external templates accessed via URI. Hence any accessible template may supply prototypes for other templates. See "Defining Prototypes" on page 195.

Bar code encryption, version 2.4

Barcode data can be encrypted using a public key before rendering as a barcode. This makes it possible to transfer confidential data via facsimile. See "Special Processing of Barcode Data" on page 361.

Barcode character encoding, version 2.4

Barcode data can be translated into a specified character encoding, as opposed to always being encoded as UTF-8 serialized Unicode. In particular QRCode barcodes can use the customary Shift-JIS character set. See "Special Processing of Barcode Data" on page 361.

Exclusion group element's capability expanded, version 2.1

The exclusion group container element (`exclGroup`) now has most of the same characteristics as the field container element. For example, exclusion group may now contain the properties event, connect, validate, and calculate.

Previously, an exclusion group was simply a logical grouping of fields.

Hide/reveal containers depending on relevance, version 2.1

All containers now include an attribute that can result in the container being excluded from the form if it is not relevant to the current form view. For example, this capability can automatically cause the form to reformat itself, depending on whether the form is being viewed online or printed.

Growable containers, version 2.1

Certain types of containers may be defined as growable along one or both axes. The extent to which the container can grow or shrink is defined by a size range. Whether the container grows within the given range is driven by the size of the data provided.

Paragraph formatting, version 2.1

Paragraph formatting instructions may be associated with any of the elements that contain displayable text. Such instructions include horizontal and verticle alignment, line spacing, and margins.

Barcode formatting, version 2.1

A container may now specify that its data should be displayed as a barcode, rather than as text.

Image aspect, version 2.1

The aspect of an image may be retained as the size of its host container changes. That is, if a container grows or shrinks, the image it contains may be grown or shrunk or may remain unchanged.

Noninteractive fields, version 2.1

Fields may be designated to be non-interactive. Such fields participate in data binding, but after that they are treated as `draw` elements.

Automation and Web-Related Interactions

Secure submit, version 2.5

Submitted data can be signed with one or more signatures. In addition signatures can be verified and/or cleared. The submitted data can also be encrypted.

Index change event, version 2.5

An event is provided to trigger scripts when the index of an instance is modified. For example, suppose a dyamic form has three instances of a subform and the middle instance is deleted. This causes an `indexChange` event to be generated for the last subform because its index changes from 2 to 1. For more information see "Instance Manager Events" on page 342.

URL-encoded option for submit, version 2.4

Data submitted to a host can be URL-encoded in a consistent way. For more information see the "Template Specification" on page 482.

Choice-list enter and exit events pair up, version 2.4

The event model for choice lists is modified to ensure enter and exit events always pair up. See "Field Events" on page 338.

Manifests as scripting variables, version 2.4

Manifests (sets of objects for signing or other processing) can be specified via scripting variables. For more information see the "Template Specification" on page 482.

Support for Web Services, version 2.1

XFA template now supports Web Services that implement 'doc-literal' SOAP operations over HTTP. In such operations, the Web Service's WSDL defines SOAP binding operations with 'document' style, and SOAP messages with 'literal' encoding. This capability has added properties to the elements `xfa:datasets`, `dd:dataDescription`, and `xfa:event`.

Submission of form parts to a target URI, version 2.1

An event may include a submission property. When such an event is activated, the submission property causes the form (all or part) to be submitted to a target URI. The submission property indicate the parts, the packaging, the encoding, and the destination URI.

Subforms may include calculations, verison 2.1

Subforms may include calculations, which simplies declaring a calculation influenced by multiple child-containers within a subform.

Calculations may specify override conditions, version 2.1

Calculate elements may include override the `override` attribute, which specifies whether the calculation may be executed or not. If allowed to execute, the `override` attribute specifies whether the user is allowed to override the calculated value.

Scripts specify whether they should be executed on the client, server or both, version 2.1

Scripts may now specify whether they should be executed on an XFA processing application that thinks it is a client, one that thinks it is a server, or both.

Previously, there was no such distinction.

Event for populating drop-down choice list widgets, version 2.2

Fields containing drop-down choice lists may have a new event, which is triggered when the user clicks on the down-arrow symbol. This new event is intended to house scripts that add choices to the choice list. The new event is especially useful when the choice list is infrequently used and its choices take a while to load.

Document variables, version 2.1

Subforms may be defined with variables that specify various content types, such as text, external data, and images. The variables can be used by scripts to establish the value of a container as the value of the variable. See “Document Variables” on page 325.

Validation checks against validation-specific picture clauses, version 2.1

Validation checks a newly supplied or calculated value against the picture clause contained in the validate element. That is, the picture clause used for validation checks is independent of the picture clause used for formatting.

Previously, such validation was checked against the picture clause contained in the format element. The requirement to perform such validation was indicated by the value of the validate element’s `formatTest` attribute.

When reading in legacy files, if the validate element’s `formatTest` attribute is set to `"warning"` or `"error"`, copy the <format>/<picture> to the <validate> tag.

Event source included as an event attribute, version 2.1

The event element now includes an attribute used to indicated the source of the trigger that activates the event. In the following template sample, the script fires in the context of field "X" when the button associated with "Y" is clicked.

```
<field name="X">
   <event ref="Y" activity="click">
      <script> … </script>
   </event>
</field>

<field name="Y"/>
```

The default value for `ref` is "$", indicating the event context is the current node.

Other

See "FormCalc support for East Asian scripts in locale designators" on page 1082.

Naming Conventions

Support for tags and attribute names containing "." and "_", version 2.2

In earlier versions of XFA, XFA names were not allowed to include the characters "." (period) or "_" (underscore). This simplified parsing of SOM expressions in which "." is a special character. However this meant that XFA could not support data files containing element tags or attribute names containing either "." or "_".

In XFA 2.2 the definition of an XFA name is relaxed so that almost any valid tag or attribute as defined in [XML] can be used in data. The sole exception is that XFA still does not support the colon (":") character in tags and attributes; although the colon character is allowed by XML, it is rarely used because it conflicts with [XML Namespace].

Note that XFA 2.1 also requires XFA-SOM expression parsers to support an escape notation using "\" (backslash) to escape the special meaning of "." inside SOM expressions. In addition scripts written in FormCalc ("FormCalc Specification" on page 891) require special handling of references to data nodes having names containing the "." character. See "Using SOM Expressions in FormCalc" on page 92 for more information.

Support for xsi:nil, version 2.1

The original XML specification [XML1.0] does not provide a way to distinguish between elements and attributes that contain a zero-length string ("") and those which have no value at all assigned to them (null values). This distinction is important in many applications. Later the [XML-Schema] standard introduced a notation to represent values which are truly null. XFA 2.1 supports the representation of null values in data using the notation defined by the XML-Schema, the `xsl:nil` attribute. However it is still implementation-defined whether a particular XFA processor supports null values as distinct from zero-length strings or not. See "Data Values Representing Null Data" on page 125 for more information about null-value handling.

Data Mapping (Data Loading)

New grouping transform, version 2.1

A new grouping transform has been added in XFA 2.1. This transformation allows for contiguous related data items to be grouped automatically into a hierarchical structure, as though they had been wrapped inside an enclosing element. This added structure makes it possible to take maximum advantage of XFA's intelligent merging and layout. See "Extended Mapping Rules" on page 423 for more information.

Support for references in image data, version 2.1

Image data may now include hypertext references to content. Depending on the trusted nature of the source of such hypertext references, the image data may be loaded into the XFA Data DOM.

Specifying data attributes to use for naming nodes in the XFA Data DOM, version 2.1

The config element can specify the data attribute from which the data loader obtains the node name, rather than using the data element name. This config attribute is useful in situations where the element names are not meaningful. See "The nameAttr Element" on page 440.

Data Unloading

Use of data description when writing out XML, version 2.1

When loading data from XML XFA does not need to know the schema of the input XML data document. However when unloading to XML, XFA 2.1 allows for the use of a schema to control the form of the output XML data document. When no schema is supplied the behavior is unchanged from previous versions. The schema, if present, is contained in an XFA Data Description as described in "Unloading Node Type Information" on page 136. See "Data Values Representing Null Data" on page 125 for a detailed description of the way in which the data description is applied when unloading.

Data Binding

Complex binding, version 2.4

Data binding can update properties other than `value` via an explicit reference to the Data DOM or to a web service. Almost any property can be updated this way at bind time, for example a caption can be copied from a data value. See "Bind to Properties (Step 5)" on page 183.

Conditional binding, version 2.4

Data binding to the `value` property of a container via an explicit data reference can be conditional upon the value of the data. This is accomplished via an extension to the grammar for SOM expressions. See "Selecting a Subset of Sibling Nodes" on page 88.

Dynamic forms, version 2.1

These are forms that change in structure in accordance with the data. See "Static Forms Versus Dynamic Forms" on page 286 for detailed information.

Repeating subforms, version 2.1

In XFA 2.0, when it was desired to repeat the same subform multiple times, it was necessary to re-declare the subform once for each instance. In XFA 2.1 a repeating subform can be declared just once along with properties that control how many times it repeats. See "Forms with Repeated Fields or Subforms" on page 202 for detailed information.

Explicit data references, version 2.1

The automatic data-binding logic can now be overridden on a per-field basis. The field can be forced to bind to an arbitrary node in the Data DOM. See "Explicit Data References" on page 177 for detailed information.

Subform sets, version 2.1

Subforms can now be grouped under a controlling object called a subform set. The subform set declares the logical relationship of the subforms, for example, that they are mutually exclusive. The logical relationships supported correspond closely to the relationships supported by common schema languages. See "Subform Set" on page 302 for detailed information.

Record processing, version 2.1

Data can now optionally be processed a record at a time. In this mode only a logical record of data is loaded into memory at any one time. Processing in record mode limits consumption of memory and CPU cycles. It also limits the scope of data binding to the current record in most cases. See "Creating, Updating, and Unloading a Basic XFA Data DOM" on page 108.

Global fields, version 2.1

A field can now be declared global. A field declared this way can bind to certain data outside the current record. Globals were not required in XFA 2.0 because it did not support record processing. See "The Bind Element" on page 158 for detailed information.

Note that in every case XFA 2.1 data binding is backwards-compatible with XFA 2.0 templates and data. The default behavior is always the same as the XFA 2.0 behavior.

Data description element, version 2.1

XML data documents consumed by XFA processing applications may include data description elements. Such elements provide information used during data binding to ensure the XFA Form DOM for the data corresponds to a desired schema.

Default data binding to include attribute data, version 2.1

XML data consumed by an XFA processing application frequently uses attributes to supply data. To accommodate such use, the config grammar default value for the `attribute` property has been changed to "`preserve`".

Previously the default value for the `attribute` property was "`ignore`".

Subform scope option, version 2.1

It is now possible to make a subform transparent to the data binding process without making it nameless. Hence it can be referenced in XFA-SOM expressions even though it is a non-entity for data binding.

Layout

XFA Foreground (XFAF), version 2.5

A new way of declaring boilerplate is defined. In traditional XFA forms the boilerplate is defined as `draw` elements and laid out at run time. This applies whether the form is static or dynamic. In the new XFAF forms the boilerplate is laid out in advance as a PDF appearance stream split up into pages. This reduces run time overhead at the cost of fixing the appearance (but not the order) of each page. It also allows finer control over the appearance of boilerplate text within a page because PDF supports features such as kerning that XFA does not.

It is still necessary to use `draw` for boilerplate when you need to add or omit sections within a page dynamically. For this reason traditional boilerplate is still fully supported by XFA.

An XFAF form is restricted to a subset of template elements. The type of form in which each element is used is indicated in the element description within the template syntax specification. See "How to Read an Element Specification" on page 482.

For more information about XFAF see "The Relationship between XFA and PDF" on page 19 and "Static versus Dynamic Forms" on page 27

Change to initial page selection, version 2.5

The algorithm for selecting the initial `pageArea` has been changed. The old algorithm surprised form authors by causing a blank page to be emitted at the start of processing under some circumstances. The new algorithm conforms better to intuition. See "Determining the start point" on page 255.

Explicit control of printer pagination, version 2.5

Templates can control pagination on printers with greater flexibility and precision. The same template can print optimally on both simplex (single-sided) and duplex (double-sided) printers. See "Pagination Strategies" on page 254.

Support for right-to-left text flow, version 2.4

Locales in which text flows right to left are now supported. Any block of text may contain any mixture of left to right and right to left flow. In addition draws can specify a locale (and therefore a default flow direction for text). See "Flowing Text Within a Container" on page 52.

Conditional breaking, version 2.4

Layout markup can supply a scripting expression to decide at layout time whether or not a particular break should be taken. See "Break Conditions" on page 232.

Nesting tables, version 2.4

Tables may nest to any depth. See "Tables" on page 281.

Automatically breaking layout, version 2.1

Subforms may now specify their appearance in the event their content forces them to grow across a page boundary. This appearance may specify the inclusion of a leader subform at the top of a subform break or the inclusion of a trailer subform at the bottom of a subform break.

Containers may describe the appearance of borders around containers that break across pages.

Dynamic layout, version 2.1

XFA supports dynamic forms. Such forms automatically adjust depending on data being entered. In a dynamic form, the arrangement of the form is determined by the arrangement of the data. For example, if the data contains enough entries to fill a particular subform 7 times, then the Form DOM incorporates 7 copies of the subform. Depending on the template, subforms may be omitted entirely or rearranged, or one subform out of a set selected by the data. Dynamic forms are more difficult to design than static forms but they do not have to be redesigned as often when the data changes. In addition dynamic forms can provide an enhanced visual presentation to the user because unused portions of the form are omitted rather than simply left blank.

In contrast, XFA 2.0 supports only static forms. In a static form, the template is laid out exactly as the form is to be presented. When the template is merged with data, some fields are filled in. Any fields left unfilled are present in the form but empty (or optionally given default data). These types of forms are uncomplicated and easy to design.

Flowing layout strategy, version 2.1

In flowing layout, containers may be positioned from top-right to bottom-left. Additionally, flowing layout also supports tables, as described in "Flowing layout support for tables and table-rows, version 2.1" (below).

Previously, text layout could flow only from left to right.

Flowing layout support for tables and table-rows, version 2.1

Flowing layout now supports tables, by allowing subforms to specify a layout strategy for tables. If such a subform contains child subforms that specify a layout strategy for rows, the implied columns and rows are kept in sync with one another.

Rich Text

Embedded objects, version 2.1

Rich text may now include references to text or images. Such references may be expressed using the Scripting Object Model (for internal references) or using URI's (for external references). Embedded objects are resolved dynamically.

Subscript and superscript support, version 2.1

Rich text may now employ the superscript and subscript markup defined by[XHTML].

Accessibility and User Interface

New Widget Types

Signature widget, version 2.1

There is a new type of widget defined for use in affixing a digital signature to a form.

Image entry widget, version 2.1

There is a new type of widget defined for use in picking an image for entry into a form.

Rich text option for text widget, version 2.1

The text widget can now optionally accept rich text from the user and allow editing of existing rich text.

Widget Appearance

Control over scrolling, version 2.5

New properties `hScrollPolicy` and `vScrollPolicy` have been added to various widgets to give explicit control over horizontal and vertical scrolling when being used interactively. For more information see "Date/Time Editing Widget" on page 411, "Numeric Edit" on page 412, and "Text Edit Widget" on page 414.

Checkmark shapes, version 2.5

For improved compatability with Acroforms, checkmarks in checkbuttons can take any of a specified set of shapes. For more information see the description of the `checkButton` element within the "Template Specification" on page 482.

Button highlight, version 2.5

For improved compatability with Acroforms, buttons may specify a highlight mode. When the highlight mode is `push` the button can have two captions, one that is displayed when it is depressed and another

that is displayed when it is released. For more information see the description of the `button` element within the "Template Specification" on page 482.

Comb support in numeric and date edit widgets, version 2.5

Previously combs were only supported in text edit widgets. For more information see the description of the `comb` element within the "Template Specification" on page 482.

Explicit control over number of cells in combs, version 2.5

A new attribute `numberOfCells` is provided to explicitly specify the number of character cells in the comb. This provides an override for those situations in which the field's `maxChars` property does not necessarily equal the number of cells (for example because the text contains combining characters). For more information see the description of the `comb` element within the "Template Specification" on page 482.

Widget margins, version 2.1

Any of the user interface widgets may specify a margin, which insets from the edge of the containing field. The margin prevents the widget from being eclipsed by the field's border, especially in the situation where the border is wide and is even- or right-handed.

Widget borders, version 2.1

Any of the user interface widgets may specify a border.

Choice List Widgets

Multiple selections, version 2.1

Choice list widgets can now allow the user to select multiple options from the list of options.

Immediate commitment of selections, version 2.1

The default behavior of choice lists widgets is now to commit the selected data, as soon as the user makes the selection. When data is committed, it is propagated to the XFA Data DOM. Previously, choice list selections were submitted only when the choice-list field was exited.

Note: Templates based on XFA versions prior to 2.1 may need to be modified to retain their original behavior. This is accomplished by adding to the `choiceList` element the attribute definition `commitOn="exit"`.

Caption Appearance

Clarification of caption reserve, version 2.5

Previous versions of this specification did not make clear the meaning of a caption reserve set to zero. This is now clarified. For more information see the description of the `caption` element under the "Template Specification" on page 482.

Captions can differ between views, version 2.4

The `caption` element now accepts a `relevant` attribute. This makes it possible for a caption to differ in different views of the form, for example when printed versus when filled in interactively. For more information see the "Template Specification" on page 482.

Caption margins, version 2.1

Captions may specify a margin, which insets the caption text from the edge of the containing field. The margin prevents the widget from being eclipsed by the field's border, especially in the situation where the border is wide and is even- or right-handed. Also, the margin allows more refined placement of captions.

Form Navigation

Accelerator key allows keyboard sequence to bring fields into focus, version 2.2

Fields and Exclusion Groups may now have an accelerator key property (`accessKey`). When the character assigned to a field's accessibility key is selected in combination with the system's modifier key (on Windows, Alt), the form's focus shifts to the indicated field.

Aids for Vision-Impaired Users

Role of a container may be defined, especially for table headings and rows, version 2.1

The assist element now includes a role attribute, which can be used to declare the role any container plays. XFA processing applications can use this property to identify the role of subforms, exclusion groups, fields, and draws. One possible use of this new attribute is to assign it values from the HTML conventions. Such values would declare the role of the parent container, such as `role="TH"` (table headings) and `role="TR"` (table rows). Such role declarations may be used by speech-enabled XFA processing applications to provide information about a particular container.

Speech order prioritized, version 2.1

The speech order for a field may be re-prioritized.

Localization and Picture Clauses

Data Localization

Locale Set, an XML grammar for representing localization information, version 2.1

The Locale Set contains locale-specific data used in localization and canonicalization. Such data includes picture clauses for representing dates, times, numbers, and currency. It also contains the localized names of items that appear in dates, times and currencies, such as the names of months and the names of the days of the week. It also contains mapping rules that allow picture clauses to be converted into a localized string that can be used in UI captions and prompts.

Data Picture Transform (template and config), version 2.1

It is now possible to specify a bind picture clause for converting incoming localized data into canonical format and outgoing data back into localized format. Such a picture clause would target a known data format. For example, a picture clause transform could strip the currency symbol from certain incoming data (leaving a pure number suitable for manipulation by scripts) and insert the currency symbol into the corresponding outgoing data. See "Localization and Canonicalization" on page 138 for more information.

The config grammar may now specify a picture clause to use in localizing data. Such a picture would override the template-provided data picture clause described in the previous paragraph. A config-provided data picture clause may be useful in providing a picture clause specific for the data, especially when the format may not be known at the time the XFA template is created. It is also overrides

template-provided picture clauses, in resolving canonicalization/localization when multiple XFA fields provide conflicting bind picture clauses for the same data. See "Transforms" on page 425.

Default output format reflects locale, version 2.2

The date/time/number fields that omit a format picture clause are displayed in a locale-sensitive manner. Previously, such un-pictured field values were displayed in canonical format.

This change does not alter the behavior for data-binding or other data transfer specification. That is, a bind element that does not enclose a picture element still consumes and produces data in a canonical format only. And placing a picture element within a bind will result in the data being formatted and parsed based on the locale. See "Rule 4, Output Formatting When Output Picture Clause Omitted" on page 149.

Picture Clause Expressions

Uppercase versus lowercase picture symbols, version 2.5

The specification was unclear about the differences between the **Z** and **z** picture symbols and between the Ｚ (U+FF3A) and ｚ (U+FF5A) picture symbols. A new section has been added to clarify the differences. See "Uppercase Picture Symbols versus Lowercase Picture Symbols" on page 1019.

Generic pre-defined picture clauses, version 2.4

Picture clause syntax is extended to allow templates to invoke a predefined picture format (such as `date.short`) which adapts to the particular locale automatically. These picture clauses are defined in the Locale Set and can be altered by the form creator. See "Predefined Picture Clauses" on page 996.

New picture symbol "8", version 2.4

Anew numeric picture symbol is defined for pictures that always retain the supplied precision of the data. See "Numeric Pictures" on page 1017.

Picture clause symbols for zero and null values, version 2.2

Picture clauses now include picture clause symbols that can format/parse null data and zero data. These new symbols are identified with category designators that identify them as applying to null data or zero data. See "Null-Category Picture Clauses" on page 1024 and "Zero-Category Picture Clauses" on page 1025.

Retention of precision in decimal numbers parsed, version 2.2

When a decimal number is input parsed as the value of field that lacks a UI picture element, the XFA processing application retains the specified number of digits to the right of the decimal point. The field properties may specify an upper limit of fractional digits to retain or may specify the number of fractional digits as being data-driven.

This change and the change "Number picture clause symbol for fractional digits, version 2.2" (below) allow XFA processing applications to retain and format significant fractional digits entered for decimal numbers.

Number picture clause symbol for fractional digits, version 2.2

A new number picture clause symbol has been added that specifies the number of significant digits to the right of the decimal radix (decimal point). If the fractional digit that corresponds with the symbol is present in the data, it is included in the number. If it is not, it is represented as a space.

This change and the change "Retention of precision in decimal numbers parsed, version 2.2" on page 1081 allow XFA processing applications to retain and format significant fractional digits entered for decimal numbers.

Compound picture clauses, version 2.1

Picture clauses may now include multiple parts, with each part calling out a specific locale or a specific category of data. Such a compound picture clause is similar to a C-language union data type.

Symbols used for whitespace characters, version 2.1

Picture clauses may include the asterisk (*) or plus sign (+). For input parsing, these symbols indicate, respectively, zero or more whitespace characters or one or more whitespace characters. For output parsing, these symbols indicate a single space.

Support for Asian-Language Representations, version 2.1

The date, time, and number picture clauses now support for Asian-language representations. In particular, they allow specification of the formats described below.

Imperial era years

Date picture clauses can specify that years be expressed relative to imperial era years. Additionally, date picture clauses can specify a particular style of imperial era. *Imperial eras* assign a starting point for counting years. They are equivalent to the Gregorian calendar's implied assignment of AD to a year.

- *Full-width numeric values.* All time, date, and number data formats can be specified as full-width characters. In text that combines Latin numbers and Asian-language ideographs, full-width numbers provide a consistent size and a squared shape that is more consistent with Asian-language ideographs.

Ideographs

All time and date information may be expressed using ideographs. The particular script (ideograph system) used may be the default for the prevailing locale or may be explicitly declared in the prevailing locale.

Tens rule

All ideographic numeric values may be expressed using the Arabic numeral system or the tens rule numeral system. A *numeral system* is a method for using numerals to represent numbers.

Support for East Asian scripts in locale designators

The locale designator used in the picture clauses can now specify a script. A script is an entire set of characters or ideographs, such as Korean Hangul or Latin.

FormCalc support for East Asian scripts in locale designators

FormCalc now supports East Asian scripts in locale designators used in picture clauses. Picture clauses are used in the

A script is an entire set of characters or ideographs, such as Korean Hangul or Latin. This change is consistent with corresponding changes in "Localization and Picture Clauses" and "New XFA Template Features".

Note: XFA template values for the locale attribute is limited to language and country code designators, such as fr_FR (French specific for France) and zh (Chinese).

Scripting Object Model (SOM)

Value tests in SOM expressions, version 2.4

The SOM expression syntax is extended to allow selection of a subset of nodes by applying a Boolean expression to each. See "Selecting a Subset of Sibling Nodes" on page 88.

Referencing objects by their class names, version 2.1

Objects may now be referenced by their class names, as described in "Reference by Class" on page 83.

Previously, an object could be referenced only by its name. This limitation presented a problem for un-named objects.

Document variables used as named script objects, version 2.2

Scripts may now reference properties and methods declared in a named script object. Such a named script object is declared in a variables element. See "SOM Expressions That Reference Variables Properties" on page 107.

FormCalc

New functions to access locale, version 2.1

FormCalc now provides several functions that support locale. In particular, it allows conversion of canonical localizable data, such as currency, date, and time, from/to localized presentations of such data. It also allows access to the prevailing locale for any form data, where prevailing data is obtained from the template, from the host system, or from the default locale for XFA processing applications.

FormCalc support for East Asian scripts in locale designators

This feature is described above in "FormCalc support for East Asian scripts in locale designators" on page 1082.

Security and Control

MDP+ document signatures, version 2.5

The grammar for XML digitial signatures is extended to support MDP+ document signatures. For more information see "Signed Forms and Signed Submissions" on page 464.

XML digital signatures, version 2.2

XFA now supports XML digital signatures, using the mechanism described by W3C for an XML Digital Signature [XMLDSIG-CORE]. This support includes new event properties that describe actions for creating, clearing, and validating signatures. It also includes metadata additions to the Signature object described [XMLDSIG-CORE]. These additions are used in signature validation. See "Signed Forms and Signed Submissions" on page 464.

Uniquely identifying templates, version 2.2

XDP and PDF specify properties for uniquely identifying a template and for time-stamping the template whenever it is changed. In addition, XFA designing applications and XFA processing applications are required to retain/propogate the the template identifier and to update the time stamp when applicable.

Modified XFA Template Features

See "Validation checks against validation-specific picture clauses, version 2.1" on page 1073.

See "Immediate commitment of selections, version 2.1" on page 1079.

See "Default data binding to include attribute data, version 2.1" on page 1076.

Deprecated XFA Template Features

Template Syntax

Refactored break element, version 2.4

The syntax of the `break` element was confusing, largely because it was overloaded with too many functions. It has been replaced with a set of simpler single-purpose elements. The `break` element is still legal in XFA 2.4 but is deprecated and slated for removal in a future version.

Deprecated hAlign and vAlign attributes on container elements, version 2.4

The `hAlign` and `vAlign` attributes on container elements (`subform`, `field`, `draw` and `exclGroup`) are redundant. The same-named attributes on the `para` element, which is a property of each of the container elements, should be used instead. The redundant attributes on the container elements are deprecated in XFA 2.4 and slated for removal in a future version.

Note: These attributes were never implemented in Acrobat.

Deprecated stateless attribute on the script element, version 2.4

The `stateless` attribute on the `script` element is not practical. It intrudes too deeply into the architecture of the scripting languages. This attribute is deprecated in XFA 2.4 and slated for removal in a future version.

Note: This attribute was never implemented in Acrobat.

Deprecated transient attribute on the exclGroup element, version 2.4

The `transient` attribute on the `exclGroup` element has no real effect. An `exclGroup` is always effectively transient because it is data-driven. This attribute is deprecated in XFA 2.4 and slated for removal in a future version.

E Schemas

About the Schemas

The XFA schemas are written in the language RELAX: Next Generation (RNG) which is described in [RELAX-NG]. XFA schemas are written in this language rather than the more usual XML-Schema 1.0 [XMLSchema] because XFA allows free ordering in situations where XML-Schema cannot support it. This free ordering allows differently-named children of an element to appear in any order without changing the meaning. In RNG this is signified by the `interleave` directive. For example, in RNG one can say:

```
<element name="a">
   <interleave>
      <element name="x"/>
      <element name="y"/>
      <element name="z"/>
   </interleave>
</ref>
```

This declares that the element `a` has three child elements `x`, `y`, and `z` which may appear in any order. There are six possible permutations for the child elements: xyz, xzy, yxz, yzx, zxy, and zyx . This could be declared in XML-Schema but it would have to be declared as six alternative sequences of child elements. Unfortunately the number of permutations goes up as the factorial of the number of distinct child elements and XFA has elements with dozens of distinct children. Hence it is not practical to represent XFA in XML-Schema while preserving the free ordering.

XFA Profiles

Starting with XFA 2.5 there is a facility to specify that a form uses a subset of the full XFA capability. This is indicated in the `template` element by a non-default value for the `baseProfile` attribute. Currently the only specified value is `interactiveForms`, which corresponds to the XFAF subset.

When the template element specifies a profile, the contents of the template must be restricted to the set of elements and attributes allowed by that profile. For the restrictions of the XFAF profile see "Grammar Excluded from XFAF" on page 235.

Caution: The XFA Schema attached to this Appendix is defined for the full XFA grammar. It will not detect the presence of elements or attributes that are not appropriate for the profile specified by the form.

Extracting the Schemas

The schemas are included as file attachments in the PDF form of this document. If you are viewing a hard copy, obtain the PDF file from the Adobe website at http://adobe.com/go/xfa_specifications.

Each schema file is denoted by an attachment symbol in the rightmost column of the table below. To extract a schema using Acrobat, right-click on the attachment symbol and select "Save embedded file to disc..." from the context menu. Be sure to extract all of the files into the same directory. Also, for each file use the supplied filename. This is necessary because the XDP schema (`xdp.rng`) incorporates all of the other schemas in the set by reference using their filenames.

File	Description	Attachment
`config.rng`	configuration	
`connectionset.rng`	connections to web services	
`data.rng`	user data portion of the dataSets packet	
`dataDescription.rng`	data description portion of the dataSets packet	
`localeset.rng`	locale definitions	
`pdf.rng`	accompanying PDF	
`sourceset.rng`	connections to databases	
`stylesheet.rng`	XSLT stylesheet(s) for custom transformation of data and/or the template	
`template.rng`	template	
`xdp.rng`	container for everything else	
`xfdf.rng`	annotations	
`xmldsig-core-schema.rng`	signing control	

Note: These schemas validate XFA 2.5 grammars only. They ignore packets generated for other versions of XFA which use different namespaces. Some namespaces stay the same from one version of XFA to the next, while others change. The template grammar in particular always changes from one version of the specification to the next, hence the template packet namespace always changes.

Using the Schemas to Validate an XFA Document

The validator usually used for RNG is Jing. Jing is written in java. You can download Jing from the location given in the bibliography as [JING]. To validate an XDP file, start Jing on the command line with the command

```
java -jar jing.jar xdp.rng myfile.xdp
```

Note: The XDP schema allows it to contain arbitray custom packets as child elements. The validation accepts any and all such packets. However if a packet matches one of the types declared in any of the schemas then the content of that packet is validated against that schema.

Using the Schemas to Generate an XFA Document

RNG is not a generative grammar. The `interleave` operator does not determine an order for the elements it governs. By contrast XML-Schema is deliberately more restrictive than RNG so that it can be used to generate new documents. There is a program for creating a schema in XML-Schema format automatically from the RNG schema, however in order to accomplish this the program "freezes" the document order of the RNG schema into the order of child elements in the XML-Schema schema. Hence the XML-Schema schema is a subset of the RNG schema, with all content included but most orderings excluded.

Caution: This XML-Schema schema should only be used for generating XFA documents, never for validating them.

The program for generating an XML-Schema schema from an RNG schema is called Trang and it is available for download at the location given in the bibliography as [TRANG]. For information about running Trang see the manual that accompanies the program.

The XML-Schema grammar is defined in [XMLSchema].

Bibliography

The references in this section are grouped in the categories: "General References", "Fonts and Character Encoding References", and "Barcode References".

General References

[ADO]

ADO API Reference. Microsoft.
At press time the most current version, 2.8, is available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/ado270/htm/mdmscadoapireference.asp

[Adobe-Patent-Notice]

Adobe Patent Clarification Notice.
At press time there are two relevant Patent Clarification Notices, one for PDF and one for XDP, and they are available at http://adobe.com/go/developer_legal_notices.

[CSS2]

Cascading Style Sheets, level 2 (CSS2) Specification. B. Bos, H. W. Lie, C. Lilley, I. Jacobs, 12 May 1998.
Available at http://www.w3.org/TR/REC-CSS2

[ElectronicSecurity]

A primer on electronic document security, Adobe Systems Incorporated.
Available at http://adobe.com/go/security_primer.

[ECMAScript]

ECMAScript Language Specification. ECMA International, 1999.
http://www.ecma-international.org/publications/files/ecma-st/Ecma-262.pdf.

[ECMAScript357]

ECMAScript for XML (E4X) Specification. ECMA International, 2005.
http://www.ecma-international.org/publications/files/ecma-st/Ecma-357.pdf.

[EXCLUSIVE-XML-CANONICALIZATION]

Exclusive XML Canonicalization. World Wide Web Consortium, 2002.
Available at http://www.w3.org/TR/xml-exc-c14n/.

[FOM]

LiveCycle Designer ES Scripting Reference.
Available at http://adobe.com/go/learn_lc_XMLFormObject.

[HTTP]

Hypertext Transfer Protocol -- HTTP/1.1. World Wide Web Consortium.
Available at http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.5

[IANA]

Character Sets. Internet Assigned Number Authority.
Available at http://www.iana.org/assignments/character-sets.

[IEEE754]

IEEE 754: Standard for Binary Floating-Point Arithmetic. This standard may be purchase through http://grouper.ieee.org/groups/754.

[ISO-639-1]

Codes for the representation of names of languages -- Part 1: Alpha-2 code. International Organization for Standardization.
Available for purchase at http://www.iso.org/

[ISO-3166-1]

Country Codes. International Organization for Standardization.
Available for purchase at http://www.iso.org/

[ISO-4217]

Codes for the representation of currencies and funds. International Organization for Standardization.
Available for purchase at http://www.iso.org/

[ISO-8601]

Data elements and interchange formats — Information interchange — Representation of dates and times. International Organization for Standardization (ISO), 2000.
Available for purchase at http://www.iso.org/.

[JING]

JING. A RELAX NG validator in Java. Thai Open Source Software Center Limited, 2001, 2002, 2003.
Available at http://www.thaiopensource.com/relaxng/jing.html.

[MIMETYPES]

MIME Media Types. Internet Assigned Number Authority.
Available at http://www.iana.org/assignments/media-types/.

[PDF]

PDF Reference, sixth edition, *Adobe Portable Document Format, Version 1.7*. Adobe Systems Incorporated, 2006.
Available at http://adobe.com/go/acrobatsdk_pdf_reference.

[RELAX-NG]

RELAX NG Specification, Committee Specification: 3 December 2001. The Organization for the Advancement of Structured Information Standards (OASIS), 2001.
Available at http://www.oasis-open.org/committees/relax-ng/spec-20011203.html.

[RFC1738]

Uniform Resource Locators (URL). T. Berners-Lee, L. Masinter, M. McCahill, 1994.
Available at http://www.ietf.org/rfc/rfc1738.txt.

[RFC1951]

DEFLATE Compressed Data Format Specification version 1.3. P. Deutsch, 1996.
Available at http://www.ietf.org/rfc/rfc1951.txt

[RFC1766]

Tags for the Identification of Languages. Internet Engineering Task Force, March 1995 http://www.ietf.org/rfc/rfc1766.txt.

[RFC2045]

Multipurpose Internet Mail Extensions (MIME) Part One, Format of Internet Message Bodies. N. Freed, N. Borenstein, November 1996.
Available at http://www.ietf.org/rfc/rfc2045.txt.

[RFC2046]

Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. N. Freed, N. Borenstein, 1996.
Available at http://www.ietf.org/rfc/rfc2046.txt.

[RFC2119]

RFC2119: Key words for use in RFCs to Indicate Requirement Levels. S. Bradner, March 1997. Available at http://www.ietf.org/rfc/rfc2119.txt.

[RFC2376]

XML Media Types. E. Whitehead, M. Murata, July 1998. Available at http://www.ietf.org/rfc/rfc2376.txt.

[RFC2396]

RFC2396: Uniform Resource Identifiers (URI): Generic Syntax. T. Berners-Lee et al., August 1998.
Available at http://www.ietf.org/rfc/rfc2396.txt.

[RFC2397]

RFC2397: The "data" URL scheme. L. Masinter, August 1998.
Available at http://www.ietf.org/rfc/rfc2397.txt.

[RFC3161]

RFC3161: Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP). C. Adams et al., August 2001.
Available at http://www.ietf.org/rfc/rfc3161.txt.

[RFC-3066bis]

RFC 3066bis is the identifier for a working draft of *Tags for Identifying Languages*.
Available at http://xml.coverpages.org/draft-phillips-langtags-03.txt

[RFC3280]

Public Key Infrastructure (X.509) (PKIX) Certificate and Certificate Revocation List (CRL) Profile. IETF RFC 3280, April 2002 ,http://www.ietf.org/rfc/rfc3280.txt

[SOAP1.1]

Simple Object Access Protocol (SOAP) 1.1. World Wide Web Consortium, 2000.
Available at http://www.w3.org/TR/2000/NOTE-SOAP-20000508/. Note that although this is merely a note, not a recommendation, it has been adopted as the framework for [[WSDL1.1].

[SRGB]

IEC 61966-2-1: Multimedia systems and equipment - Colour measurement and management - Part 2-1: Colour management - Default RGB colour space sRGB. October 1999. Amended by *IEC*

61966-2-1-am1: Amendment. March 2003.
Available at http://webstore.iec.ch/.

[TRANG]

Trang. Multi-format schema converter based on RELAX NG. Thai Open Source Software Center Limited, 2002, 2003.
Available at http://thaiopensource.com/relaxng/trang.html.

[URI]

RFC2396: Uniform Resource Identifiers (URI): Generic Syntax. T. Berners-Lee, R. Fielding, L. Masinter, August 1998. This document updates RFC1738 and RFC1808.
Available at http://www.ietf.org/rfc/rfc2396.txt.

[WSDL1.1]

Web Services Description Language (WSDL) 1.1. World Wide Web Consortium, 2001.
Available at http://www.w3.org/TR/2001/NOTE-wsdl-20010315

[XFDF]

XML Forms Data Format Specification Version 2.0. Adobe Systems Incorporated, 2003.
Available at http://www.adobe.com/go/learn_lc_XFA.

[XHTML]

XHTML 1.0: The Extensible HyperText Markup Language - A Reformulation of HTML 4 in XML 1.0. World Wide Web Consortium, 2000.

Available at http://www.w3.org/TR/2000/REC-xhtml1-20000126/.

[XML1.0]

Extensible Markup Language (XML) 1.0 Specification. World Wide Web Consortium, 1998.
Available at http://www.w3.org/TR/REC-xml

[XML1.1]

Extensible Markup Language (XML) 1.1 (Second Edition). World Wide Web Consortium, 2006.
Available at http://www.w3.org/TR/xml11.

[XMLBASE]

XML Base. World Wide Web Consortium, June 2001.
Available at http://www.w3.org/TR/xmlbase/.

[XMLDOM2]

Document Object Model (DOM) Level 2 Specification: Version 1.0. World Wide Web Consortium, 1999.
Available at http://www.w3.org/TR/DOM-Level-2/.

[XMLDSIG-CORE]

XML-Signature Syntax and Processing. World Wide Web Consortium, 2001.
Available at http://www.w3.org/TR/xmldsig-core/.

[XMLEncryption]

XML Encryption Syntax and Processing. World Wide Web Consortium, 2002.
Available at http://www.w3.org/TR/xmlenc-core/.

[XMLNAMES]

Namespaces in XML. T. Bray, D. Hollander, A. Layman, 14 January 1999.
XML namespaces provide a simple method for qualifying names used in XML documents by associating them with namespaces identified by URI.
Available at http://www.w3.org/TR/REC-xml-names

[XMLSchema]

XML Schema Part 1: Structures and *XML Schema Part 2: Datatypes*. World Wide Web Consortium, 2001.
Available at http://www.w3.org/TR/xmlschema-1/ and http://www.w3.org/TR/xmlschema-2/ , respectively.

[XMPMeta]

XMP Specification. Adobe Systems Incorporated, January 2004.
Available at http://adobe.com/go/XMP_spec.

[XPATH]

XML Path Language (XPath) Version 1.0. World Wide Web Consortium, 1999.
Available at http://www.w3.org/TR/xpath.

[XSL-FO]

Extensible Stylesheet Language (XSL) Version 1.1. World Wide Web Consortium, October 2006.
Available at http://www.w3.org/TR/xsl11/.

[XSLT]

XSL Transformations (XSLT) Version 1.0. World Wide Web Consortium, November 1999.
Available at http://www.w3.org/TR/xslt.

Fonts and Character Encoding References

[Adobe-Fonts]

Font Technical Notes. Adobe Systems Incorporated, 2003.
Available at http://adobe.com/go/font_tech_notes.

[Code-Page-950]

See: http://www.microsoft.com/globaldev/reference/dbcs/950.htm.

[GB2312]

Chinese Character Encoding Charset for Information Exchange — Base Set (National Standard GB2312-80). State Bureau of Standardization of the People's Republic of China (PRC), 1980. A paper copy of this standard, with English titles but Chinese text, can be purchased at http://webstore.ansi.org/ansidocstore/chinese_standards.asp?.

[ISO-8859-1]

Information technology — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1. International Organization for Standardization (IS0), 1998.
Available for purchase at http://www.iso.org/.

[ISO-8859-2]

Information technology — 8-bit single-byte coded graphic character sets — Part 2: Latin alphabet No. 2. International Organization for Standardization (IS0), 1998.
Available for purchase at http://www.iso.org/.

[ISO-8859-7]

Information technology — 8-bit single-byte coded graphic character sets — Part 7: Latin/Greek alphabet. International Organization for Standardization (IS0), 1998.
Available for purchase at http://www.iso.org/.

[ISO-10646]

Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane. International Organization for Standardization (ISO), 2000.
Available for purchase at http://www.iso.org/.

[KSC5601]

Code for Information Interchange (Hangul and Hanja). Korea Industrial Standards Association, 1987, Ref. No. KS C 5861-1992.

[Shift-JIS]

JIS X 0208: 7-bit and 8-bit double byte coded KANJI sets for information interchange. Japanese Industrial Standards Committee, 1997.
Available for purchase at http://www.webstore.jsa.or.jp/webstore/Top/indexEn.jsp?lang=en.

[UAX-9]

Unicode Standard Annex #9. The Unicode Consortium, 2005.
Available at http://www.unicode.org/unicode/reports/tr9/.

[UAX-14]

Unicode Standard Annex #14. The Unicode Consortium, 2002.
Available at http://www.unicode.org/reports/tr14/tr14-12.html.

[Unicode-2.1]

The Unicode Standard, Version 2.1.9. The Unicode Consortium, 2002. This version of Unicode is referenced only for its definition of Basic Multilingual Plane (BMP) character set. All other Unicode references relate to the Unicode Standard, Version 3.2 (below).
Available at http://www.unicode.org/standard/versions/components-pre4.html#Unicode_2_1_9.

[Unicode-3.2]

The Unicode Standard, Version 3.2. The Unicode Consortium, 2002.
Available at http://www.unicode.org/standard/versions/components-pre4.htmll#Unicode_3_2_0.

Barcode References

[APO-Barcode]

Customer Barcoding Technical Specifications. Australia Post, 1998. Available at http://www.auspost.com.au/GAC_File_Metafile/0,,2043_techspec,00.pdf.

[Codabar]

ANSI/AIM BC3-1995, USS - Codabar. American National Standards Institute, Inc. and the Association for Automatic Identification and Data Capture Technologies, 1993.
Available for purchase at http://www.aimglobal.org/aimstore/.

[Code2Of5Interleaved]

ANSI/AIM BC2-1995, USS - Interleaved 2 of 5. American National Standards Institute, Inc. and the Association for Automatic Identification and Data Capture Technologies, 1993.
Available for purchase at http://www.aimglobal.org/aimstore/.

[Code39]

ANSI/AIM BC1-1995, Uniform Symbology Specification - Code39. American National Standards Institute, Inc. and the Association for Automatic Identification and Data Capture Technologies, 1993.
Available for purchase at http://www.aimglobal.org/aimstore/.

[Code49]

ANSI/AIM BC6-1995, Uniform Symbology Specification - Code49. American National Standards Institute, Inc. and the Association for Automatic Identification and Data Capture Technologies, 1993.
Available for purchase at http://www.aimglobal.org/aimstore/.

[Code93]

ANSI/AIM BC5-1995, Uniform Symbology Specification - Code93. American National Standards Institute, Inc. and the Association for Automatic Identification and Data Capture Technologies, 1993.
Available for purchase at http://www.aimglobal.org/aimstore/.

[Code128-1995]

ANSI/AIM BC4-1995, Uniform Symbology Specification - Code128. American National Standards Institute, Inc. and the Association for Automatic Identification and Data Capture Technologies, 1995.
Available for purchase at http://www.aimglobal.org/aimstore/.

[Code128-1999]

ANSI/AIM BC4-1999, Uniform Symbology Specification - Code128. American National Standards Institute, Inc. and the Association for Automatic Identification and Data Capture Technologies, 1999.
Available for purchase at http://www.aimglobal.org/aimstore/.

[ISO-15420]

Information technology — Automatic identification and data capture techniques — Barcode symbology specification EAN/UPC (ISO/IEC 15420:2000). International Organization for Standards (ISO) and International Electrotechnical Commission (IEC), 2000.
Available for purchase at http://www.iso.org/.

[LOGMARS]

MIL-STD-1189B. United States (of America) Department of Defence, 1984.
Note: this standard has been withdrawn. The Department of Defence has adopted [Code39] in its place. However according to http://www.afmc.wpafb.af.mil/HQ-AFMC/LG/LSO/LOA/stands.htm

on 5 December 2003, "Users are cautioned to evaluate this document for their particular application before citing it as a replacement document."

[Maxicode]

ANSI/AIM BC10-ISS, Maxicode. American National Standards Institute, Inc. and the Association for Automatic Identification and Data Capture Technologies, 1996.
Available for purchase at http://www.aimglobal.org/aimstore/.

[PDF417]

Uniform Symbology Specification PDF417. The Association for Automatic Identification and Data Capture Technologies, 1998.
Available for purchase at http://www.aimglobal.org/aimstore/.

[QRCode]

International Symbology Specification - QR Code (AIM ITS/97/001). Automatic Identification Manufacturers (AIM) International, 1997.
Available for purchase at http://www.aimglobal.org/aimstore/.

[RM4SCC]

Mailsort User Guide, sections "Mailsort 700" and "Mailsort 120". Royal Mail (United Kingdom), 2006.
Available at http://www.mailsorttechnical.com/downloads_mailsort_user_guide.cfm.

[Telepen]

Uniform Symbology Specification - Telepen. Automatic Identification Manufacturers (AIM) International (year of publication not available).
Available for purchase at http://www.aimglobal.org/aimstore/.

[USPS-C100]

Facing Identification Mark (FIM). United States (of America) Postal Service, 2006. Replaces the C100 standard.
Available at http://pe.usps.com/text/dmm300/708.htm#wp1316612.

[USPS-C840]

Barcoding Standards for Letters and Flats. United States (of America) Postal Service, 2006. Replaces the C840 standard.
Available at http://pe.usps.com/text/dmm300/708.htm#wp1352817.

Glossary

A

ambient locale

The locale specified for the operating system or environment in which an application operates. In the rare circumstance where the application is operating on a system or within an environment where a locale is not present, the ambient locale defaults to English United States (en-US), which is the default locale.

application processor

See "XFA application processor"

annotation

Additional content added to a PDF document. Such content includes comments.

B

boilerplate

See fixed content (boilerplate).

C

canonical format

A locale-agnostic, standardized way to represent date, time, numbers, and currencies. Canonical time and date formats are subsets of the ISO-8601 standard [ISO-8601].

canonicalization

The part of input parsing that considers locale when converting locale-specific dates, times, numbers, and currencies into canonical format. See also localization.

character data

All text within an XML document that is not markup constitutes character data. See the description of character data within section "2.4 Character Data and Markup" of the XML specification [XML].

connection set

The connections used to initiate or conduct web services. Such a set defines connections for web services (WSDL), sample data (XML), and schema files (XSD).

container element

A type of XFA template element that specifies either of the following: (a) content elements and the form-related aspects of navigating to, displaying and processing those elements; or (b) other container elements and the form-related aspects of navigating to, displaying and processing those container elements.

Container elements include `pageArea`, `area`, `subform`, `field`, and `exclGroup`.

container object or node

An object that can be populated with content objects or with other subordinate container objects. Container objects are created during data binding and layout processing.

Container objects include `area`, `subform`, `field`, and `exclGroup`.

content element

A type of XFA template element that houses datatyped text or graphic elements (lines and images). Such text or graphic elements may be defined as default data or un-changeable data in the content element.

content object or node

An object that houses datatyped pcdata (text) or graphic elements (lines and images). New content objects are created during data binding and layout processing.

The data may be pre-defined in the content element or may be provided by the form user or by some other source, such as Web Service interactions.

content type element

A content type element defines the type of the data in the parent content element. It may also include default data or un-changeable data, which is used when the form is displayed. Examples of such datatype elements are `date`, `decimal`, and `image`.

current record

It is possible to read and process the data document one record at a time rather than loading it all into memory at once. When operating in this mode the record which is currently being processed is called the current record. Records immediately preceding and following the current record may also be loaded, depending upon the setting of a configuration option. When record processing is not being done the current record should be understood as including the entire data document. See "Creating, Updating, and Unloading a Basic XFA Data DOM" on page 108 for more information about record processing.

D

data binding (merging)

The process of merging the Data DOM with the Template DOM.

Data DOM (XFA Data DOM)

The Data DOM is the tree-structured representation of user data. During the data binding process, the Data DOM supplies the content for fields in the merged form. The term *Data DOM* differs from the XML Data DOM.

data group

A data group is an object in the XFA Data DOM that corresponds to an element holding other elements (as opposed to character data) in an XML data document. Within the XFA Data DOM interior nodes are usually data groups. A data group may have other data groups and/or data values descended from it.

data independence

An XFA feature that allows a form designer to change a template without requiring that corresponding changes be made to the structure of the data. This feature also retains the structure of the XML Data DOM and the values of objects in the XML Data DOM that are not used used in the XFA Form DOM.

data loader

A program or subsystem responsible for loading data from an XML data document into the XFA Data DOM.

data unloader

A program or subsystem responsible for unloading data from the XFA Data DOM into a new XML data document.

data value

A data value is an object in the XFA Data DOM that corresponds to an element holding character data (and possibly other elements) in an XML data document. Within the XFA Data DOM leaf nodes are usually data values. A data value may have other data values descended from it but it must not have any data group descended from it.

date/time format style

A locale-independent style of representing date or time. Supported date and time styles include short, medium, long, and full. One date style is designated the default, as is one time style. The date/time format styles may be defined in the localeSet element, described in"The localeSet Element" on page 150.

default locale

See "ambient locale".

default mapping rule

A rule that governs, by default, how an XML data document is mapped to an XFA Data DOM.

document

An serialized XML tree. A document is typically stored as a file.

document object model (DOM)

A Data Object Model is an in-memory representation of data as a tree of objects. An object which belongs to a DOM may be referred to as a "node" in order to emphasize its role as a tree

member. For example, a "form node" is an object belonging to the Form DOM.

document order

The order in which a Form DOM is traversed. Document order starts at the root of the content subtree of the Form DOM and traverses the content subtree depth-first and left-to-right (oldest-to-newest).

document range

The section(s) of the XML data document that is/are loaded into the XFA Data DOM.

dynamic form

An XFA form that automatically adds containers and rearranges its layout depending on the data being entered into it. In a dynamic form, the arrangement of the form is determined by the arrangement of the data supplied to the Form DOM. A dynamic form is enabled by an XFA template that specifies subforms that may be replicated, depending on the data supplied to the Form DOM. See also static form.

E

element content

XML elements that contain only XML child elements, optionally separated with white space, constitute "element content". See the description within section "3.2.1 Element Content" of the XML specification [XML].

element type

The first token within an XML start or end tag identifies the "element type". The element type is a string containing a qualified name. A qualified name consists of an optional namespace prefix and colon, followed by a mandatory local name. See the description within section "3. Qualified Names" of *Namespaces in XML* [XMLNAMES].

empty element

An XML element that does not enclose any content.

empty merge

An "empty merge" occurs when a template is merged with an empty data document (or no data document at all). The rules for an "empty merge" are slightly different than the rules for a non-empty merge. Different attributes of The Occur Element are used and default data, if defined by the template, is inserted.

extended mapping rule

A rule that is not in effect by default but is available as an override or extension to the default-mapping rules.

F

fixed content (boilerplate)

Data associated with a draw. Such data is defined by the template designer and does not vary throughout the life of the form, with the following exception: Dynamic forms may omit fixed content that appears in unused containers. See also variable content.

form creator

The person and/or software that creates a form template, possibly along with other information such as a data description.

form data

The variable data within a form. This is data that the user can enter or modify and/or that is loaded from an external source such as a database at the time the form is presented to the user.

Form DOM

The Form DOM is the tree-structured representation of the filled-in form. The Form DOM is created and populated by the data binding process. The Form DOM is not, however, ready to display; there is another step required to perform a physical layout, then another to render the form to a display or printer. The Form DOM embodies structural relationships, not physical representations.

form template

A collection of related subforms and optional business logic, constraints, and processing rules.

G

global

When record processing is in effect the current record and optionally other records adjacent to it are loaded into the Data DOM. In addition, "global" data is loaded into the Data DOM and kept in memory while records cycle in and out. Hence, global data is available for use by scripts throughout the document. For example, when an organization carries on business under several different names, the appropriate name is often made global so that it can be displayed on every page of a multi-page document without having to incorporate it in the data more than once. Data is made global by placing it in elements that are at the same level as or higher in the hierarchy than the records. In addition a field may be marked global, which means it is a candidate for matching to global data (but it can still match to non-global data).

grammar

Set of rules that specify the use of words in a particular namespace. This specification describes several XFA-related grammars, including template, config , and localeSet.

I

input parsing

The process of transforming a formatted input value into a raw elemental value, under the direction of a picture clause. This term is the inverse of output formatting.

instance manager

An object placed into the Form DOM by the data binding process for the use of scripts. One instance manager is placed in the Form DOM for each dynamic subform in the Form DOM. A script can use the instance manager to determine how many instances of the subform have been copied into the Form DOM and it can delete instances or insert more instances.

L

Layout DOM

layout node

A layout node is any object in the Layout DOM.

layout processor

The layout processor is an entity tasked with laying out displayable content on the display surface(s), typically on behalf of an application.

locale

A standard term used to identify a particular nation (language and/or country). A locale defines (but is not limited to) the format of dates, times, numeric and currency punctuation that are culturally relevant to a specific country.

locale-dependent format

A style of representing dates, times, numbers, and currencies that is unique to the locale.

locale set

Locale-specific data used in localization and canonicalization. Such data includes picture clauses for representing dates, times, numbers, and currency. It also contains the localized names of items that appear in dates, times and currencies, such as the names of months and the names of the days of the week. It also contains mapping rules that allow picture clauses to be converted into a localized string that can be used in UI captions and prompts.

localization

The part of output formatting that involves converting canonical dates, times, numbers, and currencies into formats and characters commonly used in a particular locale. See also canonicalization.

localized date or time format

Locale-specific character strings intended for use in UI captions and prompts. Such strings are defined in the `localeSet` element (locale set). Some FormCalc functions return localized date or time formats.

Localized date or time formats cannot be used as date or time picture clauses (picture clause (pattern)). In some locales, localized date or time formats are identical to their picture clause counterparts; however, this similarity is not consistent across locales. For example, en_US might use MM/DD/YY as both a picture clause and a localized date format. In contrast, the fr_CA locale might use a localized date format of aa/nn/jj as the counterpart to the MM/DD/YY picture clause.

M

merge

The data-binding process is sometimes called the "merge" process because it can be thought of as merging content from the Data DOM with structure from the Template DOM to create a single document, the Form DOM. However it should be noted that it is possible to perform a data binding operation without a Data DOM, in which case the Form DOM gets its content from default data in the Template DOM.

metadata

In this specification, "metadata" refers to data expressed via XML attributes.

mixed content

XML elements that contain character data interspersed with XML child elements constitute "mixed content". See the description of mixed content within section "3.2.2 Mixed Content" of the XML specification [XML].

N

node

An object linked to other objects within a hierarchical structure. In XFA the hierarchical structure is always one of a predefined set of DOMs (Document Object Models). All objects specified in XFA are nodes.

nominal extent

The nominal extent of an object is a rectangle aligned with the X and Y axes that covers the region on the page reserved for the object. The nominal extent does not *necessarily* include the whole physical extent of a visible object or, in the case of a container, its contents.

normalizing the Data DOM

A process optionally performed by XFA processing applications to move data nodes around to reconcile data-binding contradictions. An example of such a contradiction is a data node being bound to a form node, even though the nearest merge-able ancestor of the data node and the nearest merge-able ancestor of the form node are not bound to each other. "Re-Normalization (Step 4)" on page 178.

O

output formatting

The process of transforming a raw value into a formatted value, under the direction of a picture clause. This term is the inverse of input parsing.

P

page area

A description of a rendering surface, such as one side of a printed page or a screen-display.

page set

An XFA element used to represent a set of display surfaces, such as a stack of sheets of paper. For example, a page set for a duplex document would nominally consist of two page areas: a front page area and a back page area. In another example, a page set for an invoice might consist of three page areas: a first page bearing a letter, followed by alternating statement-front and statement-back pages.

PDF subassembly

An unit of content added to the top level of a PDF document tree. Examples of PDF subassemblies are annots, data, and signature.

picture clause (pattern)

A sequence of symbols (characters) that specify the rules for formatting and parsing textual data, such as dates, times, numbers and text. Each

symbol is a place-holder that typically represents one or more characters that occur in the data.

plain text

Text that does not contain any markup signifying formatting, hence, text that is not rich text.

prevailing locale

The locale obtained after resolving locales supplied from the following sources (in priority order) the picture clause, the field or subform locale property, or the ambient locale. The prevailing locale affects input parsing and output formatting. It also affects the results of FormCalc functions.

R

rich text

Text containing markup signifying formatting such as bold and underline.

S

script

A set of instructions for processing data or initiating events such as data exchange with a server. In XFA scripts are not necessary for common electronic form functionality, but scripts may be used to provide greater control or meet unusual needs.

SOM (XFA Script Object Model)

A model for referencing values, properties and methods within a particular Document Object Model (DOM).

SOM resolver

A software component that interprets a SOM expression, yielding the set of all nodes that match the expression. The resulting set may be empty.

source set

ADO database queries, used to describe data binding to ADO data sources

static form

An XFA form that has a set number of subforms. Unlike dynamic forms, static forms cannot add subforms to accommodate additional data.

T

template

See form template.

Template DOM

The Template DOM is the tree-structured representation of the template for the form. During the data binding process it supplies the prototype objects and relationships between objects which are copied into the Form DOM. Hence the Template DOM dictates the structure of the resulting merged form.

U

UI

A type of element that describes how data should be presented to a form user. UI elements are defined in the XFA template grammar.

V

variable content

Data associated with an exclusion group or field. Such data, which varies throughout the life of the form, may be provided by any of the following: the template as a default value, the person filling out the form, an external source, a calculation, and other sources. See also fixed content (boilerplate).

W

Web service

An automated service provided by an external (non-XFA) processor and accessed using the Simple Object Access Protocol (SOAP) [SOAP 1.1] and Web Services Description Language (WSDL) [WSDL 1.1]. Such services are often made available to all comers across the Internet, hence the name "web service".

widget

A simulated mechanism displayed by the user interface to enable the user to enter or alter data. Examples of widgets are radio buttons and popup-up lists.

X

XCI (XML Configuration Information)

Configuration information for a Presentation Agent output driver. The root element i n the XCI grammar is the config element ("Config Specification" on page 761).

XDC (XML Device Control)

A package within an XDP that holds information specific to a particular output device (such as a printer) or input-output device (such as a cell phone). This specification does not define the contents of the XDC package.

XDP (XML Data Package)

Provides a mechanism for packaging specific types of content within a surrounding XML container. The types of content include PDF, XML configuration information (XCI), dataSet, sourceSet, XSLT style sheet, XFA template, and XFDF (form data). XDP may also contain undocumented packets, such as those used to communicate events to a Form Server. The XDP format is intended to be an XML-based companion to PDF.

XFA (XML Forms Architecture)

An application of XML for modeling electronic forms and related processes. XFA provides for the specific needs of electronic forms and the processing applications that use them.

XFA is a collection of specifications, including template and data. XFA is a superset of XForms.

XFA application processor

A program which implements all or part of this document, the *XFA Specification*. DOM (Document Object Model) - a tree-structured set of data as represented internally inside an XFA processor. Although the word "object" suggests an object-oriented programming language, the XFA DOMs can be implemented in any language. Document order - the order in which the contents of a DOM would appear if written out as an XML document. To traverse a DOM in document order, start at the topmost node and perform a depth-first descent of the tree, descending from each node through its eldest child first, then upon returning to that node descending through the next-eldest child, and so on.

XFA Configuration DOM

The "XFA Configuration DOM" provides a set of software interfaces to the data obtained from an XFA configuration document. The "XFA Configuration DOM" includes sections for all of the different components of XFA, including a section for the data loader.

XFA Data DOM

The "XFA Data DOM" provides a set of software interfaces to the data loaded from an XML data document. The data in the "XFA Data DOM" is in general a subset of the data in the XML data document, but it may also contain other data not present in the XML data document as well as data that originated in the XML data document but has been modified.

XFA name

A string suitable for identifying an object in an XFA DOM, using the XFA Scripting Object Model syntax. A valid XFA name must be a valid XML name, as defined in the XML specification version 1.0 [XML], with the additional restriction that it must not contain a colon (:) character.

XFD (XML Form Data)

XML representation of the content of a form. XFA can employ any XML data but by convention the name XFD indicates form-specific data. For example, when a user partially fills out a form and saves the partial data as a file, the resulting file is conventionally known as an XFD.

XFDF (XML Forms Data Format)

XML representations of Adobe PDF annotations.

XFT (XFA Template)

The filename suffix and preferred namespace prefix for the XFA Template grammar.

XML (Extensible Markup Language)

A grammar for packaging arbitrary data using standard markup elements. XML is intended to be both human- and machine-readable. The controlling specification is [XML1.0], as modified by [XMLNAMES].

XML Data Document

Well-formed XML document containing data that is processed by XFA processing applications. Such a document is intended to be processed as data in the context of a form or workflow processing application, such as displaying or printing the data with a form, or manipulating the data via a workflow process.

XML data DOM

An "XML data DOM" provides a set of software interfaces to the data in an XML data document.

XMP (XML Metadata)

The filename suffix and preferred namespace prefix for XML Metadata, which is an XML representation of PDF metadata. Such metadata includes information about the document and its contents, such as the author's name and keywords, that can be used by search utilities.

XSS (XFA Source Set)

The filename suffix and preferred namespace prefix for the "source set" grammar.

!

$

&

.

@

\

0

9

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z