DISCLAIMER: This is a Draft version of the 'Adobe XFA' specification ("Draft Specification") and is subject to change without notice from Adobe. This Draft Specification does not represent the final version that Adobe may choose to publish, in its sole discretion, and may contain errors, omissions, inconsistencies and/or incorrect information, and is made publicly available on an "AS IS" basis only. Adobe disclaims any and all warranty or liability obligations of any kind with respect to the Draft Specification. Use and/or reliance upon the Draft Specification is done entirely at the user's own risk. ADOBE MAKES NO EXPRESS, IMPLIED, OR STATUTORY WARRANTY OF ANY KIND WITH RESPECT TO THE DRAFT SPECIFICATION INCLUDING, BUT NOT LIMITED TO, ANY WARRANTY WITH REGARD TO PERFORMANCE, MERCHANTABILITY, SATISFACTORY QUALITY, NONINFRINGEMENT OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL ADOBE BE LIABLE TO ANY PARTY FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES EVEN IF ADOBE OR ANY COMPANY REPRESENTATIVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
NOTICE: All information contained herein is the property of Adobe Systems Incorporated. Any references to company names in the specifications are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe is a registered trademark of Adobe Systems Incorporated in the United States and/or other countries.

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Mac OS is a trademark of Apple Computer, Inc., registered in the United States and other countries. JavaScript is a registered trademark of Netscape Communications Corporation. Unicode is a registered trademark of Unicode, Inc.

All other trademarks are the property of their respective owners.

This limited right of use does not include the right to copy other copyrighted material from Adobe, or the software in any of Adobe's products that use the Portable Document Format, 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 the Bibliography).

Intellectual Property

Adobe will enforce its intellectual property rights. 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 information interchange among diverse products and applications. Accordingly, Adobe gives anyone permission to use Adobe's intellectual property, 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 intellectual property, 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 Adobe intellectual property, as stated above, must include the appropriate intellectual property and patent notices.

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 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.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA. Notice to U.S. Government End Users. The Software and Documentation are “Commercial Items,” as that term is defined at 48 C.F.R. §2.101, consisting of “Commercial Computer Software” and “Commercial Computer Software Documentation,” as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA. For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.
# Contents

1 Changes Introduced in this Version ................................................................................................................. 5  
   New XFA Template Features ................................................................................................................................. 5  
   Deprecated XFA Template Features ....................................................................................................................... 6  

2 Introduction to the XML Forms Architecture (XFA) ............................................................................................ 7  
   Template ........................................................................................................................................................... 7  
   Data .................................................................................................................................................................... 8  
   Data Binding ......................................................................................................................................................... 8  
   Authoring and Sourcing ....................................................................................................................................... 9  
   Scripting ............................................................................................................................................................ 9  
   Role of DOMs in Form Processing ...................................................................................................................... 10  
   XDP .................................................................................................................................................................. 11  
   Internal Architecture/Lifecycle of a Form ........................................................................................................... 11  

3 Basic Form Templates ........................................................................................................................................ 12  
   Basic Form Structure ................................................................................................................................. 12  
   Form Structure .............................................................................................................................................. 15  
   Basic Composition .......................................................................................................................................... 20  
   Content Types ............................................................................................................................................. 34  
   Images .......................................................................................................................................................... 34  
   Barcodes ..................................................................................................................................................... 34  

4 Basic Forms ........................................................................................................................................................ 35  
   Basic Data Handling .................................................................................................................................... 35  
   Locale Handling .......................................................................................................................................... 35  
   XFA-Scripting Object Model ......................................................................................................................... 46  
   Basic Data Binding ..................................................................................................................................... 73  
   Form Processing ........................................................................................................................................ 98  
   Data Output ................................................................................................................................................. 99  

5 Intermediate Forms ......................................................................................................................................... 100  
   Intermediate Data Handling ....................................................................................................................... 100  
   Prototypes .................................................................................................................................................. 100  
   Intermediate Data Binding .......................................................................................................................... 105  
   Forms with Repeated Fields or Subforms ........................................................................................................ 111  
   Determination of Content Types in Data ...................................................................................................... 123  

6 Dynamic Forms .............................................................................................................................................. 124  
   Advanced Data Binding ........................................................................................................................... 124  
   Intermediate Layout ..................................................................................................................................... 137  
   Tables .......................................................................................................................................................... 187  
   Page Background ........................................................................................................................................ 191  
   Layout Objects .......................................................................................................................................... 193  

7 Automation Objects ......................................................................................................................................... 202  
   About ....................................................................................................................................................... 202  
   Calculation ............................................................................................................................................. 205  
   Validation ................................................................................................................................................ 207
8 Scripting ......................................................................................................................... 227
   Purpose of Scripting ........................................................................................................ 227
   Script Language Selection .............................................................................................. 227
   Selecting a Scripting Environment ................................................................................... 227
   Setting up the scripting environment ................................................................................ 228
   Exception handling .......................................................................................................... 228
   Picture clauses ................................................................................................................ 228
   Naked references in JavaScript ........................................................................................ 228
   Writing Scripts for calculate Elements ............................................................................. 228

9 Web Service Integration ................................................................................................ 229
   Submission ....................................................................................................................... 229
   Data Description and Connection Set .............................................................................. 231
   Schema and WSDL .......................................................................................................... 254
   Null handling .................................................................................................................. 254

10 User Experience .......................................................................................................... 255
   Detailed description of all widgets .................................................................................... 255
   Sequencing (tabbing orders) ............................................................................................. 255
   Accessibility .................................................................................................................... 255

11 Transformations .......................................................................................................... 256
   XSLT ............................................................................................................................... 256
   Built-in XFA-Data DOM Transformations .......................................................................... 256

12 XFA Template Element Reference .............................................................................. 257
   Guide to the Template Element Reference ....................................................................... 257
   Element Reference .......................................................................................................... 263

13 Rich Text Reference ................................................................................................... 479

14 Config Syntax Reference ............................................................................................ 480

15 LocaleSet Reference ................................................................................................... 481

16 Canonical Date Format Reference ............................................................................. 501

17 FormCalc Reference .................................................................................................... 502

18 XDP Element Language Reference ............................................................................ 503
   About the XDP Grammar ............................................................................................... 503
   XDP Element Language Syntax ..................................................................................... 507

A References ................................................................................................................... 513
   General ......................................................................................................................... 513
   Fonts and Character Encoding ...................................................................................... 515
   Bar Codes ...................................................................................................................... 516

Glossary ............................................................................................................................ 519
1 Changes Introduced in this Version

New XFA Template Features

Data Binding

In XFA 2.1 data binding is enhanced, compared to XFA 2.0, by the addition of the following features:

- **Dynamic forms.** These are forms that change in structure in accordance with the data. See “Static Forms Versus Dynamic Forms” on page 77 for detailed information.

- **Repeating subforms.** 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 111 for detailed information.

- **Explicit data references.** In XFA 2.1 the automatic data-binding logic can 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 90 for detailed information.

- **Subform sets.** Subforms can 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 131 for detailed information.

- **Record processing.** In XFA 2.1 data can 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 Record Processing for detailed information.

- **Globals.** In XFA 2.1 a field can 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 Globals 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.

Layout

**Dynamic Layout**

XFA 2.1 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.

**Flow Direction**

In XFA 2.1, layout can flow from left to right or from right to left.

In contrast, XFA 2.0 text layout could flow only from left to right.

**Deprecated XFA Template Features**

**Grouping TBD**

Save operation has been deprecated.
Introduction to the XML Forms Architecture (XFA)

The XML Forms Architecture (XFA) provides a template-based approach to building business form documents. XFA forms provide a wide range of features on several axes:

- **Workflow:** Data presentation, data capture and data editing, application front-end, printing
- **Dynamism:** From interactive, human edited forms with dynamic calculations, validations and other events to server-generated machine-filled forms
- **Complexity:** Single-page static forms, dynamic document assemblies based on data content, large production runs containing hundreds of thousands of transactions

XFA has always distinguished between template and content. The template defines presentation, calculations and interaction rules. Content is customer's application data. Though they are often packaged together, template and content are separate entities.

Template

XFA-Template was designed from the ground up to be an XML-based template language. That design follows a declarative model: elements in XFA-Template describe what the components of the form are in declarative XML, not how to construct or render them via script.

XFA-Template distinguishes between content defined in the template, and that which is defined later, for example when data is merged with the template or values filled in by an end-user. We often use the word *static* to refer to content defined in the template and dynamic to refer to content defined later. Static content is also referred to as boilerplate and includes lines, rectangles and static text. Simple dynamic content appears after binding some data with a template. Indeed, all content is dynamic to a degree in that the binding operation and user interaction may cause static objects to move or be resized.

XFA-Template also makes a distinction between content (e.g., text, lines, rectangles, images) and template objects that are containers of that content (e.g., fields). Four key container types define the document building blocks:

- **draw:** Container of static content
- **field:** Container of dynamic content
- **area:** Static container of other containers
- **subform:** Dynamic container of other containers

Each subform and area is a little form in and of itself. Subforms are assembled together, often in response to structure in the bound data, in order to create the final document. Subforms also support repeating, optional and conditional data groups. It is reasonable to define a repeating element once in the template, and rely on the data binding process to create enough instances to handle the data present. Not only is this less error-prone than defining multiple instances, but also the template author 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-Template 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.
Data

Typically, XFA dynamic content is the customer's XML data, matching the customer's schema. Data could also come from a database or other source (e.g., comma-delimited values). Often, form data elements are plain text, but may also include rich text and eventually graphics. Note that XFA uses the term *data* to refer to dynamic content.

XFA defines a data value to be a XFA name/value pair, where today the value is plain or rich text, or a graphic. Data values may contain nested data values, though today's implementation of XFA-Template cannot yet leverage this structure. An XFA name is a string suitable for identifying an object in an XFA DOM, using the XFA Scripting Object Model [XFA-SOM] syntax. 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 abstracted structure, inferred from the customer's data, to assist the XFA data binding process.

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 user 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 tool. Currently, Acrobat fills that role. In the past, we have developed our own form filling clients. In addition, the Adobe Form Server reads an XFA Template and produces a fillable form that matches the client's capabilities (e.g., basic HTML forms for older browsers, Microsoft DHTML where appropriate, WML for phones, through to rich client supported forms).

For more information, please see the chapter "Basic Forms" on page 35.

Data Binding

Generally, XFA 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 [XFA-Data-Binding]. Alternatively, the template designer may choose to provide data binding rules in the template definition. The user can also establish bindings at bind time through a simple binding language.

Default binding attempts to handle mismatches between the template structure and the data structure in a fairly passive way. In other words, it doesn't need to alter the structure of either. Naturally though, user-initiated changes to the data will be reflected if the data is saved.

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 multiple instances of singly-defined template elements. As part of binding, multiple run-time instances may be created to map the multiple instances present in the data.

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 two golden rules of 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

As an option, one can supply XSLT scripts for any XML processed by XFA. This includes both the template and data.

For a complete description of data binding, please see the XFA Data Binding Specification [XFA-Data-Binding]

### Authoring and Sourcing

There are two common ways in which a user creates a template: through the Designer, a graphical layout tool; or automatically by software (e.g., SAP Smart Forms conversion). In addition, since the XFA-Template language is XML-based, one can hand-edit XFA-Template files.

In the Designer, a template author 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, and template-defined bindings.

### 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 [XFA-FormCalc]. 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) [XFA-SOM]. A key feature of XFA-SOM is that it manages relative references. For example, when defining an invoice detail line in the Template Designer, the user might set up fields unitPrice, quantity and amount. The calculation for amount would simply be unitPrice*quantity. XFA-SOM would manage the scope in two ways: find the correct instances of unitPrice and quantity if the instances of those field names are in other subforms; and find the correct instances of unitPrice and quantity, 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 (e.g., saving the data).

Finally, scripts can be assigned to user actions, for example, onEnter, onExit, onClick, and so on.

Scripting is discussed in more detail in the chapter “Scripting” on page 227.

Role of DOMs in Form Processing

Note: Information on this topic will be provided in a later release of this specification.
XDP

The format "XML Data Package" (XDP) provides a mechanism for packaging units of form-related content within a surrounding XML container. Such form-related content may include PDF objects, XFA form data, and custom XFA-related form content. Packaging form-related content within an XML container may be important for XML-based applications that produce or consume XFA form content.

An XFA processing application may produce an XDP document when it submits a form (or a form's data) to a server or when web services are activated.

Internal Architecture/Lifecycle of a Form

This section describes the cycle of data binding, manipulation, updating back into the underlying XML.

Note: Information on this topic will be provided in a later release of this specification.

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. The method for requesting the save is application-specific. The format used for saving a form is typically to serialize the form data and any other parts of the form not represented by the original form components. Serialization of the XFA form data and other XML-based component would be structured according to the schema of the component being saved.

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.

Typically, a user would submit a form by clicking a button, where the underlying button object has a click event property, a validate property, and a submit property. Before the content submission is allowed to progress, the form data must be successfully validated. 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 state.
Basic Form Templates

This chapter describes basic form templates when used to produce static forms with uniquely named fields and data.

Basic Form Structure

In order to provide an understanding of what an electronic form is, this section provides some further background information.

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.

This subsection introduces key XFA concepts and terminology. “Form Structure” (below) expands on these concepts in the context of the structure of the form.

Anyone that has used a traditional paper form as a consumer has a very clear impression of what a “form” is. There is little thought given to the distinction between a blank form, and a form which has been completed. On the other hand, a form designer is concerned only with the creation of blank forms. Therefore, XFA clearly distinguishes between the two stages via the following terminology, as detailed in subsequent sections.

- Form — what the form consumer works with
- Template — what the form designer creates

Consider the following diagram:
FORM-1 — An example of a 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.

Form

A form is the result of combining a template with data. This includes the initial case of a blank form which is a form template with an empty data set. The formal definition is:

“A form is the combination of a template and state (data).”

An end-user (form consumer) interacts with a form, typically for one of the following reasons:

- To fill the form interactively
- To view a form which was filled by another user
- To print the form
- To submit the form to another process, such as a workflow

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 selects this template that the form-filling application uses to construct a “form”, which at first appears blank.

The user may fill out the form, and save the data content to a file, a database or in a package with a copy of the template. It is important to note that the data was saved, not the form (and not necessarily the template).

**Template**

As described in the previous section, a template represents the potential for a form. A template is a collection of related subforms and processing rules.

A form designer interacts with the form template for one of the following reasons:
- To design a new template from scratch or from a schema
- To design a new template based upon one or more existing templates
- To modify an existing template

For more information, please see how a template participates in “Form Structure”, or the syntax description of the template element.

**Content**

This specification uses the term content to refer to any piece of data that may appear on a form. Content includes, but is not necessarily limited to:
- Interactive data (enclosed in field elements)
- Static data (enclosed in draw elements)
- Textual data (both plain and rich)
- Graphical data, such as lines, boxes, images

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 (e.g., data type and limitations), while field elements are concerned with presentation and interaction.

For more information, please see the description of “Field Content” from the Form Structure perspective.

**Container**

The term container refers to an element that houses content. The container is concerned with all form-related aspects of dealing with its content, for example:
- Rendering, as described in “Rendering”
- User interaction, as described in “User Interface”
- Calculations and other interactions with other containers and content

For more information, please see the description of “Content” from the Form Structure perspective.
Field

A field is a container element that houses one piece of dynamic content in a form. As instances of forms are created from a common template, the static content is generally the same across those instances. However, the dynamic content is very likely to change from instance to instance.

The end user filling out a form effects change to dynamic content. Often, the user interacts with a field via data entry (typing). However, every interactive element on a form is a field. So, push buttons, check-boxes, and even some images are also fields.

The end user is not the only source of change to dynamic content. Applications, database input, dynamic calculations and many other sources can also change this content.

In diagram “FORM-1 — An example of a form”, call-outs indicate a few of the many fields. Note the push-button in the lower-right corner is also a field.

For more information, please see how a “Field” participates in form structure, or the syntax description of the field element.

Form Structure

This section describes the most significant XFA objects and their relationships. Such objects fall in the groups: “Containers”, “Content”, and “User Interface”.

Containers

The content within a form template is presented to the user by enclosing the content within container objects. Containers determine the layout and formatting of the form content and include the field element, as well as the draw, area, subform and exclusion group elements. This specification describes a variety of containers that vary in terms of their behavior and the types of objects that they can enclose.

Draw

Forms invariably contain static 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 static content. A user cannot directly interact with a draw object. Refer to the diagram “FORM-1 — An example of a form” for some examples of draw objects. Note that call-outs indicate only two of the many draw objects 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):

```xml
<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.
Field

A field object is the workhorse of a template and represents a data-entry region. A user is typically expected to interact with the fields by providing data input. Fields provide a pluggable user interface and support for a broad variety of content data-typing.

The following is an example of a field element that will produce a one-inch wide data-entry region capable of accepting textual input, positioned at an (x,y) coordinate of (0,0):

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

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

Area

An area is a grouping of form container objects ("Container"). The grouping itself is not visible, although the objects themselves may be visible. For example, in the diagram "FORM-1 — An example of a form", the vendor name and address data entry elements, along with the corresponding static text elements might be grouped into an area. Areas provide the designer with a means of organizing elements on a form, so that they may be moved or manipulated as a whole.

An area is itself a container of containers.

The following is an example of an area element that encloses two text fields:

```xml
<area x="1" y="2">
  <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.

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 diagram "FORM-1 — An example of a form". The form is really a collection of these sections and subsections, each of which XFA refers to as a subform. One can think of a subform as a sort of interactive area. Some of the features offered by subforms include:

- Management of scope of object names in scripting operations
- Validation of the content of the subform as a whole
- Hierarchical data binding

The subform likely 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:

```xml
<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>
```

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

An exclusion group is a non-geographical grouping of objects. By grouping the objects together, they exhibit mutual exclusivity semantics commonly associated within radio-buttons or ballot/check-boxes; only one of the objects is permitted to have a value or be selected by the user.

```xml
<exclGroup>
  <field name="Agree" x="0" y="0" w="1in" h="12pt">
    <ui>
      <checkButton/>
    </ui>
  </field>
  <field name="Disagree" x="0" y="12pt" w="1in" h="12pt">
    <ui>
      <checkButton/>
    </ui>
  </field>
</exclGroup>
```

For more information, please see the syntax description of the exclusion group element.

Template

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

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

```xml
<template>
  <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>
```

Container Value

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

Content

A form is composed of objects that the user perceives as the form content, such as the graphical and textual content that is part of the static form design, as well as the content present in the fields typically provided by a user. These content objects are arranged within the template coordinate space and presented to the user by enclosing the content within container objects such as draw, field, area, or subform elements. (See “Coordinates and Measurement” on page 20.)
Content is available in a variety of types. The following is the list of content types described by this specification:

- arc element
- boolean element
- date element
- dateTime element
- decimal element
- exData element
- float element
- image element
- integer element
- line element
- rectangle element
- text element
- time element

It should be recognized that not all types of content are appropriate everywhere on a form. Specifically, while it may be possible for all of the content types to be rendered on the form, it is likely that user will have a restricted set of content types that s/he may use for data input into the form. This is explained in the following two sections.

**Draw Content**

Draw content refers to the set of possible content elements that can all be enclosed inside a draw object, and this corresponds to the entire set of content types. Thus, while practically all content objects can be presented or “drawn”, a form filling application may not be capable of providing a method for the user to input all types of content objects.

For example, a typical form will have textual content presented on the form itself (providing field labels and instructional text to the user) and permit the user to input textual content into the fields of the form. Draw content type also includes rectangles and lines that provide the graphical composition, but the user would not typically have the ability to input a line or a rectangle into a field on the form.

**Field Content**

Field content is the subset of the available content types that a user can input with a form filling application and may be enclosed inside a field element. The majority of form field content is textual or numeric data.

This specification requires the following subset of content objects as the minimum that a form filling application should permit within a field container:

- arc element
- boolean element
- dateTime element
- decimal element
- float element
- integer element
User Interface

This specification 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 was motivated by the following three requirements:

- The form designer may want to exert some control over the user interface, selecting the widget most appropriate for each instance of a given type of content
- A future version of this specification might allow the form designer to select several different user interface elements for a particular piece of content, relying on each application to choose the one most appropriate to that application and its platform
- There may eventually be a way to include third-party controls as full participants on the form
- A future version of this specification may require that an application should be able to degrade gracefully, selecting from a set of alternatives provided by the form designer, or to an application-supplied default user interface, when a particularly rich one is not available

Structure

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.

Locale

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.

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.

A locale identifier is a unique string representing a locale. The structure and meaning of locale identifiers are defined in [RFC1766].

In general, every application operates in an environment where a locale is present; this is the ambient locale. 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 will default to English United States (en-US); this is the default locale.
Basic Composition

This section describes the most common aspects of how objects are arranged and presented on the presentation medium.

Coordinates and Measurement

Templates are composed in a space that has maps directly to the real world; the location and size of objects are expressed in real-world units such as inches and centimeters. There is no provision for addressing the device space of a display or print device directly in device units.

The origin of the template modeling space is located at the top-left corner, and the x and y coordinates increase in value towards the bottom-right.

This specification refers to the means of expressing coordinates and dimensional information as 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 several different font elements with the equivalent font size expressed in a variety of different measurements.

```xml
<font typeface="Helvetica" size="72pt"/>
<font typeface="Helvetica" size="1in"/>
<font typeface="Helvetica" size="1"/>
```

Value

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 object, the position of an object, or an offset from a coordinate.

The format of a measurement is:

- A value, consisting of
- 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 it shall be assumed that 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

Box Model

Every object that has a visual presence on a form presents itself through a rectangular region, or box, that describes the appearance of the object. This mechanism is known as the box model. All container objects conform to this model.

The box model describes a rectangular region with the following characteristics:

- A nominal extent corresponding to the perimeter of the box. The nominal extent is used for graphical placement, however, the actual rendering of the object may include graphic elements that draw somewhat outside of the nominal extent
- An optional graphical border that may be inset from the nominal extent
- An interior nominal content region for the presentation of content that may be inset from the nominal extent
- An optional Caption that appears within the nominal content region
The following diagram illustrates the box characteristics of an object; this particular example shows a box with margins and a border surrounding a gray ellipse content.

**BOX-1 — Box model**

**Nominal Extent**

Every container object has a nominal extent rectangle.

In a GUI-based graphical form designer application, the designer typically defines the nominal extents of positioned objects by using a pointing device to define a rectangular region, such as defining the rectangular region for a field object.

**Margins**

Margins independently inset the container’s content and border from its nominal extent. The specification of independent top, left, bottom and right margin values permit the insets to vary with each side of the box. Negative margins are reserved for future use.

Margins can be omitted or explicitly set to zero.

The application of margins to a nominal extent results in a rectangular nominal content region, which displays the container’s content. The particular type of enclosed content may also permit the specification of additional margining within the content region; this is referred to as content margins and is distinct from this description of box model margins. For example, rich text content may have individual paragraph margins independent of box model margins.

**Nominal Content Region**

The Nominal Content Region is the rectangular area left over after a 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.

**Borders**

The box model has an optional mechanism, known as borders, for rendering a rectangle around the content. Independent control over the appearance of sides and corners of the rectangle is provided.

The border has its own margins, independent of the content’s margins. This means that it is, in fact, possible to create an object whose border is located graphically within the nominal content region.
However, the content shall always have rendering precedence over the border, as described in “Z-Order” on page 33.

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, with the rendering algorithm being applied for all edges. The direction of rendering is important for the notion of handedness and edge element rendering.

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.

**Location**

In this specification, a container object is located relative to the its anchor point (typically the container’s top-left corner) of the enclosing container’s content region. This form of location is referred to as positioning. Content, on the other hand, is located in its container in a manner similar to how text is flowed within a word processing application. This is referred to as flowing.

A future version of this specification may relax the rule that containers are positioned and content is flowed. It may eventually be possible for containers to be flowed within their parents or for a content object to be positioned within its container.

Consider the following example and diagram. It includes a subform that:

- Has a blue border, inset from the subform’s nominal extent
- Encloses two positioned objects:
  - A draw object enclosing the text content “XML Forms Architecture”
  - A draw object enclosing content of an ellipse

---

**CONTAINER-1 — The relationship between content and containers**

Note that the small gray circles represent the anchor point positions of the objects, and are purely illustrative — they would not appear in the actual rendered form. Also note that nominal extents are indicated by broken lines. Again, these are purely illustrative and are not part of the rendering of the form.

The XFA syntax for the example might appear as follows:

```xml
<subform name="sample" x="0" y="0" w="3in" h="2in">
  <border>

```
Positioning

Positioned objects are located within a coordinate system mapped into the nominal content region of the enclosing container. The location is specified via a pair of measurements representing X and Y coordinates.

A positioned object is located by an anchor point which is typically the top-left corner of the container’s nominal extent. However, mechanisms are provided for altering this behavior as described below.

All positioned containers have three parameters that specify how they are positioned within their enclosing containers:

- **Nominal extent** — specified as W (width) and H (height) measurement attributes
- **Anchor point** — specified as X and Y measurement attributes, relative to the enclosing container’s nominal content region
- **Anchor point type** — specifying how the (child) container is positioned about the anchor point position

The following diagram shows where the anchor point types map onto the container’s nominal extent.

**POSITION-1 — Anchor point locations**

The enclosing container positions a child container in the following manner:

1. The anchor point specifies a coordinate within the enclosing container’s coordinate space, this is point Ep.
2. The anchor point type specifies a location about the child’s nominal extent, this is point Cp.
3. The point Cp (attached to the child) is positioned at point Ep (located within the enclosing container’s coordinate space).

For more information on the mechanics of the transformations that occur during positioning, see the section Transformations.

Flowing

While containers themselves are positioned, content is flowed inside an enclosing container. As illustrated by the following example, while a draw object may have an (x,y) coordinate that determines its position, the content inside the draw object (in this case some text) cannot assert a position and may only be flowed inside its enclosing draw object.

```
<draw x="0" y="0" w="2in" h="12pt">
  <value>
```

<text>Hello World</text>
</value>
</draw>

Flowed content delegates a great degree of control over its location to its enclosing container. The container typically employs some sort of 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 height of each line is determined by the line height.

**Line Height**

To flow text and other (future) 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. Line height 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.

**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 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.

This section describes the transformations required. Note that we use the term parent to refer to the enclosing container, and child to refer to the enclosed object.

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 child object knows and what the enclosing parent container knows. Regardless of margins and the internal coordinate origin, a child object makes coordinates available to the parent where (0,0) is the top-left corner of the child’s nominal extent. We refer to these as common coordinates — coordinates the parent can easily transform into its own coordinate space.

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:

\[
\begin{align*}
O_x &= Ax \quad (\text{TopLeft, MiddleLeft, BottomLeft}) \\
O_x &= Ax - W/2 \quad (\text{TopCenter, MiddleCenter, BottomCenter}) \\
O_x &= Ax - W \quad (\text{TopRight, MiddleRight, BottomRight}) \\
O_y &= Ay \quad (\text{TopLeft, TopCenter, TopRight}) \\
O_y &= Ay - H/2 \quad (\text{MiddleLeft, MiddleCenter, MiddleRight}) \\
O_y &= Ay - H \quad (\text{BottomLeft, BottomCenter, BottomRight})
\end{align*}
\]

Now, it's a very simple transformation to generate parent coordinates \((Px, Py)\) from common coordinates:

\[
\begin{align*}
Px &= CC_x + O_x \\
Py &= CC_y + O_y
\end{align*}
\]

Or,

\[
\begin{align*}
Px &= C_x + M_x + O_x \\
Py &= C_y + M_y + O_y
\end{align*}
\]

A slight optimization could be to avoid recalculating the invariants \((M_x + O_x, M_y + O_y)\) through a little cooperation between the parent and the child.

**Growable Containers**

The designer can designate a container object as being growable — its size may change at run-time. Growable objects are very useful in forms processing environments. Growth may occur along either the \(X\) and \(Y\) axes as follows:

- growth along both \(X\) and \(Y\) axes — for example, a region of freeform text on a form with no imposed width to force word-wrapping, and no limit on height
- growth in \(Y\) only — for example, the body text field of a memorandum form where many lines of input causes the field to grow vertically
- growth in \(X\) only — this is a type of growth that is unlikely to be required, and is reserved for future use

When a user utilizing GUI-based template design software places an container object on the template, the software application typically works form the outside-in. It starts with the rectangle drawn by the designer and applies the margins to determine the available nominal content region.

The growing ability of a container object tends to be more important at run-time. When a container object is growable, the run-time 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.

There are a number of implications that result from the ability of container objects to grow, as described in the following subsections. At a minimum, the requirement that container objects can grow means that all container objects must have the potential of attributes for minimum size (attributes minW/minH) and maximum size (attributes maxW/maxH). However, since majority of container objects on a form do not exhibit growth, we permit a simpler way to describe the size of a container object via width and height attributes \((w/h)\) respectively.

The interpretation of the value 0 for any of the minW, minH, maxW, maxH measurements is not to constrain the corresponding aspect of growth. For example, specifying minW as 0 allows the width to shrink to zero. Specifying maxW as 0 allows for unlimited growth of the width. The absolute omission of
any of these attributes causes the attribute to be interpreted as having the value 0, unless overridden by the \textit{w} and \textit{h} attributes.

Specifying the width and height via the \textit{w}/\textit{h} attributes, is exactly the same as specifying a \textit{minW} attribute that equals the \textit{maxW} attribute, and a \textit{minH} attribute that equals the \textit{maxH} attribute, which both equal the values specified in the \textit{w} and \textit{h} attributes respectively. In this case the processing application should always behave as if the \textit{min}/\textit{max} attributes exhibit values derived from the \textit{w}/\textit{h} attributes. Specifying \textit{w} or \textit{h} attributes in addition to a \textit{minW}, \textit{maxW}, \textit{minH}, or \textit{maxH} attributes is allowed, but the \textit{w} and \textit{h} attributes have precedence over the \textit{minW}/\textit{minH} and \textit{maxW}/\textit{maxH} attributes; that is, the \textit{minW}/\textit{minH} and \textit{maxW}/\textit{maxH} will be ignored in this case.

\subsection*{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 \textit{nominal content region} to appropriately enclose the content. As a result, this triggers a change in the container's \textit{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.

\subsection*{Growth and Enclosed 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 \textit{Z-order}. 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.

\section*{Formatting}

A container object may describe a number of formatting characteristics. This specification provides the following formatting characteristics:

- \textbf{Alignment}
- \textbf{Fill}
- \textbf{Caption}
- \textbf{Handedness}

\subsection*{Alignment}

Alignment is applicable to flowed content only, and determines how the content shall be positioned within the enclosing container object.

Alignment is divided into two axes, horizontal and vertical, with the following values which are combined to refer to a specific alignment:
• Vertical
  • top
  • middle
  • bottom
• Horizontal
  • left
  • center
  • right
  • justify
  • justifyAll

When text has an alignment of justify, space is inserted between words so that the first and last word of
the text lines are aligned against the margins. Normally, the last line of a justified paragraph is treated only
as left aligned, with a ragged right edge. An alignment of justifyAll causes justification to occur on the last
line as well.

The following diagram demonstrates how nine common alignments affect the positioning of text.

ALIGNMENT-1 — Nine common text alignments

Though alignment is most often thought of in terms of text, it applies to other forms of content as well.

For more information, please see the description of the para element.

Fill

Any closed graphic can be filled. Closed graphics include the box model border, as well as some content
objects, such as rectangle and arc. The fill element indicates how the region enclosed by the graphic 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 often think of this as specifying the background color for the fill. The fill element also has a child element specifying the type of fill (e.g., solid, pattern, stipple, linear, radial). This child, in turn, has its own child color element. This second color can often be thought of as the foreground color. For example, the following would create a fill of horizontal black lines on a gray background.

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

Note that the absolute omission or the presence of an empty fill element signifies solid white fill.

Caption

Each container object may include a caption. Some common purposes for the caption include:

- A text prompt or label that helps the end user understand the purpose of a text field (e.g. "Name:" or "Address:")
- A string of text that appears on the form and is part of the hit target for a field (for example, the text associated with a radio button or check box)
- The label on a push button

One can see that the rendering of the caption and its behavior are quite closely tied to the particular user interface for the container. Different user interface elements may place the caption in different ways.

For more information on captions, please see the description of the caption element.

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.

**Stroke Elements**

This specification describes two elements that represent strokes: edges and corners. Many XFA elements that represent graphical objects (such as lines, rectangles, and borders) have outlines that are rendered according to one or more strokes.

These elements possess 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:

```xml
<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>
```

Borders and Rectangles

Border and rectangle objects 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 object's graphical footprint doesn't exceed its nominal extent
Even-handed borders allow for alignment of container objects 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:

```xml
<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="right">
        <edge thickness="0.125in">
            <color value="128,128,255"/>
        </edge>
    </border>
    <value>
        <text/>
    </value>
</field>
```
<border hand="even">
  <edge thickness="0.125in">
    <color value="128,128,255"/>
  </edge>
</border>

<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>Left-handed borders</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>

**Rendering**

**Z-Order**

While the form coordinate space is two-dimensional, it should be recognized that container objects 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 and area encloses other container objects. The subform or area 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.

**Picture Clause**

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.

For more information, please see [XFA-Picture-Clause].

**Content Types**

This section examines the content types supported by XFA. Such content types include text, rich-text, and images.

*Note:* Information on this topic will be provided in a later release of this specification.

**Images**

*Note:* Information on this topic will be provided in a later release of this specification.

**Barcodes**

*Note:* Information on this topic will be provided in a later release of this specification.
4 Basic Forms

Basic Data Handling

This section does not address records or transformations.

Note: Information on this topic will be provided in a later release of this specification.

Locale Handling

In XFA a *locale* is a predefined set of conventions for representing dates, times, numbers, and currency. Each locale is identified by a short string defined in accordance with [RFC 1766]. For example, *en_US* is the locale identifier associated with English-language documents in the United States of America. Specifying *en_US* for a field’s locale tells the XFA application that, for example, a particular number is formatted as “12,345.67” whereas *fr_CA* (Canadian French) would mean that the same number would be formatted as “12.345,67”.

As first conceived the locale system provided for an application to operate in the appropriate locale where the computer was located. The operating system or some other facility was expected to provide a locale string known as the *ambient locale*. The application would then look up the details of the locale in some fixed table and know what to expect on input and produce on output. Unfortunately this simple plan failed badly, for a number of reasons. It did not meet the needs of multilingual or multi-currency environments. It did not provide for changes to the locale (for example the adoption of the Euro by many European countries). It did not provide for cross-locale data interchange. And although the locale names were largely standardized, no standards body stepped in to define the details of each locale. The result has been a hodge-podge of incompatible locale implementations. In addition many platforms have incomplete or faulty databases.

Despite it’s problems, locale is so deeply ingrained in software that XFA also makes use of it. However XFA includes some workarounds for locale’s problems.

The template schema allows for locale to be specified on a per-field, per-exclusion group, or per-subform basis. The [XML 1.0] standard already defines the xml:lang attribute. XFA makes this attribute available on all the appropriate elements and defines an inheritance rule.

Picture clauses can specify locales. When supplied the picture clause locale overrides the locale that would otherwise be inherited from the field, exclusion group, or subform.

Inside the Form DOM data is always stored in a *canonical* format. In this format numbers are formatted with an optional leading minus, with the decimal point signified by a “.” (period) character, and no grouping separator. Monetary amounts are also formatted as simple signed numbers without any currency symbol. Dates and times are formatted in a subset of [ISO 8601] format, which represents months and days numerically and always uses four digits for years.

The XDP schema includes a packet just for locale definitions. If these are supplied 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.
Conversions between canonical and local format are controlled by picture clauses contained in the template. A picture clause consists of one or more data pictures separated by "|" characters. A data picture (usually referred to here as just a picture) consists of a sequence of literals and operators. Some of the operators 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 only used if it matches the pattern of the input data. When the picture clause contains multiple pictures 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 form 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 form. Conservatism is desirable here because canonicalization often involves throwing away characters.

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 ambient 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.

It is recommended that locale definitions be included with a template when it is created. This ensures that the template creator and the template user see the same definition for the locale. However it would be inefficient to include all possible locales, as the definitions take about 3kbytes apiece. It is more sensible to include only those that are going to be used. Also, 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 via SOM expressions. Built-in locale definitions are not accessible this way, although of course the locales they define can still be used.

Scripts must not alter locale definitions. The result of any such attempt by a script is undefined.

Where and when localization?

The various XFA Data DOM and Form DOM maintain their contents in canonical form. When data in a localized form is entered or loaded the data is converted to canonical form as it is copied into the DOM. When data is presented in a human-readable form the canonical data is localized for presentation. The diagram below illustrates where conversions take place:
Localization is controlled by the operators in picture clauses. An individual picture clause can include any mixture of localizing operators and/or non-localizing operators. Up to four different picture clauses can be associated with a particular field, exclusion group, or subform. In addition, in the absence of a particular picture clause, some localizations are done automatically.

The four different picture clauses are:

**input picture clause**

This is declared as a picture element that is child to a ui element, as follows:

```
<field name="field1" ... >
  <ui>
    <picture> ... </picture>
  </ui>
</field>
```

This picture clause is used when accepting data keyed in by the user. It is used to canonicalize the data before placing it in the DOM.

**output picture clause**
This is declared as a picture element that is child to a format element, as follows:

```xml
<field name="field1" ... >
  <format>
    <picture> ... </picture>
  </format>
</field>
```

This picture clause is used when (re-)displaying the field, exclusion group, or subform. For example, when the user tabs out of a field the field is redisplayed using the format picture clause. This is used to localize the data retrieved from the DOM.

**bind picture clause**

This is declared as a picture element that is child to a bind element, as follows:

```xml
<field name="field1" ... >
  <bind>
    <picture> ... </picture>
  </bind>
</field>
```

This picture clause is used when merging data into a form @@tbd - and when saving merged data to an XML file?? It is also used when sending data to another computer via HTTP (a submit operation) or exchanging data with another computer using WSDL. The bind picture clause is used in both directions, to canonicalize data going into the Form DOM and to localize data being exported from the Form DOM.

**validate picture clause**

This is declared as a picture element that is child to a validate element, as follows:

```xml
<field name="field1" ... >
  <validate>
    <picture> ... </picture>
  </validate>
</field>
```

At validation time the data in the Form DOM (that is, in canonical format) is compared to the validation picture. If the data does not match the picture the validation fails. Because validation tests the data in canonical form, the same validation picture can be used in different locales.

Not all data is localizable. The following rules determine to which data localization applies.

**Rule 1**

Non-textual data (such as an image) is not localizable.

**Rule 2**

Text entered into a field with an input picture clause is canonicalized per the picture clause, provided it matches the picture clause. If it does not match the picture clause, or there is no picture clause, the text is assumed to have been entered in canonical form.

For example, assume a field has an input picture clause and a textEdit widget, as follows:

```xml
<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 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:

```xml
<field name="field1" ...
  <ui>
    <picture>D-MMMM-YYYY</picture>
    <dateEdit ... />
  </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:

```xml
<field name="field1" ...
  <ui>
    <picture>D-MMMM-YYYY|D-M-YYYY</picture>
    <textEdit ... />
  </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 3**

Text in a field or exclusion group 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:

```xml
<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:

```xml
<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**

Text displayed or printed from a field or draw with no output picture clause but containing float or decimal data is automatically localized if the UI widget is a `numericEdit`. This has no effect on the content in the DOM. Localization is limited to substituting the local radix symbol when displaying or printing. Separators are *not* inserted.

For example, a field is defined in the template as follows:

```xml
<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, but the type declaration still applies. Thus the field qualifies for automatic localization. Assume further that the locale is `fr_FR` and that after filling the content of the field is “1234.56”. When the field is printed, it appears as “1234,56”.

Note that input from the user is *not* canonicalized automatically. @@tbd – it seems likely we want the symmetrical functionality on input when there is no input picture clause??

**Rule 5**

Text in a field or exclusion group with a bind picture clause may be localized and/or canonicalized by the bind picture clause during merging or saving. The template schema allows a bind picture clause to be supplied for a subform as well, but it is hard to imagine a good use for this. Whether or not bind picture clauses for subforms are supported is implementation-defined.

For example, a field is defined as follows:

```xml
<field name="field1" ... >
  <bind>
    <picture>S9999.99</picture>
  </bind>
</field>
```

The bind picture clause is used to canonicalize the data when it is copied from the Data DOM to the Form DOM during a merge operation. The bind picture is used again to localize the data when it is saved from the Form DOM to a data file.

On input, if the picture clause contains more than one picture, the first picture that matches the supplied text is used. On output only the first picture in the clause is used. For example, the following field accepts data in either of two date formats:

```xml
<field name="field1" ... >
  <bind>
    <picture>D-MMMM-YYYY|D-M-YYYY</picture>
  </bind>
</field>
```

On input either “12-janvier-2004” or “12-01-2004” is canonicalized to “20040112”. On output the first picture is used.
Rule 6
Text in a field or draw that doesn't fall under any earlier rule is not localizable. On input it is copied literally into the DOM and on output it is displayed, printed, saved, or sent literally.

Rule 7
Scripts written in FormCalc can call functions to localize or canonicalize specific data, as discussed in “FormCalc and locale”. ECMAScript does not have any corresponding methods.

Picture clauses and locale

Picture clauses may contain any mixture of operators and literal text. Not all operators are locale-sensitive. The locale-sensitive operators are listed in this section.

Locale-sensitive number operators are shown in the following table:

<table>
<thead>
<tr>
<th>Date/time operator(s)</th>
<th>Locale definition element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>&lt;numberSymbol name=&quot;decimal&quot;/&gt;</td>
<td>Radix symbol for plain numbers (non-currency)</td>
</tr>
<tr>
<td>v</td>
<td>&lt;numberSymbol name=&quot;grouping&quot;/&gt;</td>
<td>Thousands grouping separator</td>
</tr>
<tr>
<td>(none defined)</td>
<td>&lt;numberSymbol name=&quot;percent&quot;/&gt;</td>
<td>Percent sign</td>
</tr>
<tr>
<td>s</td>
<td>&lt;numberSymbol name=&quot;minus&quot;/&gt;</td>
<td>Minus sign</td>
</tr>
</tbody>
</table>

Note: Although a percentage symbol can be defined, there is currently no way to invoke a localized percent sign via a picture clause. However a script could extract the localized percentage symbol for its own use.

Locale-sensitive currency operators are shown in the following table:

<table>
<thead>
<tr>
<th>Currency operator(s)</th>
<th>Locale definition element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>&lt;currencySymbol name=&quot;symbol&quot;/&gt;</td>
<td>Currency symbol</td>
</tr>
<tr>
<td>.</td>
<td>&lt;currencySymbol name=&quot;decimal&quot;/&gt;</td>
<td>Radix symbol for currency only</td>
</tr>
<tr>
<td>(none defined)</td>
<td>&lt;currencySymbol name=&quot;isoname&quot;/&gt;</td>
<td>ISO currency name (e.g. “USD”)</td>
</tr>
</tbody>
</table>
**Note:** Although the ISO currency code can be defined, there is currently no way to invoke a localized ISO currency code via a picture clause. However, a script could extract the localized ISO currency code for its own use.

**Note:** The DB and CR operators are not locale-sensitive.

Locale-sensitive date operators are shown in the following table:

<table>
<thead>
<tr>
<th>Date/time operator(s)</th>
<th>Locale definition element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEE</td>
<td>&lt;dayNames abbr=&quot;1&quot;/&gt;</td>
<td>Abbreviated day of week name</td>
</tr>
<tr>
<td>EEEE</td>
<td>&lt;dayNames abbr=&quot;0&quot;/&gt;</td>
<td>Full day of week name</td>
</tr>
<tr>
<td>G</td>
<td>&lt;eraNames/&gt;</td>
<td>Era name (e.g. &quot;BC&quot; or &quot;AD&quot;)</td>
</tr>
<tr>
<td>MMM</td>
<td>&lt;monthNames abbr=&quot;1&quot;/&gt;</td>
<td>Abbreviated month name</td>
</tr>
<tr>
<td>MMMM</td>
<td>&lt;monthName abbr=&quot;0&quot;/&gt;</td>
<td>Full month name</td>
</tr>
</tbody>
</table>

Locale-sensitive time operators are shown in the following table:

<table>
<thead>
<tr>
<th>Date/time operator(s)</th>
<th>Locale definition element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>&lt;meridiemNames/&gt;</td>
<td>Meridiem name (e.g. AM or PM)</td>
</tr>
<tr>
<td>Z</td>
<td>(can not be redefined)</td>
<td>Abbreviated time zone name</td>
</tr>
</tbody>
</table>

**Note:** The time zone name is localized but the localization cannot be redefined. For example, if the locale is en_US or en_CAN the time zone equivalent to GMT-5 is displayed as “EST”, which stands for “Eastern Standard Time”. When the locale is fr_CAN the same time zone is displayed as “HNE”, which stands for “Heure Normale de l’Est”. There is no way to change either the string “EST” or the string “HNE”.

**Note:** When the time zone is displayed as so many hours ahead of or behind Greenwich Mean Time (“GMT+N” or “GMT-N”), the “GMT” part of the string cannot be changed.

**FormCalc and locale**

FormCalc provides a facility `localdatefmt()` for generating a localized prompt based upon a non-localized picture clause. For example, a picture clause may specify that a date is to be entered as “YY-MM-DD”. In English, the picture clause itself makes a useful prompt. However, in French locales this prompt would likely puzzle the user. Instead we wish to substitute the prompt “aa-mm-jj” where “a” stands for “année”, “m” for “mois”, and “j” for “jour”. (The French prompt uses lower case letters because in French the names of months and days are not capitalized.) To accomplish this, a character substitution table is provided for each locale. Each symbol in the picture clause is mapped through the character substitution table to make the prompt string. For example, the French substitution table looks like this:

<table>
<thead>
<tr>
<th>Date or time symbol</th>
<th>Substitute</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>j</td>
</tr>
<tr>
<td>M</td>
<td>m</td>
</tr>
</tbody>
</table>
All other symbols are left as-is. The result is that a picture clause saying “YY-MM-DD” is translated, in French locales, into a prompt saying “aa-mm-jj”.

The character substitution table is encoded in the locale definition as an ordered string of substitute characters. The original string from which they substitute is:

```
GyMdkHmsSEDFwWahKzZ
```

Thus to achieve the substitutions shown in the above table, the string of substitute characters is:

```
GanjkHmsSEDFwWxhKzZ
```

Comparing the two shows that the desired substitutions are achieved, as shown in **bold** below:

```
GyMd\_kHmsSEDFwWahKzZ – original character list
GanjkHmsSEDFwWxhKzZ – mapped character list
```

For each locale definition in the `localeSet` packet, the string of substitute characters is contained in the `dateTimeSymbols` element.

Each locale also has four predefined date pictures of different verbosity levels and four predefined time patterns, also of different verbosity levels. The verbosity levels are named `short`, `med`, `long` and `full`. These four values correspond to the style numbers 1 through 4, respectively, that may be passed in as a parameter to various FormCalc date and time functions. Hence, FormCalc scripts that use these functions are automatically localized.

For each locale definition in the `localeSet` packet, predefined date pictures are contained within the `datePatterns` element and predefined time pictures are contained within the `timePatterns` element.

**Note:** There is no equivalent functionality in ECMAScript.

### The `localeSet` packet

The following example illustrates the overall structure of an XDP containing a `localeSet` packet:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xdp xmlns="http://ns.adobe.com/xdp/">
  <localeSet xmlns="http://www.xfa.org/schema/locale-set/2.1/">
    <locale name="fr_FR" desc="French (France)"
      <calendarSymbols name="gregorian">
        <monthNames>
          <!--Jan--><month>janvier</month>
          <!--Feb--><month>février</month>
          <!--Mar--><month>mars</month>
          <!--Apr--><month>avril</month>
          <!--May--><month>mai</month>
          <!--Jun--><month>juin</month>
          <!--Jul--><month>juillet</month>
          <!--Aug--><month>août</month>
        </monthNames>
      </calendarSymbols>
    </locale>
  </localeSet>
</xdp>
```
<!-Sep--> <month>septembre</month>
<!-Oct--> <month>octobre</month>
<!-Nov--> <month>novembre</month>
<!-Dec--> <month>décembre</month>

</monthNames>

<monthNames abbr="1">
<!-Jan--> <month>janv.</month>
<!-Feb--> <month>févr.</month>
<!-Mar--> <month>mars</month>
<!-Apr--> <month>avr.</month>
<!-May--> <month>mai</month>
<!-Jun--> <month>juin</month>
<!-Jul--> <month>juil.</month>
<!-Aug--> <month>août</month>
<!-Sep--> <month>sept.</month>
<!-Oct--> <month>oct.</month>
<!-Nov--> <month>nov.</month>
<!-Dec--> <month>déc.</month>

</monthNames>

<dayNames>
<!-Sun--> <day>dimanche</day>
<!-Mon--> <day>lundi</day>
<!-Tue--> <day>mardi</day>
<!-Wed--> <day>mercredi</day>
<!-Thu--> <day>jeudi</day>
<!-Fri--> <day>vendredi</day>
<!-Sat--> <day>samedi</day>

</dayNames>

<dayNames abbr="1">
<!-Sun--> <day>dim.</day>
<!-Mon--> <day>lun.</day>
<!-Tue--> <day>mar.</day>
<!-Wed--> <day>mer.</day>
<!-Thu--> <day>jeu.</day>
<!-Fri--> <day>ven.</day>
<!-Sat--> <day>sam.</day>

</dayNames>

<eraNames>
<!-BC--> <era>av. J.-C.</era>
<!-AD--> <era>ap. J.-C.</era>

</eraNames>

<meridiemNames>
<!-AM--> <meridiem>AM</meridiem>
<!-PM--> <meridiem>PM</meridiem>

</meridiemNames>

</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>GyMdkHmsSEDfWahKzZ</dateTimeSymbols>
<numberSymbols>
<numberSymbol name="decimal">,</numberSymbol>
<numberSymbol name="grouping"></numberSymbol>
<numberSymbol name="percent">%</numberSymbol>
<numberSymbol name="minus">-</numberSymbol>
</numberSymbols>
<currencySymbols>
<currencySymbol name="symbol">€</currencySymbol>
<currencySymbol name="isoname">EUR</currencySymbol>
<currencySymbol name="decimal">,</currencySymbol>
</currencySymbols>
</locale>

<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/locale-set/2.1/.

The order in which the locales appear is not significant. The information for each locale consists of seven sections, which may be present in any order. The sections are as follows:

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 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.

If any of the above sections is omitted, or an element or attribute omitted from within a section, the effect is to select the corresponding default value.

The `localeSet` element and its contents are described in detail in the schema reference section which follows.

### XFA-Scripting Object Model

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 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:
<?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.

Note: This diagram will be provided in a later release of this XFA Specification.

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:
<template xmlns="http://www.xfa.org/schema/xfa-template/2.1/"
    name="Template for receipt sample">
  <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>
    <field name="Sub_Total" ...> ... </field>
    <field name="Tax" ...> ... </field>
    <field name="Total_Price" ...> ... </field>
  </subform>
</template>
</xfd>

How Do I Use XFA-SOM?

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. For example:

```xml
<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 Multiple Nodes” on page 58, 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 $\text{\textit{sum}}()$ 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:

```xml
<?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>

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

```xml
<xfd ...
  <template ...
    <subform name="Receipt" ...
      <pageSet name="ReceiptPageSet" ...
        <subform name="Detail" ...
          <field name="Description" ...
          <field name="Units" ...
          <field name="Unit_Price" ...
          <field name="Total_Price">
            <calculate>
              <script>Units * Unit_Price</script>
            </calculate>
          </field>
        </subform>
        <subform name="Detail" ...
          <field name="Description" ...
          <field name="Units" ...
          <field name="Unit_Price" ...
          <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>
      </pageSet>
    </template>
  </subform>
</xfd>
```
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.**r**eceipt.**s**ub_**t**otal
```

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

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

**Shortcuts**

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

<table>
<thead>
<tr>
<th>Short and long forms</th>
<th>Short form examples</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$data</td>
<td>$data.Receipt.Sub_Total</td>
<td>Data that fills the form (Data DOM)</td>
</tr>
<tr>
<td>xfa.datasets.data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$template</td>
<td>$template.Receipt</td>
<td>Template for the form (Template DOM)</td>
</tr>
<tr>
<td>xfa.template</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$connectionSet</td>
<td>$connectionSet.ShoppingCart</td>
<td>Declares schema(s) or interfaces to host(s)</td>
</tr>
<tr>
<td>xfa.connectionSet</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Note: The meaning of $record varies depending whether record processing is enabled or not, as follows:

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 tens of thousands of records.

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 Data Handling 2.1 Specification for more information about record processing.

### Repeated Elements

When two or more 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):

$\text{data.Receipt.Detail}$

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:

<table>
<thead>
<tr>
<th>Short and long forms</th>
<th>Short form examples</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$form \text{ xfa.form}$</td>
<td>$form.\text{Receipt}$</td>
<td>Joined template and data after a merge operation (Form DOM)</td>
</tr>
<tr>
<td>$layout \text{ xfa.layout}$</td>
<td>$layout.\text{ReceiptPageSet}$</td>
<td>Form laid out on a page or pages (Layout DOM)</td>
</tr>
<tr>
<td>$host \text{ xfa.host}$</td>
<td>$host.\text{setFocus} \quad ($template.\text{Receipt.Tax}$)</td>
<td>Catch-all for objects that do not belong to a particular DOM</td>
</tr>
<tr>
<td>$record \text{ varies } (\text{See note below})$</td>
<td>$record.\text{Vendor.Name}$</td>
<td>Current record</td>
</tr>
<tr>
<td>$event \text{ xfa.event}$</td>
<td>$event.\text{name}$</td>
<td>Contains properties of the current event</td>
</tr>
</tbody>
</table>

! $\text{xfa.datasets.}$ !data.\text{PurchaseOrder.VendorCode}$ Does not require a “.” before the next name in the expression
$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, if the data document is changed to

```xml
<?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>
```

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 which takes a “name” attribute, rather than relying on the element tag to supply it with a name. The following example shows an XFA connection set that contains the named objects ShoppingCart, Catalogue, Shipping and TsAndCs.

```
<xfd ...
<connectionSet xmlns="http://www.xfa.org/schema/xfa-connection-set/2.1/">
  <wsdlConnection name="ShoppingCart" ...
  <wsdlConnection name="Catalogue" ...
  <wsdlConnection name="Shipping" ...
  <xmlConnection name="TsAndCs" ...
</connectionSet>
</xfd>
```

The above 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 and so is called transparent.

For example, if the receipt template is changed to omit the name attribute from the detail subforms:

```xml
<xfd ...
<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>
</xfd>
```

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. Instead, 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, so 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 resolvers makes them transparent because it is convenient to be able to insert or remove such second-class objects without being forced to modify scripts.

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.

Also within the Template and Form DOMs, the root subform object may have a child variables object. A variables object holds document variable objects, each of which is explicitly nameable. 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).

<xdp>
  <template ...
    <subform name="Receipt">
      <variables>
        <text name="CompanyName">AnyCo</text>
        <float name="TaxRate">7.25</float>
      </variables>
    </subform>
  </template>
</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.

Transparent objects are quite common inside the Form and Template DOMs but otherwise quite rare. Indeed the following example depicts the only place where potentially transparent objects can be found outside of the Form and Template DOMs.

<xfd ...
  <connectionSet ...>
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, in the template above the second detail subform object 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 named 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 Data Handling Specification 2.1 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 can not be explicitly named, although it is redundant. For example, when referring to the following configuration information

<xfd ...
   <config ...
       <present ...
           <copies>4</copies>
           ...
       </present>
   </config>
</xfd>

these SOM expressions are equivalent:

$config.present.copies $config.present.#copies

Attributes

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

For example:

<xfd ...
   <connectionSet xmlns="http://www.xfa.org/schema/xfa-connection-set/2.1/"
       <wsdlConnection name="ShoppingCart" dataDescription="cartDD">
           ...
       </wsdlConnection>
   <wsdlConnection name="Catalogue" ...)
The `dataDescription` attribute of the `wsdlConnection` named `ShoppingCart` can be referenced in a SOM expression as:

```
$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 the Data Handling Specification 2.1 for more information about loading attributes into the Data DOM.

**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:

```
<field name="Description">
  <event action="click">
    <script>
      if ($event.shift) then ...
    </script>
  </event>
</field>
```

$host is another object that is purely internal. The following template fragment invokes a method of $host to set the keyboard focus when the user tabs out of a field:

```
<field name="Unit_Price">
  <exit>
    <script>$host.setFocus(...)</script>
  </exit>
</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 a name clash between an attribute name and the name explicitly assigned in a child element.

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

The expression

\$\text{template.Detail.x}

refers to the explicitly named object which is the field named \textit{x}. By contrast the expression

\$\text{template.Detail.#x}

returns the attribute \textit{x} on \textit{Detail}, which has a value of \texttt{7.76mm}.

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

```
<xfd …>
  <template …>
    <subform name="Detail">
      <field name="name">Ernest</field>
    </subform>
  </template>
</xfd>
```

The XFA-SOM expression

\$\text{template.Detail.name}

returns the value of the field named \textit{name}, which is \texttt{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 \textit{name} attribute of \textit{Detail}, use “.#”.

For example,

\$\text{template.Detail.#name}

returns the value of the property \textit{name} on the \textit{Detail} subform which is the string \texttt{Detail}.

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:

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

\$\text{data.Receipt.Detail} \hspace{1em} \$\text{data.Receipt.Detail[0]}

resolves to Acme, whereas the expression

\$\text{data.Receipt.Detail[1]}

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

\$\text{data.Receipt.Detail[2]}

resolves to the node representing the second Detail element. This behavior is unique to the Data DOM. See the Data Handling 2.1 Specification for more information.

**Selecting Multiple Nodes**

The syntax “\*” can be used to select all child nodes, regardless of their names, which match the subsequent portions of the expression. For example, given that the following data is loaded into the Data DOM,

```
<?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>
```

the expression

\$\text{data.Receipt.\*} \hspace{1em} \$\text{data.Receipt.\*.Total_Price}

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

\$\text{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 the Data Handling 2.1 Specification for more information about loading of attributes into the Data DOM.

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

\$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.

**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. Instead the function of going up one level is performed by "..parent". For example, given the receipt data, the expression

\$data.Receipt.Detail.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. See the section Form DOM References for more information.

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. For example, given the original receipt example data,

```xml
<?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>
  </Detail>
</Receipt>
```
either of the expressions

$\text{data..Total_Price} \ \text{data..Total_Price}[0]$

resolves to $\text{data.Receipt.Detail.Total_Price}$, 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 $\text{data..Total_Price}[1]$ does not match any node, because there is no node corresponding to $\text{data.Receipt.Detail.Total_Price}[1]$.

### 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 any valid SOM expression can be used anywhere that FormCalc requires you to specify a DOM node.

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

$\text{data.Receipt.Detail[1].Units}$

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

$\sum(\text{data.Receipt.Detail.Total_Price[*]})$

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.

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

$\text{form.resolveNode(My_String)}$

There is also a $\text{resolveNodes()}$ method that returns a list of zero or more object references. For example:

$\sum(\text{form.resolveNodes(My_String)})$

Note: all nodes in all XFA DOMs have $\text{resolveNode()}$ and $\text{resolveNodes()}$ methods. Furthermore, for the types of SOM expressions described under Basic Object References, the $\text{resolveNode()}$ or $\text{resolveNodes()}$ method of any node can be used. However for the advanced expressions described in Relative References you must use the methods of the $\text{form}$ object or the $S$ object. It is recommended to use $\text{form}$ at all times to reduce the chance of error.
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-ecmascript it signifies that the language is ECMAScript. ECMAScript was formerly known as JavaScript.

ECMAScript, unlike FormCalc, is oriented towards objects rather than strings. It does not automatically try to resolve a node reference into a value. Hence to make reference to the value property of a field or other object in ECMAScript you must explicitly invoke the value property of the node. For example, to reference the value of the Tax field in the receipt example, you must use the SOM expression

\$data.Receipt.Tax.value

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. Instead scripts must pass such SOM expressions as strings to a resolveNode() method. 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 50, the resolveNode() method of any node can be used. However for the advanced expressions described in “Relative References” on page 62 you must use the methods of the $form object or this object. It is recommended to use the methods of $form at all times.

For example, the following line of code is valid:

\$data.Receipt.Tax.value = 11.34; // this is valid ECMAScript

Whereas the following line of code is not valid:

\$data.Receipt.Detail[1].Units = 3; // this is NOT valid ECMAScript - [] not allowed

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

$form.resolveNode("$data.Receipt.Detail[1].Units").value = 3; // this is valid ECMAScript

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. For example,

var aList = $form.resolveNodes("$data.Receipt.Detail[*].Units"); // this is valid ECMAScript

creates a variable containing a list of zero or more dataValues corresponding to Units elements of receipt detail records.

As with the resolveNode() method, there is a resolveNodes() method on every node in any XFA SOM, but the methods on different nodes do not all handle every type of SOM expression. It is recommended to use $form.resolveNodes() at all times.

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 50) can be used, however it must always resolve to a single node of the appropriate type in the Data DOM. See the Data Binding 2.1 Specification 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 the Template 2.1 Specification.

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.

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

```xml
<xfd ...
  <template ...
    <subform name="Receipt"...
      <field name="Tax"...> ... </field>
      ...
    <validate>
      <script>$ .Tax > 0</script>
    </validate>
    ...
  </subform>
</template>
</xfd>
```

In the example above, the full name of the referenced object is $form.Receipt.Tax.
For scripts written in ECMAScript, the name of the current container is “this” in native ECMAScript expressions but “$” in SOM expressions. For example, in the following template,

```
<xfd ...
  ...
    <script
        contentType=
            "application/x-ecmascript"> this.value > 0  // ECMAScript
    </script
    ...
</subform
</template
</xfd
```

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.value > 0  // ECMAScript
```

Or, the script could have used the `resolveNode()` method on the `$form` object as follows:

```
$form.resolveNode("$.Tax").value > 0  // ECMAScript
```

Note: all nodes in all XFA DOMs have `resolveNode()` and `resolveNodes()` methods. Furthermore, for the types of SOM expressions described under Basic Object References, the `resolveNode()` or `resolveNodes()` method of any node can be used. However for the advanced expressions described in this section (Relative References) you must use the methods of either the `$form` object or the `this` object. It is recommended to use the methods of `$form` at all times to reduce the chance of error.

### 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.

```
<xfd ...
  ...
    <script>Tax > 0</script>
    ...
</subform
</template
</xfd
```

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 $form.Receipt.Tax

The equivalents in ECMAScript are:

this.Tax.value  // ECMAScript native expression $form.resolveNode("Tax").value
// ECMAScript SOM expression $form.resolveNode("$.Tax").value  // ECMAScript
SOM expression $form.Receipt.Tax.value  // ECMAScript native 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:

\[
<\text{feld name="Sub_Total" ...}> ... <\text{ield name="Tax" ...}> ... <\text{ield name="Total" ...}>
<\text{alculate}>
<\text{cript}>\text{Sub_Total} + \text{Tax}</script>
<\text{alculate}>
</\text{ield}>
</\text{ield}>
</\text{subform}>
</template>
</xfd>

The equivalent in ECMAScript is:

\[
$\text{form.resolveNode("Sub_Total").value} + \text{\$form.resolveNode("Tax").value}  // \text{ECMAScript}
\]

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:

\[
<\text{feld name="Description" ...}> ... <\text{ield name="Units" ...}> ... <\text{ield name="Unit_Price" ...}> ... <\text{ield name="Sub_Total" ...}>
<\text{alculate}>
<\text{cript}>\text{Units} \times \text{Unit_Price}</script>
<\text{alculate}>
</\text{ield}>
</template>
<\text{feld}>
</\text{subform}>
<\text{feld name="Detail" ...}>
<\text{ield name="Description" ...}> ... <\text{ield name="Units" ...}> ... <\text{ield name="Sub_Total" ...}>
<\text{alculate}>
<\text{cript}>\text{Units} \times \text{Unit_Price}</script>
<\text{alculate}>
</\text{ield}>
</template>
</xfd>
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 contains:

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

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 124 for more information about templates for dynamic forms, and “Basic Data Binding” on page 73 for more information about how occurrence numbers affect the merge process.

**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 following template:

```
<xd:
  <template ...
    <subform name="Receipt" ...
      <subform name="Detail" ...
        <field name="Description" ...
        <field name="Units" ...
        <field name="Unit_Price" ...
        <field name="Sub_Total"
          <calculate>
            <script>Units * Unit_Price</script>
          </calculate>
        </field>
      </subform>
    </subform>
  ...
</template>
</xd:
```
The template is modified as follows:

```xml
<xfd ...
<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>
</xfd>
```

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 with the same original fragment, if a new subform is wrapped around Units and Unit_Price, it is necessary to modify the script that calculates Sub_Total as follows:

```xml
<xfd ...
<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>
... ...
</template>
</xfd>
```
Unqualified References Summary

A SOM expression is qualified if the first character is “$” or “!” or if the first term in the expression is “xfa”. 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 fully-qualified expression, yet it is relative to the script container. Thus it escapes scope-matching without giving up relocation. For example, in the following template

```
<xfd ...
  <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>
</xfd>
```

the expression 

$_.Sub_Total

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

“$.” can also be used for expressions pointing to nodes that are higher up in the hierarchy than the script’s container. Use “$.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:

```xml
<xfd ...>
  <template ...>
    <subform name="Receipt" ...> ... </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>
    <field name="Description" ...> ... </field>
    <field name="Units" ...> ... </field>
    <field name="Unit_Price" ...> ... </field>
    <field name="Total_Price" ...> ... </field>
  </subform>
</template>
</xfd>

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:

```xml
<xfd ...>
  <template ...>
    <subform name="Receipt" ...> ... </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>
    <field name="Description" ...> ... </field>
    <field name="Units" ...> ... </field>
    <field name="Unit_Price" ...> ... </field>
  </subform>
</template>
</xfd>
```
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_Price[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:

```javascript
$form.resolveNode("Units").value * $form.resolveNode("Unit_Price").value
```

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:
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 contains:

```xml
<xfd ...
<subform name="Trip">
  <field name="Speed" ...> ... </field>
  <subform name="Distance" ...
    <occur min="5" max="5" />
    <field name="Cell"...> ... </field>
    <field name="Cell"...> ... </field>
    <field name="Cell"...> ... </field>
    <field name="Cell"...> ... </field>
    <field name="Cell"...> ... </field>
  </subform>
</subform>

<subform name="Time" ...>
  <subform name="Times" ...
    <calculate>
      <script>Distance.Distances.Cell / Speed</script>
    </calculate>
    <field name="Cell"...
    <calculate>
      <script>Distance.Distances.Cell / Speed</script>
    </calculate>
    <field name="Cell"...
    <calculate>
      <script>Distance.Distances.Cell / Speed</script>
    </calculate>
    <field name="Cell"...
    <calculate>
      <script>Distance.Distances.Cell / Speed</script>
    </calculate>
    <field name="Cell"...
    <calculate>
      <script>Distance.Distances.Cell / Speed</script>
    </calculate>
    <field name="Cell"...>
</xfd>
```
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

$\text{template.Trip.Time.Times}[3].\text{Cell}[2].\text{Distance.Distances.Cell}$

is resolved as follows:

1. The current container is a field named Cell, specifically

   $\text{template.Trip.Time.Times}[3].\text{Cell}[2]$. The Cell field does not have a property or child named Distance.

2. The Cell field's parent is a subform called Times, specifically $\text{template.Trip.Time.Times}[3]$. 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 $\text{template.Trip.Time}$. Time has a sibling called Distance. Hence Distance is resolved to $\text{template.Trip.Distance}$.

4. $\text{template.Grid.Distance}$ 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, $\text{template.Trip.Times.Times}[3]$, has an index. Its index is borrowed and $\text{template.Trip.Distance.Distances}[3]$ is selected.

5. $\text{template.Trip.Distance.Distances}[3]$ 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, $\text{template.Trip.Times.Times}[3].\text{Cell}[2]$, has an index. Its index is borrowed and $\text{template.Trip.Distance.Distances}[3].\text{Cell}[2]$ is selected.

6. Because the script language is FormCalc, and

   $\text{template.Trip.Distance.Distances}[3].\text{Cell}[2]$ has a value property, the expression is resolved as $\text{template.Trip.Distance.Distances}[3].\text{Cells}[2].\text{value}$.

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 $\text{template.Trip.Speed}$. Because that field has no siblings with the same name, no index is required.

8. Because the script language is FormCalc, and $\text{template.Trip.Speed}$ has a value property, this expression is further resolved to $\text{template.Trip.Speed.value}$.

Finally the calculation can be done.

9. The number in $\text{template.Trip.Distance.Distances}[3].\text{Cell}[2].\text{value}$ is divided by the number in $\text{template.Trip.Speed.value}$ and the quotient assigned to $\text{template.Trip.Time.Times}[3].\text{Cell}[2].\text{value}$. 

```xml
<calculate>
  <script>Distance.Distances.Cell / Speed</script>
</calculate>
```
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. This is the same thing that 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 given the template

\[
\textbf{xfd} \ldots
\]

\[
\text{<template } \ldots
\]

\[
\text{<subform name="root" } \ldots
\]

\[
\text{<subform name="A" } \ldots
\]

\[
\text{<subform } \ldots \text{/subform } \ldots
\]

\[
\text{<subform } \ldots
\]

\[
\text{<field name="C" } \ldots
\]

\[
\text{<calculate } \ldots
\]

\[
\text{<script } \ldots \text{ A.} \text{#subform } \ldots \text{/script } \ldots
\]

\[
\text{</calculate } \ldots
\]

\[
\text{</field } \ldots
\]

\[
\text{</subform } \ldots
\]

\[
\text{</subform } \ldots
\]

\[
\text{<subform name="B" } \ldots
\]

\[
\text{<subform } \ldots \text{/subform } \ldots
\]

\[
\text{<subform } \ldots \text{/subform } \ldots
\]

\[
\text{</template } \ldots
\]

\[
\textbf{xfd}
\]

the expression $\text{B.} \text{#subform}$ resolves to $\text{$form.root.B.} \text{#subform[1]}$. The index "[1]" is copied from the corresponding level in the current object's full name, which is $\text{form.root.A.} \text{#subform[1].C}$.

**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 template the first Item_No field defaults to 1. After than each Item_No field is automatically set to one more than the previous Item_No field. Hence, if the user manually enters a value into the first Item_No field all subsequent fields are automatically updated.

\[
\textbf{xfd} \ldots
\]

\[
\text{<template } \ldots
\]

\[
\text{<field name="Item_No" } \ldots
\]

\[
\text{<value } \ldots
\]

\[
\text{<integer }1\text{</integer } \ldots
\]

\[
\text{</value } \ldots
\]

\[
\text{</field } \ldots
\]

\[
\text{<field name="Description" } \ldots \text{/field } \ldots
\]

\[
\text{<field name="Units" } \ldots \text{/field } \ldots
\]

\[
\text{<field name="Unit_Price" } \ldots \text{/field } \ldots
\]
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” and 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.

### Basic Data Binding

The reader is assumed to be familiar with the principles and vocabulary of XML, as set out in *Extensible Markup Language (XML) 1.0 (Second Edition)* [XML]. The reader is also assumed to be familiar with the following concepts:

- Overall principles of XFA processing of user data, as described in “Basic Forms” on page 35
- Structure of an XFA form template, as described in “Basic Form Templates” on page 12
- SOM expressions (including scope matching), as described in “XFA-Scripting Object Model” on page 46

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 upon the Template DOM. Optionally, the Data DOM may be restructured to make it consistent with the structure of the Form DOM, thereby forcing the data into a shape imposed by the template.

Note: A revision to the previous paragraph is pending.

Data binding is also known as “merging” because it can be thought of as merging the data 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 DOM alone. Optionally, default data matching the Form DOM may be inserted into the Data DOM.
Conventions

Within sample XML data indenting has been added for readability. The indentation is not a requirement, nor is it meaningful.

Many drawings in this chapter depict relationships between nodes in tree graphs. Nodes are depicted using different shapes and shadings, as shown in the following figure.

![Diagram of node symbols]

**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 PF 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.
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 template packet would follow the pattern:

```
<template ...>
  <subform name="A">
    <subform name="B">
      <field name="C" ... />
      <field name="D" ... />
    </subform>
  </subform>
</template>
```

The Form DOM is created by the data binding process in a manner defined by this specification. The individual nodes in the Form DOM are copied from prototype nodes in the Template DOM. 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 the value shown in the data node is 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 are explained elsewhere in this specification.

Notwithstanding the use of terms such as “DOM” which are associated with particular internal data structures, 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.
Design Principles

Data binding lies at the heart of data independence. 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 the remaining sections of this specification. The following is a simplified overview:

The data binding process walks through the Template DOM populating the Form DOM with nodes. If a Data DOM was supplied it attempts to match up each new form node with a data node in accordance with the following rules:

- The relative order of same-named data values or groups is significant.
- The relative order of uniquely-named data values or groups is not significant.
- The hierarchy of structure described by data values or groups is significant.

These rules 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.

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 Record Processing for more information.

This value is new in XFA 2.1.

dataRef

This is only allowed if the parent object is a field. The field will bind to the node in the XFA Data DOM specified by the accompanying “ref” property. This value is new in XFA 2.1.

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 “Locale Handling” on page 35

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.

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 for more information.

This property is new in XFA 2.1.

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. In XFA 2.0 static forms were the only type of form supported. XFA 2.1 also supports dynamic forms.

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.

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, new to XFA 2.1, 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 available in XML schemas.

Note that the principal difference between data binding in XFA 2.0 and in XFA 2.1 is that XFA 2.1 adds support for dynamic subforms and for repeating static subforms. XFA 2.0 requires that each subform be singly-occurring. XFA 2.1 data binding is completely backward-compatible with templates and data streams designed for use with XFA 2.0. Moreover, the same data that is used with a static form can also be used with a dynamic form.

Forms with Uniquely Named Fields and Subforms

The simplest kind of static form has a unique name for each subform or field and for each corresponding data group or data value. For example, suppose an online registration form is being used to gather first and last name and mailing address. The resulting data submitted to the host looks like this:
<?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).
The template was created with field names that match the data elements one-for-one. A highly simplified skeleton of the template looks like this:

```
<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).
The Form DOM resulting from this operation represents the binding between data and template. The relationship between the three DOMs is shown below.

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.

For this simple case the data binding process can be described as follows:

1. Create form nodes as copies of template nodes
2. Match non-attribute data nodes to form nodes
3. Match attributes to unmatched form nodes
4. Re-normalize
5. Perform calculations and validations
6. Issue the form ready event

The following subsections expand on some of the steps above.

Matching Data Nodes with Form Nodes

In this case 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" dataValue 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.

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.

If the data had extra elements, with names not matching anything in the template, the extra data nodes would simply be left unbound. The resulting Form DOM would in effect represent a subset of the data. Applications can therefore use multiple templates with the same data to present different views of the data.

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:

```xml
<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. Considering the same data again:

```xml
<?xml version="1.0"?>
<registration>
  <first>Jack</first>
  <last>Spratt</last>
  <apt>
    <street>99 Candlestick Lane</street>
    <city>London</city>
    <country>UK</country>
    <postalcode>SW1</postalcode>
  </apt>
</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.
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.

Matching Hierarchy

It is possible for a single data node to directly match one field node and scope match some others. However, the data-binding process can only bind the data node to one form node. To resolve this conflict 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 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 matches 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 “XFA-Scripting Object Model” on page 46.

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 84. All of the scope matches in the following figure are ancestor matches.
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.

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.

Indirect matching is fine when each field on a form has a unique name. Unfortunately sometimes data values in different data groups share the same name, although they are logically distinct. This is not a problem if the data values sharing the name cannot scope-match to each other’s fields. For example if they are cousins (sharing no ancestor closer than a grandparent) there is no problem. However there are circumstances when it is a problem. For example, consider the following fragment from a passport application:

```xml
<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>

This template is merged with the following data:

```xml
<application>
  <lastname>Abott</lastname> ... 
  <sponsor>
    <lastname>Costello</lastname>
  </sponsor>
</application>
```
The result is shown in the following figure.

**Scope-matching causes an undesirable binding**

**Note:** The example shown above and discussed below is incorrect and will be revised.

This is not the desired result. The “lastname” field within the “sponsor” subform is bound to the applicant’s last name. This comes about because “$data.application.lastname” scope-matches to “$form.application.sponsor.lastname”. After this binding took place, the “sponsor” subform failed to bind to any data groups, because there were no more unbound data groups. This in turn resulted in the “lastname” field for the applicant being left unbound. Hence when the form is displayed the applicant’s last name shows up as the sponsor’s last name and the applicant’s last name remains blank.

Fortunately, there are a number of remedies available. One remedy is to change a field name so that all field names are unique, but this requires either changing the data or renaming it on the way into the Data DOM using a configuration option as described in “Basic Forms” on page 35. Another remedy is to make sure that the fields occur in the template in the same order as they occur in the data, but that only works if the data is always in the same order. Finally, subforms can be qualified with explicit data references so that they can only bind directly to the desired data groups, but this 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:
The result using this template fragment and the same data is shown in the following figure. This is the desired result.

**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:

```xml
<template>
  <subform name="application">
    <subform name="sponsor">
      <bind match="dataRef" ref="$data.application.sponsor">
        <field name="lastname"> … </field> <!-- sponsor's last name -->
        ...
      </subform>
      <field name="lastname"> … </field> <!-- applicant's last name -->
      ...
    </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.
Nameless fields are also transparent, unless they make explicit data references. (Explicit data references are explained below in Explicit Data References.) 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.

Nodes representing “area” elements are also transparent to data binding, even when they are named. This is because they are reserved for use by template creation tools; by definition they can not have any effect at run time. There are also various nodes that are descended from subforms but are never copied into the Form DOM because they can not contain subforms or fields, for example “pageSet” nodes. “Basic Form Templates” on page 12 for more information about these objects.

**Explicit Data References**

It is possible for the template to override the automatic matching process. When a field has a “bind” property with a “match” attribute having a value of “dataRef”, the accompanying “ref” attribute supplies a SOM expression which can refer to any place in the Data DOM. Hence the dataRef node can bind to any node in the Data DOM, regardless of the location of the dataRef node in the Form DOM. On the other hand there is no need to search for the data node to which the dataRef field binds – the SOM expression explicitly identifies the node to bind. If there is no data node matching the expression, the binding process creates one with default properties. If the expression specifies ancestor nodes that do not exist, they too are created so that the referenced node can link into the Data DOM at the specified location.

When a field supplies an explicit data reference, the data binding process does not use the “name” attribute of the dataRef field. For this reason the name can be omitted for such fields (i.e. they can be anonymous).

Note that a field must not explicitly bind to a data node which is already bound.

**Exclusion Groups**

An exclusion group is a template construct that contains a set of objects, each of which has an activated state and a deactivated state. More than one member of the set must not be activated at the same time. For example, in an interactive context the members of an exclusion group are commonly fields presented in the GUI as radio buttons. When one radio button is turned “on” (depressed) all of the other radio buttons in the group are forced “off” (released). It is also permissible for every button to be “off”. Each radio button is associated with a key value. When a button is depressed a variable is set to the key value for that button. At the same time the node representing a button 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.

```xml
<subform name="main" ...>
  <exclGroup name="sex">
    <field name="male">
      <item><text>M</text></item>
      <item><text></text></item>
    </field>
    <field name="female">
      <item><text>F</text></item>
      <item><text></text></item>
    </field>
    <field name="NA">
      <item><text>NA</text></item>
      <item><text></text></item>
    </field>
  </exclGroup>
</subform>
```
Inside the Template DOM the exclusion group is represented by a node, as shown (right). The node exists purely to encapsulate the logical relationship between its children. It is not itself displayable.

Check boxes are more flexible than radio buttons. A check box field can be set to any of three values. When set to the third value the check box is greyed out in the GUI. The third value is enumerated in the template by a third “item” element.

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.

```xml
<?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.
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.

```xml
<?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.

Attributes

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 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 of data values that are bound via
explicit data references are excluded.

For example, suppose the data says:

```xml
<?xml version="1.0"?>
<registration>
  <first>Jack</first>
  <last>Spratt</last>
  <street apt="">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 are ignored.

**Re-Normalization**

**Note:** This section is out of date and will be revised in a later release of the specification. Re-normalization
now uses the data description.

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 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 example that was used above to illustrate scope matching will also serve to illustrate re-normalization.
The template has the following skeleton:

```xml
<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 as follows:

```xml
<?xml version="1.0"?>
<registration>
  <first>Jack</first>
```
The following figure shows the result of the data binding process, before re-normalization.

[Diagram showing the result of registration binding with template (duplicated from page 81)]

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.
If the re-normalized Data DOM is subsequently written out in XML the result is:

```xml
<?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.
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

Note that since this specification does not dictate a particular Application Programming Interface (API), any given implementation may omit re-normalization, make it compulsory, or provide it as an option. However if it is performed it must conform to this specification.
Calculations and Validations

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 “Scripting” on page 227.

Form Ready Event

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 these and other scripts and events see “Scripting” on page 227.

Remerge and Incremental Merge

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 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 by a script, for example after receiving updated data from a web service.

Form Processing

Note: Information on this topic will be provided in a later release of this specification.
Data Output

Note: Information on this topic will be provided in a later release of this specification.
Intermediate Data Handling

Note: Information on this topic will be provided in a later release of this specification.

This section describes intermediate data handling, including records.

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

Almost any XFA template element can be enclosed within a proto element. An element enclosed in such a way is called a prototype. A prototype may have child elements. Though these are part of the parent prototype, they are not prototypes themselves. Only the direct children of proto elements are considered prototypes. For example, the following defines a prototype font with a particular color, but the color is not itself a prototype.

```
<proto>
  <font id="HELV-RED"
    typeface="Helvetica"
    size="10pt"
    weight="normal"
    posture="normal">
    <color value="255,0,0"/>
  </font>
</proto>
```

Prototypes don’t participate directly in the form. Instead, their participation is indirect only, when referenced by another element. For example, a prototype field element, even though fully specified, 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. For example, the following are equivalent.

```
<subform>
  <proto>
```

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 its use attribute. This takes a fragment identifier as a reference to the prototype element. For a successful reference to occur, the reference must satisfy all three of the following:

- Refer to a single element
- Refer to a direct child of a proto element
- Refer to an element that is of the same type as the referencing element

For example:

```
<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.
Prototypes may reference other prototypes. In addition, descendant elements of a prototype may reference prototypes. For example,

```xml
<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>
```

This achieves the same effect as the previous example, only the prototype font and its color element refer to other prototypes.

### 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, consider the following:

```xml
<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>

This 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.

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 application default.

### Overriding Data Content

The presence of data content in a referencing element overrides data content from the prototype. For example:

```xml
<proto>
  <text id="TEXT"/>default TEXT</text>
</proto>
<field ...>
  <value>
    <text use="#TEXT">Overriding text</text>
  </value>
</field>
```

The text value of the field will be “Overriding text”. Note that 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. Consider the following example:

```xml
<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>

Here we have a field that inherits from a prototype field element. 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 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 other words, 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:

```
<proto>
  <border id="#DEFAULT-BORDER">
    <edge thickness="2pt"/>
    <edge thickness="1pt"/>
  </border>
</proto>
<field ...
  <border use="#DEFAULT-BORDER">
    <edge thickness="3pt"/>
  </border>
  ...
</field>
```

The prototype border specifies two edges, which are taken as the top/bottom and left/right edges. Using the prototype without any overrides would therefore result in 2pt 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.
Intermediate Data Binding

The Occur Element

Each subform and subform set object in XFA 2.1 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. The `occur` property is new in XFA 2.1.

In a template the `occur` property is expressed as an “occur” element. It is always the child of a subform or subform set. When the `occur` element is missing all of its properties default to “1”. If the element is present but any of its attributes is missing, the property associated with that attribute defaults to “1”. Settings of “1” reproduce XFA 2.0 behavior, ensuring backwards compatibility with XFA 2.0 templates.

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:

```xml
<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 on nested subforms**

When the “initial” attribute is not supplied, this property defaults to “1”. Note that the root (outermost) subform in a template must have its “initial” property defaulted or explicitly 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:

```xml
<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 “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.

**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.
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.

```xml
<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.
Forms with Repeated Fields or Subforms

Static forms may also 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.

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.

As shown (left), this year the club has only two members. The complete data document follows:

```xml
<?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>
```

Anytown Garden Club
2023 Anytown Road
Anytown, USA

Date 01/01/04

Membership List

John Brown
Betty White

Filled Static Form as Printed
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. See the “Scripting Object Model Expression Specification” [SOM] for more information.

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:

```xml
<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. 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.
The following figure shows the DOMs for the membership form after data binding.

![Diagram of DOMs for membership form]

**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, exactly 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:

```xml
<?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>
```
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:

```xml
<?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:

```xml
<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>
  </subform>
</template>
```

When this is loaded into the Template DOM, the result is as shown at right.
Similarly, the data has the corresponding data values directly under the “Members” data group, as shown at right.

```xml
<?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.

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 111). However this method of constructing the form has an important drawback. Suppose that the first name of "John Brown" is omitted from the data, as before:

```xml
<?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.

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 regrouping facility in the data loader. The 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 “Basic Forms” on page 35.

### Fixed Occurrence Numbers

A more concise way to represent static forms of this type is introduced in XFA 2.1. Subforms have an “occur” property which in turn has “max”, “min”, and “initial” values. By default these values 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 in XFA 2.1 as:

```xml
<template ...>
  <subform name="Members">
    <field name="Date" ...>
    <subform name="Member">
      <occur min="3" max="3" initial="3"/>
      <field name="First" ...>
      <field name="Last" ...>
    </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.
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 the “occur” property is specific to subforms and subform sets. 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.

### 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 [Config]. 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.
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:

```xml
<?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. This may be just what is desired. However, it may be desired to process records for full members only. In that case the XFA processor should be configured to recognize only “Member” data groups as record containers. The effect 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
<?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>
```

Processing instructions are never loaded into the Data DOM either in record mode or non-record mode, as specified in “Basic Forms” on page 35. 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:

```xml
<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 "Basic Forms" on page 35 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:

```xml
<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:

```xml
<?xml version="1.0"?>
<registration>
  <first>Jack</first>
  <last>Spratt</last>
  <address>
    <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, 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 that 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 cannot 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. Instead 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.

**Determination of Content Types in Data**

This section explains how to determine the content types in data. Such content types include images, rich-text, and embedded data.

**Note:** Information on this topic will be provided in a later release of this specification.
As explained in “Static Forms Versus Dynamic Forms” on page 77, 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” attributes of the “occur” property. For example, consider the same membership list described under “Forms with Repeated Fields or Subforms” on page 111, 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.

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.
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 around images. The layout process can also insert headers, footers, and bookends. See "Intermediate Layout" on page 137 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 different values 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 124, omitting decorative elements. The attributes that make it dynamic have been highlighted in bold.

```xml
<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>
```

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 “Intermediate Layout” on page 137 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 ($\text{form.Members.Member[0]}$), because this is the minimum, and binds it to the first “Member” data group ($\text{data.Members.Member[0]}$). 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 $\text{form.Members}$ 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 $\text{form.Members.Member[1]}$, binding it to $\text{data.Member.Members[1]}$, 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 $\text{template.Members.Member}$ and returns to the next higher level in the template ($\text{template.Members}$) 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.
The template for the dynamic purchase order on page 125, omitting decorative elements, is as follows:

```xml
<template ...>
  <subform name="PO">
    <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>
  <subform name="Delivery">
    <occur min="0"/>
    <field name="Del_Instrctn"...> ... </field>
  </subform>
</subform>

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.

**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 124 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.

Recall that the template was defined as follows:

```xml
<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.
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:

```
<template>
  <subform name="application">
    <subform name="sponsor">
      <occur max="7"/> <!-- up to seven sponsors -->
      <field name="lastname"> ... </field> <!-- sponsor's last name -->
      ...
    </subform>
    <field name="lastname"> ... </field> <!-- applicant's last name -->
    ...
  </subform>
</template>
```

This template is merged with the following data:

```
<application>
  <lastname>Abbot</lastname>
  ...
  <sponsor>
    <lastname>Costello</lastname>
  </sponsor>
</application>
```

The result is shown in the following figure.

Scope-matching leads to dynamic subform greedily devouring data it shouldn’t

This is not the desired result. Due to greedy matching the “sponsor” subform has been replicated and the “lastname” field within it (which is for a sponsor’s last name) is bound to the applicant’s last name. This
comes about because $data.application.lastname scope-matches to
$form.application.sponsor.lastname. 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:

```xml
<template>
  <subform name="application">
    <subform name="sponsor">
      <!-- up to seven sponsors -->
      <bind match="dataRef" ref="$data.application.sponsor"/>
      <field name="lastname"> ... </field> <!-- sponsor's last name -->
    </subform>
    <field name="lastname"> ... </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**

**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 containing an ordered subform set:

```xml
<subformSet name="topping" relation="unordered">
  <subform name="pepperoni">
    <occur initial="0"/>
    ...
  </subform>
  <subform name="greenPeppers">
    <occur initial="2"/>
    ...
  </subform>
  <subform name="olives"> ...
</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.

```
<subformSet name="topping" relation="choice">
  <subform name="pepperoni">
    <occur initial="0">...
  </subform>
  <subform name="greenPeppers">
    <occur initial="2">...
  </subform>
  <subform name="olives">...</subform>
</subformSet>
```

Now consider the same fragment modified by changing the subform set from "unordered" to "choice":

```
<subformSet name="topping" relation="choice">
  <subform name="pepperoni">
    <occur initial="0">...
  </subform>
</subformSet>
```

Empty merge of an ordered subform set
The following figure shows the result of an empty merge using this subform set. The “initial” attributes of the “occur” elements of the 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. Once the first copy of the subform set has been placed in the Form DOM, its max occurrence attribute limits how many siblings may be given to the copy. For example, consider the following template fragment:

```xml
<subformSet name="topping" relation="unordered">
  <subform name="pepperoni">
    <occur min="0">
      ...
    </occur>
  </subform>
  <subform name="greenPeppers">
    <occur min="2">
      greenPeppers[0]
      greenPeppers[1]
    </occur>
  </subform>
  <subform name="olives">
    ...
  </subform>
</subformSet>
```
The following figure shows 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”.

```
pepperoni
  relation = "unordered"
  occr.min = "0"
  greenPeppers
    occr.min = "2"
    greenPeppers
    greenPeppers
olives
```

```
pepperoni
  relation = "unordered"
  greenPeppers
  greenPeppers
olives
```

```
pizza
```

```
pepperoni
```

```
pizza
```

```
pizza
```

“min” forces inclusion of template siblings
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”.

**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”. The name of the instance manager cannot conflict with any another node because XFA node names cannot legally start with the character “_”.

"max" attribute forces exclusion of data siblings
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 127, but this time showing the instance manager.

DOMs after merge showing instance manager

For more information about using instance managers, see [Scripting].

Intermediate Layout

This section specifies how XFA processing applications place preformatted data (boilerplate) and variable data (user data) upon the page, and within a sequence of pages. The process of placing preformatted and variable data onto the page is called layout. This specification does not cover the process of associating data elements with template elements (merging), which is assumed to have already been done. Neither does this specification cover creating a stream of printer commands or API calls to display the finished pages (rendering).

XFA is 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.

The reader should be familiar with the XML 1.0 Specification [XML], with the Overview of XFA [Overview], and with the XFA Template Specification [Template]. In addition the reader should be familiar with programming concepts such as tree graphs and nodes.

Readers who are familiar with the specification XFA-Template, Version 1.0 [XFT-1.0] should take extra care: the layout scheme defined in this specification is substantially different from the one described there.
Character Notation

Character codes are given using the notation described in the preface to The Unicode Standard, Version 3.0 [Unicode-Preface], p. xxvii. Character names are as given in the Unicode character tables.

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, both of which are handled downstream in the renderer.

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 [Data Binding] 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.

Basic Concepts

Document Object Model Notation

To facilitate discussion of Document Object Models (DOMs), this specification uses a particular notation to illustrate their contents in examples, as described 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) = "value"]
where “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"
  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"]
```

**Layout Objects**

The objects upon which the layout processor operates originate in the Form DOM, which is the output from a data binding operation, and in the Template DOM. There are two general classes of layout objects.

Any layout object originating from a `<pageSet>` element (pageSet), from a pageArea element (pageArea) or from a `<contentArea>` element (contentArea) represents a physical display object or a region of a physical display object. A pageArea represents a display surface, for instance one side of a printed page. A pageSet holds an ordered collection of pageAreas, representing for example a stack of printed pages. A contentArea represents a rectangular region of a certain size at a certain position within a display surface.

All other layout objects are layout content. The purpose 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 object includes those objects which have no other function than to be visually presented upon the display, such as text, images, and geometric figures. Objects originating from `<draw>` elements are also classified as displayable because they 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 objects embody relationships between displayable entities and/or other structural layout objects. Subforms and exclusion groups are examples of structural layout objects. Note that structural layout objects 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 objects are purely passive. They are acted upon by other layout objects but have no effect upon other layout objects except by the simple act of taking up space. Physical layout objects, by contrast, are active; they both act directly upon and set constraints upon other layout objects. For example, a block of text may flow across multiple contentArea objects and be split up by them. Structural layout objects are passive for the most part, but with a few exceptions. Most of the active work of structural layout objects has already been done in the merge process.

The “w” (width) and “h” (height) attributes of layout objects are particularly likely to be a source of confusion. The height of a contentArea must be 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, up to and including version 2.1 of this specification, XFA does not 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 contentAreas to include splitting by width, and of subforms to include wrapping by height.)

The following table summarizes the types of layout objects:

<table>
<thead>
<tr>
<th>Type</th>
<th>Subtype</th>
<th>Description</th>
<th>Object Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>physical</td>
<td>-</td>
<td>physical display objects or regions thereof</td>
<td>pageSet, pageArea, contentArea</td>
</tr>
<tr>
<td>layout</td>
<td>content</td>
<td>structural logical and some physical relationships between layout objects</td>
<td>subform, subform set, area, exclusion group, field, draw</td>
</tr>
<tr>
<td>displayable</td>
<td></td>
<td>objects visibly presented upon the display</td>
<td>text, image, line, arc, rectangle, bar code, push button, checkbox, radio button, choice list, text edit widget, date edit widget, time edit widget, password edit widget, image picker widget, signature widget</td>
</tr>
</tbody>
</table>

“Layout Objects” on page 193 contains a table showing the characteristics and capabilities of each type of layout object.

**Drawing Conventions**

Most drawings in this specification use certain conventions, which are illustrated at right. The drawings employing these conventions represent 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.

Object boundaries are shown with dashed or solid black rectangles. pageSet objects and pageArea objects don’t have boundaries, at least not for layout purposes. contentArea boundaries are shown with dashed lines whereas content boundaries are shown with solid lines. None of these boundaries would be visible on the page. Some objects optionally have visible borders. Borders may coincide with the boundaries of the objects, 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 boundary.

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.

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.

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 contentAreas.

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. Similarly among sibling pageArea objects the document order determines the page order; the leftmost (eldest) sibling is the first (lowest page number) in the document.

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 must default 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. Any of these objects may be either foreground or background material. 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 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. 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:

```xml
<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>
```

After merging the Form DOM contains:

```xml
[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 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:

```xml
[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"]
```

...
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.

**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.

**Error - Template Without ContentArea**

Every template must have at least one pageArea containing a contentArea. Given a template that violates this constraint, the layout processor has nowhere to put content, so it cannot proceed. Hence this is a fatal error.

**Box Model**

Layout employs a box model. In this 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 must be reserved for a particular object. Some nominal extents are calculated at layout time and some are presupplied.

Some layout objects (subforms, fields, images, draws containing geometric figures, and contentAreas) may have nominal extents supplied explicitly via the “w” (width) and “h” (height) attributes. Each of these attributes, if supplied, must be set to a measurement. The object’s margins, if any, and its caption, if any, lie inside the nominal extent.

When a width or height is supplied to an object explicitly by “w” and/or “h” attributes, the layout processor must use the supplied width and/or height in the nominal extent, regardless of the content of the object. Therefore the nominal extent may be larger or smaller than the visual representation of the object. A nominal extent which is smaller than the visual representation of an object may lead to clipping or unwanted overlap in a subsequent rendering step, but this is no concern of the layout processor.
Some objects, including draws and subforms, may have borders. Thick borders may extend beyond the nominal extent of the object or on top of content but they do not affect layout processing. However, borders are affected by layout; the location of a border is determined by the nominal extent of the object in combination with its border inset.

The relationship between the nominal extent, borders, margins, captions, and content is shown at right.

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

**Area**

An area 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.

**Bar Code**

There are two types of bar codes, one-dimensional and two-dimensional.

For some one-dimensional bar codes, the width is fixed by the bar code 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 bar codes, the width of a symbol is adjustable.

For some one-dimensional bar codes, the height is fixed by the standard. For others, the height is adjustable.

For two-dimensional bar codes, 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 bar code standards often dictate a margin around the bar code as well. The bar code size as computed by the layout processor does not include any such mandated margin. It is up to the creator of the template to set the appropriate margin in the container. Hence, the form creator can "cheat" as desired.

**ContentArea Object**

A contentArea 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

Depending upon the value of the “aspect” property the 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.
**PageSet Object**

A pageSet represents a set of display surfaces, such as a stack of sheets of paper. Width and height do not apply.

**PageArea Object**

A pageArea represents a display surface, such as one side of a page. Its actual size is unknown (and irrelevant) to the layout processor. It is up to the creator of the template to ensure that contentArea objects and page background do not extend beyond the usable area of the surface.

**Subform**

Subforms may have explicit widths supplied by "w" attributes and explicit heights supplied by "h" attributes. When one of these attributes is not supplied the layout processor must calculate it. The width and/or height must be calculated considering the nominal extents and positions of the enclosed content and the container's own margins and caption. The overall width is the minimum width required to include all the enclosed content, plus left and right margins. The overall height is the minimum height required to include all the enclosed content, plus top and bottom margins and caption. Borders do not affect extent calculations. The figure at right shows the components of height and width for a subform with a caption and margins.
Text

Text objects may have explicit widths supplied by "w" attributes and/or explicit heights supplied by "h" attributes. When one of these attributes is not supplied, the layout processor must calculate it. The width and/or height is calculated based upon the text content. 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, plus the caption height. 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](image)

In XFA 2.1, text objects which are dynamically sized may have size limits set by "minW", "maxW", "minH" and/or "maxH" attributes. Each of these attributes, if supplied, must be set to a measurement. If any of these limits is supplied the layout processor applies it after the above calculation. If both a minimum and a maximum are specified for the same object the minimum must be smaller than or equal to the maximum. For any given text object explicit sizing and minimum/maximum sizing in the same direction must not be combined. XFA 2.0 did not include these minimum and maximum size attributes.

Widget

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. Most widgets do not have explicit size attributes. All widgets are growable; if the field or draw 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, 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.

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.

<table>
<thead>
<tr>
<th>widget</th>
<th>natural width</th>
<th>natural height</th>
</tr>
</thead>
<tbody>
<tr>
<td>button</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>check box or radio button</td>
<td>The size property plus left and right margins. Defaults to 10 points and no margins.</td>
<td>The size property plus top and bottom margins. Defaults to 10 points and no margins.</td>
</tr>
<tr>
<td>choice list</td>
<td>Same as a text edit.</td>
<td></td>
</tr>
<tr>
<td>date, time, or date-time picker</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>numeric edit</td>
<td>Same as a text edit.</td>
<td></td>
</tr>
<tr>
<td>password edit</td>
<td>Same as a text edit except each character of content is replaced with an asterisk (“*”) character.</td>
<td></td>
</tr>
<tr>
<td>signature</td>
<td>Implementation-defined.</td>
<td></td>
</tr>
<tr>
<td>text edit</td>
<td>The width of the text block plus the left and right margins. Defaults to no margins.</td>
<td>The height of the text block plus the top and bottom margins. Defaults to no margins.</td>
</tr>
</tbody>
</table>

**Error - Missing Width or Height Attribute**

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 default the value to “0”, which allows it to proceed but is likely to produce an unattractive result.

**Error - Invalid Minimum or Maximum Size**

Minimum and maximum width attributes ("minW" and "maxW") must not be combined in the same element with an explicit width ("w"). Similarly minimum and maximum height attributes ("minH" and "maxH") must not be combined in the same element with an explicit height ("h"). 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 give priority to the explicit size, ignoring the conflicting minimum and/or maximum size attributes.
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.

**Error - Invalid Measurement**

A conforming template must specify measurement units for all non-zero measurements. In addition it must supply a non-negative floating-point decimal number for each measurement. If the layout processor encounters a measurement that does not conform to these restrictions it should issue a warning. How to deal with the invalid measurement is unspecified.

**Layout Strategies**

There are two layout strategies, positioned layout and flowing layout. Some containers (pageAreas and contentAreas) 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 does not have a “layout” attribute it defaults to inheriting the layout strategy of its container, except for the root subform which defaults to positioned layout. Finally, fields always use flowing layout.

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.

**Positioned Layout**

When using positioned layout each contained object may supply an 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 attributes named “x” and “y”. The “x” attribute must be interpreted as measurement to the right and the “y” attribute as measurement down from the top-left-corner of the container. The values of these attributes must be **measurements**. If there is no “x” or “y” attribute for a contained object a value of “0” must be assumed by the layout processor. For example, assume a template identical to the first example:

```xml
<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" ...
```

Assume, as in the first example, that the merge 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, this time 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). The background text, on the other hand, is positioned relative to its own container, the pageArea itself.

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 must be 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. For example, specifying the bottom-right corner as the anchor point of a field makes the field grow up and to the left.
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. However, 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. It is the template designer's responsibility to ensure that growable objects do not grow into forbidden territory. 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.

A container using positioned layout is not considered full (that is, overflow processing is not be performed) no matter how much content has been put into it. For example, the container in the above illustration is not full.

The layout processor employs positioned layout within any area, pageArea, or contentArea. It also employs positioned layout within any draw containing an arc, line, or rectangle. And, it must employ positioned layout within any subform. Exclusion groups are transparent to layout strategy, that is, they inherit their layout strategies from their parents.

**Error - Negative Coordinates**

The "x" and "y" attributes 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 coping that doesn't mean 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 manifest in the Layout DOM. 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.

**Flowing Layout**

In flowing layout the contained objects (or in some cases upper parts of objects) must be 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” attributes, as well as its anchor point, are ignored.

The two types of flowing layout described below suffice for languages that are written left-to-right, with successive lines stacked top-to-bottom in the European style. They also handle languages written from right-to-left (e.g. Arabic). They do not suffice for languages written top-to-bottom (e.g. Chinese). It is anticipated that a future version of this specification will define additional layout procedures to handle more languages.

**Top-to-Bottom Layout**

In top-to-bottom layout the layout processor must attempt to place the first contained object at the top-left of the container. If this succeeds it must attempt 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 repeat for the remaining contained objects. If it comes to an object that will not fit in the remaining height of the container, it must attempt to split the object horizontally as discussed below in “Content Splitting”. If it is able to split the object it must place 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.

The layout processor employs top-to-bottom layout within any subform having a “layout” attribute 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 must attempt to place the first contained object at the top-left of the container. If this succeeds it must attempt to place the next contained object immediately to the right of the nominal extent of the previous contained object, or if this fails immediately below it aligned with the left edge of the container, and repeat for the remaining contained objects. If it comes to an object that will not fit in the remaining height of the container, it must attempt to split the object horizontally as discussed below in “Content Splitting”. If it is able to split the object it must place the top fragment of the split object into the container and then attempt 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.

The layout processor employs left-to-right top-to-bottom tiled layout within any subform having a "layout" attribute with a value of "lr-tb". In XFA 2.1 it also employs this layout strategy when laying out text within a draw within such a subform. See “Justification” on page 154 below for more detail about text layout.

**Right-to-Left Top-to-Bottom Tiled Layout**

In XFA 2.1 right-to-left top-to-bottom tiled layout may be used. This was not available in XFA 2.0.

In right-to-left top-to-bottom tiled layout the layout processor must attempt to place the first contained object at the top-right of the container. If this succeeds it must attempt to place the next contained object immediately to the left of the nominal extent of the previous contained object, or if this fails immediately below it aligned with the right edge of the container, and repeat for the remaining contained objects. If it comes to an object that will not fit in the remaining height of the container, it must attempt to split the object horizontally as discussed below in “Content Splitting”. If it is able to split the object it must place the top fragment of the split object into the container and then attempt 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. These are the same layout objects and container used in the example for left-to-right top-to-bottom tiled layout, above.

The layout processor employs right-to-left top-to-bottom tiled layout within any subform having a “layout” attribute with a value of “rl-tb”. It also employs this layout strategy when laying out text within a draw within such a subform. See “Justification” on page 154 below for more detail about text layout.

**Error - 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 (because text has no natural flow direction)
Flowing layout applied to a geometric figure inside a draw that does not assert “w” and “h” properties (because 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 must be fixed)

Justification

Text can be automatically justified or aligned. This section defines justification and alignment methods for languages with lines stacked top-to-bottom in the European manner. It is anticipated that future versions of this specification will be extended to cover additional languages.

Whether it originates from user data or from boilerplate, 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 the XFA Data Handling Specification [Data]. Each record consists of a stream of characters, formatting markup, and/or embedded objects. Formatting markup modifies the presentation of the characters it encloses, for example by setting the type size or type face. Formatting markup is described in the XFA Data Text Handling Specification [Rich-Text]. Embedded objects are displayable objects which flow with the surrounding text, as though each one was a character except that embedded objects are not affected by formatting markup. Examples of embedded objects include bar codes and images.

In XFA 2.0 text layout had to flow from left to right. In XFA 2.1 layout from right to left is also permitted. The direction of flow is determined by the “layout” attribute of the container within which the text is being placed. The “layout” attribute must be “lr-tb” to specify left-to-right and “rl-tb” to specify right-to-left.

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.

```
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 layout processor treats text differently depending upon whether it is being placed into a region with a fixed width or one that is growable in width.

**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. The layout processor responds to the newline by starting a new line, then places subsequent characters on the new line. The final width of the region is equal to the width of the longest line. The figure at right shows the extent of the sample paragraph.

**Text Layout with Fixed Width**

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 a paragraph the layout processor treats the text as a string of text layout units. A text layout unit may be an embedded object or it may be a word. The boundaries of layout units may be delimited by the edges of embedded objects or by white space characters. “White space” is defined as any combination of characters with the code points U0020, U2000 through U200B inclusive, or U3000. These code points are the ones defined as breaking spaces in chapter 6 of the Unicode Standard, version 3.0 [Unicode-Ch6].
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).

Text layout units within a paragraph are placed in document order in the flow direction on each line. If the “hAlign” attribute is set to “left” the text layout units are used to tile the region in the same manner as a tiled layout fills a container, except that any white space characters which are supplied between text layout units are also incorporated, unless they fall at the end of a line. If “hAlign” is set to “right” the same procedure is followed except that the whole line is shifted right so that the rightmost text layout unit abuts the right edge of the region. If “hAlign” is set to “center” the same procedure is followed except that the whole 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”.

If “hAlign” is set to “justifyAll” then 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 so that the line fills the region from left to right. If “hAlign” is set to “justify” then all lines are justified except the last one in its paragraph. The following example shows justification with the “hAlign” attribute set to “justifyAll”. In this case the template contained the following declarations:

```xml
<draw ...
  <text>There are 4 spaces after this word but only 2 after this word.</text>
</draw>
```

The resulting layout is shown at right.

If “hAlign” is set to “radix”, then the text is treated as a column of numbers, one per line. The numbers are assumed to be expressed in the Western style using digits “0” through “9”. 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 content 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. Text flow direction has no effect because numbers expressed in the Western style always flow in the same direction, regardless of the language in which they are embedded.

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 object by itself is too wide to fit into the region it must be 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.

### Text Layout in the Vertical Direction

In the vertical direction the region may not all be useable, because some portion may be reserved for a caption. Vertical alignment within the useable region is controlled by the “vAlign” attribute. If “vAlign” is
set to “top” the first line of text must be positioned at the top of the useable 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 must be positioned at the bottom of the useable 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 must be positioned to the middle of the useable region. Note that there is no “vAlign” attribute for subforms. (A future version of this specification may permit the use of “vAlign” with subforms.)

In the at right, the field 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.

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.

Deferral and splitting are characteristics of a flowing layout strategy. However, it should be noted that 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 were described above in Clipping.

Deferral can be controlled via the “break” 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 for more information. Deferral can also be constrained by a requirement that adjacent subforms be kept together. See Adherence 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.

Content Splitting

Splitting is not trivial. Splitting within margins is not allowed (because it would defeat the purpose of declaring a margin.) A simple multiline block of text cannot split at just any height, because most split lines would cut through a line 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.

**Bar code**

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.

To be or not to be: that is the question. Whether 'tis nobler in the mind to bear the slings and arrows of outrageous fortune, or take up arms against a sea of troubles

**Widget**

Widgets include buttons, check boxes, radio buttons, choice lists, and signature objects. 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 must not split the subform. Instead the subform must be 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 (see Tables), 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:

```
<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](image_url)
Split Consensus

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.

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 must find the most efficient (lowest) split location that is acceptable to all of the contained objects. In other words it must find the optimum consensus. The optimum consensus may be found by the following procedure:

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,

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.

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.

By default, when the current pageArea is full, the layout processor moves on to the next pageArea object in document order. 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.

Maximum Occurrence Limits

When the layout processor finishes filling the last contentArea on a page, 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:

```xml
<pageSet 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 B. 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)]
    [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 pageAreas to descend into, so it must consider adding another instance of pageSet A. This has a maximum occurrence attribute of -1. A maximum occurrence attribute 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])]
    [pageArea (B)]
      [contentArea (C)]
      [contentArea (D)]
    [pageArea (E[0])]
      [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 pageAreas to descend into, so it must consider adding another instance of pageSet A. This has a maximum occurrence attribute of -1. A maximum occurrence attribute 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])]
    [pageArea (B)]
      [contentArea (C)]
      [contentArea (D)]
    [pageArea (E[0])]
      [contentArea (F)]
```
Assuming that the last content is used up without filling the 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 `pageSets` 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 XFA Data Binding Specification [Merge] for more information about minimum and maximum occurrence values for subforms.

**Error - Exhaustion of pageArea Occurrences**

If all available `pageSets` and `pageAreas` contain `<occur>` elements having “max” attributes that are not equal to “-1”, there is a limit to how many `pageAreas` can be included in the layout. When the last `pageArea` within this limit has been laid out, the layout processor must stop processing. If there is more content that has not yet been laid out, the additional content must be 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.

**Minimum Occurrence Limits**

Minimum occurrence attributes on `pageSets` and `pageAreas` force the layout processor 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 attribute is “0”. When the minimum-occurrence attribute 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:

```xml
<pageSet name="A">
  <occur min="2"/>
  <pageArea name="B">
    ... content ...
  </pageArea>
</pageSet>
```

... content ...

```
[pageArea (E[1])]
[contentArea (F)]
... content ...
```

```
[pageSet (A[1])]
[pageArea (B)]
[contentArea (C)]
... content ...
```
<occur min="0"/>
<contentArea name="C" ... />
<contentArea name="D" ... />
</pageArea>
<pageArea name="E">
<occur min="2"/>
<contentArea name="F" ... />
</pageArea>
</pageSet>

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 attribute 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])]
    [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 pageAreas, 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 contentAreas, and more contentAreas are added, the pageAreas are gradually consumed. No more pageAreas or pageSets are added until the existing ones are used up. At that point the Layout DOM contains:
The layout processor is data-driven; it lays down pageAreas in order to use the contentAreas 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 XFA Data Binding Specification [Merge] for more information about minimum and maximum occurrence values for subforms.

**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. These sub-properties accept three values. “none” means the subform does not adhere to the adjacent subform. “contentArea” means the adjacent parts of the two subforms must be in the same content region. “pageArea” means the adjacent parts of the two subforms must be 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.

```xml
<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>
</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.

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:
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.

**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.

For example, in the following template subform D does not adhere either to subform C or subform X because they are not siblings.
<template ...
  <subform name="root" layout="tb" ...>
  <pageSet ...
  ...
  </pageSet>
  <subform name="C" ... />
  <subform name="X" ...
    <subform name="D" ...
      <keep prev="contentArea" />
      ...
    </subform>
    ...
  </subform>
  </subformSet>
</subform>
</template>

Note that putting a subform into the template inside a subformSet makes the subformSet its parent. Thus
the subform can only adhere to other children of the same subformSet. This is the only direct effect a
subformSet can exert upon the layout process. For example, in the following template subform D does not
adhere to subform C because they are not siblings. However subform E does adhere to D.

<template ...
  <subform name="root" layout="tb" ...
  <pageSet ...
  ...
  </pageSet>
  <subform name="C" ... />
  <subformSet name="X" ...
    <subform name="D" ...
      <keep prev="contentArea" />
      ...
    </subform>
    <subform name="E" ...
      <keep prev="contentArea" />
      ...
    </subform>
    </subformSet>
  <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.
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 five “inner” subforms adheres together, but there is no adherence between groups of “inner” subforms or between “inner” and “outer” subforms.

**Break Conditions**

By default, as explained above in Flowing Between contentArea Objects, the layout processor moves on to a new contentArea if and when the current contentArea overflows (which can only happen with flowing layout). However when placing a subform it can be forced to move to a different pageArea and/or contentArea by the “break” properties of the object being placed. This can happen even when positioned layout is being used.

The <break> element supplies these properties. The <break> element can force the layout processor to go to a particular contentArea or a particular pageArea. In either case, an attribute of the <break> element provides a target-ID. If there is an object of the correct type matching the target-ID the layout processor traverses the pageArea subtree of the Template DOM, taking the shortest route to get from the node
corresponding to the current contentArea to the node with the specified target-ID. (This traversal may have side-effects which are discussed below under “Leaders and Trailers”.) 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.

**Break on Entry**

A subform may specify default behavior explicitly via a <break> element with a “before” 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 an object has a “before” property with a value of “pageArea” or “contentArea”, the layout processor gets the target-ID from the value of the object's “beforeTarget” attribute. If there is an object of the correct type matching the target-ID 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:

```
<subform name="X">
  <break before="pageArea" beforeTarget="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>
```

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 pageAreas and contentAreas 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 - Invalid beforeTarget ID**

A conforming template must not specify a target ID for the beforeTarget attribute of a <break> element that does not match an ID declared in the template. 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> element having an attribute of “startNew” with a value of “1” and an attribute of “before” 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:

```xml
<subform "W">
    <pageSet name="A">
        <pageArea name="B">
            <contentArea name="C" ... />
        </pageArea>
    </pageSet>
    <subform name="X">
        <break before="pageArea" startNew="1"/>
        <field name="Y"/>
    </subform>
</subform>
```

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 <break> element may have “before”, “startNew”, and “beforeTarget” attributes. This combination requires the layout processor to fulfill both the startNew and beforeTarget conditions, that is, the subform must be placed in a pageArea or contentArea that is empty and also corresponds to a template object with the specified ID.

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 with a particular target-ID. This is done by specifying an attribute of “overflowTarget” attribute with the target-ID as the value. If overflow occurs and there is an object of the correct type matching the target-ID, the layout processor breaks to the specified target.

Note that there is no such thing as an “overflow” attribute. One might expect that one would be required to specify “contentArea” or “pageArea”, but it is not necessary because “overflowTarget” supplies an object ID and XML requires that each object ID be unique within the document. By contrast “startNew” can be used without “beforeTarget”, so “before” is needed to tell the layout processor what level to operate at.

For example, a template contains the following declarations:

```xml
<subform name="X">
  <break overflowTarget="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:
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.

**Combining Overflow with Other `<break>` Attributes**

The “overflowTarget” attributes is processed at a different time and under different circumstances than the other break conditions. Consequently “overflowTarget” may be freely combined with any of the other attributes.

**Error - Invalid overflowTarget ID**

A conforming template must not specify a target ID for the overflowTarget attribute of a `<break>` element that does not match an ID declared in the template. 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.

**Combined Break Condition and Occurrence Limits**

A template may combine a subform asserting break conditions with contentAreas and/or pageAreas asserting occurrence limits. The layout processor must simultaneously satisfy 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:

```xml
<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" before="pageArea" beforeTarget="B_ID">
    <field name="S"> ... </field>
  </subform>
</subform>
```
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 “beforeTarget” 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:
<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" before="pageArea" beforeTarget="D_ID">
    <field name="S"> ... </field>
  </subform>
</subform>

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"]
[pageArea (D[1])]
  [contentArea (D)]
    [subform (O)]
    [subform (R)]
      [field (S) = "text in field S"]
[pageArea (D[2])]

Subform P does not fill contentArea C. However, the next subform to be laid out is R, which asserts a “before” break 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.

Leaders and Trailers

A subform may be associated with leaders and/or trailers that must be 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.

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. For example, a template includes the following declarations:
<template>
  <subform name="W">
    <pageSet ...
      <pageArea ...
        <contentArea name="A" ID="A_ID" />
      </pageArea>
    </pageSet>
    <subform name="B" layout="tb">
      <break
        before="contentArea"
        beforeTarget="A_ID"
        bookendLeader="Leader_ID"
        bookendTrailer="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.

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 must be 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 must be placed as the first content in the bottom fragment of the subform. For example, a template includes the following declarations:
<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" ...
      <break
        overflowLeader="Leader_ID"
        overflowTrailer="Trailer_ID" />
      <field name="C" ... </field>
      <field name="D" ... </field>
      <field name="E" ... </field>
    </subform>
  </subform>
</template>

<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.

In the example above E 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 E, 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.
When a field overflows the overflow leader and trailer must be supplied by the field's containing subform, because a field does not have a break 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:

```xml
<template>
  <subform name="W">
    <pageSet ...
      <pageArea ...>
        <contentArea name="A" ... />
        <contentArea name="B" ... />
        <contentArea name="C" ... />
        <contentArea name="D" ... />
      </pageArea>
    </pageSet>
  </subform>
  <subform name="E" layout="tb" ...>
    <break overflowLeader="Leader_ID" overflowTrailer="Trailer_ID"/>
    <field name="F">
      ...
    </field>
  </subform>
</template>
```
Assume that field F is very tall compared to the contentAreas. 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.

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.

**Leader/Trailer Lists**

Both overflowLeader and overflowTrailer attributes may have values which are space-separated lists of target-IDs. The separator must be a single SPACE (U0020) character. Each target-ID is (re)used as required until its maximum occurrence limit is reached, after which the layout processor goes on to the next target-ID in the list. A target-ID may appear in both lists; each use from either list counts towards its occurrence limit. It is pointless to put a target-ID 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:
<template>
  <subform name="W">
    <pageSet ...
      <pageArea ...
        <contentArea name="A" ... />
        <contentArea name="B" ... />
        <contentArea name="C" ... />
        <contentArea name="D" ... />
      </pageArea>
    </pageSet>
  </subform>
  <subform name="E" layout="tb" ...
    <break overflowTrailer="X_ID Y_ID X_ID"/>
    <field name="F"...> ... </field>
  </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.

**Error – Invalid Leader/Trailer Target**

*Note: This information will be supplied in a later release.*

**Inheritance of Overflow Leaders and Trailers**

When a subform does not specify a leader or trailer, it inherits the 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:

```xml
<template>
  <subform "W" ...>
    <pageSet ...>
      <pageArea ...>
        <contentArea name="A" ... />
        <contentArea name="B" ... />
        <contentArea name="C" ... />
        <contentArea name="D" ... />
      </pageArea>
    </pageSet>
    <subform name="E" layout="tb" ...>
      <break
        begin="contentArea"
        beginTarget="A"
        overflowTrailer="X_ID"/>
      <field name="F" ... > ... </field>
      <subform name="G" layout="tb" ...>
        <break overflowTrailer="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.

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 a leader or trailer once it has exhausted the occurrence limit(s) for its own 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 both types may be combined in the same context. For example, a template includes the following declarations:

<template>
  <subform name="W">
    <pageSet …>
      <pageArea …>
        <contentArea name="A" … />
        <contentArea name="B" … />
        <contentArea name="C" … />
        <contentArea name="D" … />
      </pageArea>
    </pageSet>
  </subform>
  <subform name="E" layout="tb" …>
    <break
      bookendLeader="Title_ID"
      bookendTrailer="Source_ID"
      overflowLeader="X_ID"
      overflowTrailer="Y_ID"/>
    <field name="F" …> … </field>
  </subform>
</template>

<proto>
  <subform name="Title" ID="Title_ID">
    <field …> … </field>
  </subform>
  <subform name="Source" ID="Source_ID">
    <field …> … </field>
  </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.

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 attribute. 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 is most often found inside a table subform, but if not it is treated as a one-row table by itself.

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”.

Combined bookend and overflow leaders and trailers
The following example shows the structure of a table in the template:

```xml
<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. It lays out the rows sequentially from top to bottom. 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. 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.

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:

```xml
<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”.

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, it is not possible to make a cell span 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 must not be descended from a table subform, hence tables can not be nested.

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.

Page Background

A pageArea may contain subforms, fields, draws and areas. 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.

In the following example, a template contains the following declarations:

```xml
<subform name="X">
  <pageArea name="A">
    <draw …>
      <image href="Preliminary.jpg" … />    
    </draw>
    <contentArea name="B" ID="A_ID" … />
  </pageArea>
  <subform name="C" …>
    <break before="contentArea" targetID="A_ID"/>
    <field …> … </field>
  </subform>
</subform>
```
The resulting layout incorporates the “Preliminary.jpg” image as background in each instance of pageArea A, as shown at right.

The layout processor must place background ahead of any other content of the same pageArea within the Layout DOM, as explained above in Layout Within a Container. 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.

In XFA 2.1, structural layout objects used in background content may use either positioned or flowing layout. This is a change from XFA 2.0 which restricted background to positioned layout.

Note that when page background contains a field, the field is not bound to data by the Data Binding process. Instead the layout processor must bind the field to data itself. For this reason each field object in page background must have a dataRef property that states explicitly what data object it binds to.

Error – Pure Boilerplate With Minimum Occurrence of -1

A pure boilerplate pageArea is a pageArea that does not contain any contentAreas. A pure boilerplate pageArea must not contain an <occur> element with a “max” attribute having a value of “-1”. This is because, should the layout processor find its way into that pageArea, it logically would have to incorporate an infinite number of instances of the pageArea into the Layout DOM before it could proceed to another pageArea. Hence templates are forbidden to do this. For the same reason, a pageSet that contains only pure boilerplate pageAreas must not contain an <occur> element with a “max” attribute having a value of “-1”. 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.

Error – Page Background Field With No dataRef

If a field in page background has no dataRef it can not be bound to any data.
## Layout Objects

The following table lists the characteristics of all the different layout objects.

<table>
<thead>
<tr>
<th></th>
<th>characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>arc</strong></td>
<td>boilerplate geometric figure</td>
</tr>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
<tr>
<td></td>
<td><strong>area</strong></td>
</tr>
<tr>
<td></td>
<td>physical and logical grouping of objects</td>
</tr>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
<tr>
<td></td>
<td><strong>bar code</strong></td>
</tr>
<tr>
<td></td>
<td>machine-readable data</td>
</tr>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>button</td>
<td>clickable region</td>
</tr>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
<tr>
<td>checkButton</td>
<td>check box or radio button</td>
</tr>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
<tr>
<td>contentArea</td>
<td>physical region of a display surface</td>
</tr>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
<tr>
<td>draw containing image</td>
<td>static displayable (boilerplate) image</td>
</tr>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
<tr>
<td>Object Type</td>
<td>Display Type</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>draw containing geometric figure</td>
<td>static displayable (boilerplate)</td>
</tr>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
<tr>
<td>draw containing text</td>
<td>static displayable (boilerplate)</td>
</tr>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
<tr>
<td>embedded object</td>
<td>non-text displayable entity embedded</td>
</tr>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
<tr>
<td>exclGroup</td>
<td>logical grouping of fields (one-of-many)</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>glyph</th>
<th>printable symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>field containing button</th>
<th>clickable widget</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>field containing check-box</th>
<th>clickable widget</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
<tr>
<td>Field Containing</td>
<td>One Item of Variable Data</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>Date, Time, or Date-Time</td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
<tr>
<td>Field Containing Image</td>
<td>variable image</td>
</tr>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
<tr>
<td>Field Containing Number</td>
<td>variable numeric data</td>
</tr>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
<tr>
<td>Field Containing Password</td>
<td>variable character data</td>
</tr>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
<tr>
<td>Field Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>Field containing radio button</td>
<td>Clickable widget</td>
</tr>
<tr>
<td></td>
<td>Layout strategy</td>
</tr>
<tr>
<td></td>
<td>Break control?</td>
</tr>
<tr>
<td></td>
<td>Natural size?</td>
</tr>
<tr>
<td></td>
<td>Growable?</td>
</tr>
<tr>
<td></td>
<td>Splittable?</td>
</tr>
<tr>
<td></td>
<td>Multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>Container for</td>
</tr>
<tr>
<td>Field containing signature</td>
<td>Clickable widget</td>
</tr>
<tr>
<td></td>
<td>Layout strategy</td>
</tr>
<tr>
<td></td>
<td>Break control?</td>
</tr>
<tr>
<td></td>
<td>Natural size?</td>
</tr>
<tr>
<td></td>
<td>Growable?</td>
</tr>
<tr>
<td></td>
<td>Splittable?</td>
</tr>
<tr>
<td></td>
<td>Multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>Container for</td>
</tr>
<tr>
<td>Field containing text</td>
<td>Variable character data</td>
</tr>
<tr>
<td></td>
<td>Layout strategy</td>
</tr>
<tr>
<td></td>
<td>Break control?</td>
</tr>
<tr>
<td></td>
<td>Natural size?</td>
</tr>
<tr>
<td></td>
<td>Growable?</td>
</tr>
<tr>
<td></td>
<td>Splittable?</td>
</tr>
<tr>
<td></td>
<td>Multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>Container for</td>
</tr>
<tr>
<td>Image</td>
<td>Bitmapped image</td>
</tr>
<tr>
<td></td>
<td>Layout strategy</td>
</tr>
<tr>
<td></td>
<td>Break control?</td>
</tr>
<tr>
<td></td>
<td>Natural size?</td>
</tr>
<tr>
<td></td>
<td>Growable?</td>
</tr>
<tr>
<td></td>
<td>Splittable?</td>
</tr>
<tr>
<td></td>
<td>Multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>Container for</td>
</tr>
<tr>
<td>line</td>
<td>boilerplate geometric figure</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
<tr>
<td>pageArea</td>
<td>display surface, such as one side of a sheet of paper</td>
</tr>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
<tr>
<td>pageSet</td>
<td>collection of display surfaces</td>
</tr>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
<tr>
<td>rectangle</td>
<td>boilerplate geometric figure</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
<tr>
<td>subform</td>
<td>logical grouping of fields and boilerplate</td>
</tr>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
<tr>
<td>subformSet</td>
<td>logical grouping of subforms</td>
</tr>
<tr>
<td></td>
<td>layout strategy</td>
</tr>
<tr>
<td></td>
<td>break control?</td>
</tr>
<tr>
<td></td>
<td>natural size?</td>
</tr>
<tr>
<td></td>
<td>growable?</td>
</tr>
<tr>
<td></td>
<td>splittable?</td>
</tr>
<tr>
<td></td>
<td>multiply-occurring?</td>
</tr>
<tr>
<td></td>
<td>container for</td>
</tr>
<tr>
<td>text</td>
<td>sequence of glyphs and/or embedded objects</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>layout strategy</td>
<td></td>
</tr>
<tr>
<td>break control?</td>
<td></td>
</tr>
<tr>
<td>natural size?</td>
<td></td>
</tr>
<tr>
<td>growable?</td>
<td></td>
</tr>
<tr>
<td>splittable?</td>
<td></td>
</tr>
<tr>
<td>multiply-occurring?</td>
<td></td>
</tr>
<tr>
<td>layout container for</td>
<td></td>
</tr>
</tbody>
</table>
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 tasks performed by automation objects are described in "Scripting" on page 227 and "Web Service Integration" on page 229.

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, a user clicking a field, or another automation event activating.

When an automation object is invoked, it performs some task particular to the type of object. Examples of such tasks include executing a script, submitting part/all of the form using HTTP Post, or executing a web services file.

About

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 [XFA-FormCalc], 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 [XFA-SOM] 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 three elements that allow a form template to specify a system of values for the form:

- Statically set an initial value — non-empty content in a value element
- Derive a value dynamically, likely from other form values or ambient properties such as date and time — 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:

```xml
<template name="ScriptExample">
  <subform name="ScriptExample">
    <area x="0" y="0"> ... </area>
    <area x="0" y="2">
      <field name="Item" x="0" y="0" w="3" h="12pt"> ... </field>
      <field name="Quantity" x="3.5" y="0" w="1" h="12pt">
        <validate>
          <script>within($, 0, 19)</script>
        </validate>
      </field>
      <field name="UnitPrice" x="5" y="0" w="1" h="12pt"> ... </field>
      <field name="Amount" x="6.5" y="0" w="1" h="12pt">
        <calculate>
          <script>Quantity * UnitPrice</script>
        </calculate>
      </field>
      <field name="Item" x="0" y="0.5" w="3" h="12pt"> ... </field>
      <field name="Quantity" x="3.5" y="0.5" w="1" h="12pt">
        <validate>
          <script>within($, 0, 19)</script>
        </validate>
      </field>
      <field name="UnitPrice" x="5" y="0.5" w="1" h="12pt"> ... </field>
      <field name="Amount" x="6.5" y="0.5" w="1" h="12pt">
        <calculate>
          <script>Quantity * UnitPrice</script>
        </calculate>
      </field>
      <field name="Item" x="0" y="1" w="3" h="12pt"> ... </field>
      <field name="Quantity" x="3.5" y="1" w="1" h="12pt">
        <validate>
          <script>within($, 0, 19)</script>
        </validate>
      </field>
      <field name="UnitPrice" x="5" y="1" w="1" h="12pt"> ... </field>
      <field name="Amount" x="6.5" y="1" w="1" h="12pt">
        <calculate>
          <script>Quantity * UnitPrice</script>
        </calculate>
      </field>
      <field name="Item" x="0" y="1.5" w="3" h="12pt"> ... </field>
      <field name="Quantity" x="3.5" y="1.5" w="1" h="12pt">
        <validate>
          <script>within($, 0, 19)</script>
        </validate>
      </field>
      <field name="UnitPrice" x="5" y="1.5" w="1" h="12pt"> ... </field>
      <field name="Amount" x="6.5" y="1.5" w="1" h="12pt">
        <calculate>
          <script>Quantity * UnitPrice</script>
        </calculate>
      </field>
    </area>
  </subform>
</template>
```
Calculation

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 "XFA Template Element Reference" 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 `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.

`calculate` elements and some types of `event` elements are similar; however, they differ in their underlying purpose:

- `calculate` elements provide a means of updating the value of a container whenever the values on which they depend change. Values that appear in a calculate script may themselves be the result of a calculate element being activated.
- `event` elements provide a means of handling events generated by the processing application.
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 Calculations, Validations, and Events Activated by the Same Trigger” on page 223.)

The stimuli that cause the processing application to activate calculate elements are described below:

- **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.

  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 227.)

  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.

Result

The processing application uses the result of executing the calculation script as described below:

<table>
<thead>
<tr>
<th>Parent element</th>
<th>Result destination in the parent element</th>
</tr>
</thead>
<tbody>
<tr>
<td>field</td>
<td>Replaces the value of the field container element.</td>
</tr>
<tr>
<td>exclGroup</td>
<td>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.</td>
</tr>
<tr>
<td>subform</td>
<td>Not applicable because subform elements do not have value elements. subform script elements can be used to initiate some other function unrelated to setting a value.</td>
</tr>
</tbody>
</table>
Validation

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. Validation tests are triggered by the same stimuli that trigger calculations and events.

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.

<table>
<thead>
<tr>
<th>Execution order</th>
<th>Test type</th>
<th>Container element</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Field</td>
</tr>
<tr>
<td>1</td>
<td>Null-content test. 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.</td>
<td>✔</td>
</tr>
<tr>
<td>2</td>
<td>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.</td>
<td>✔</td>
</tr>
<tr>
<td>3</td>
<td>Format test. The format of the value must match a validation picture clause.</td>
<td>✔</td>
</tr>
<tr>
<td>4</td>
<td>Script test. The script supplied in the validate element must return a true value for a script test to succeed.</td>
<td>✔</td>
</tr>
</tbody>
</table>

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 210.

As described in the XFA Picture Clause Specification [XFA-Picture-Clause], 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 Calculations, Validations, and Events Activated by the Same Trigger” on page 223.

**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 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.

```
Security Risk [ ] Loan amount [ ]
```
<field name="LoanAmount" y="31.3252mm" x="70.38mm" w="72.3505mm" h="13.7137mm">
  ...
  <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 loan you more than $1000.
      </text>
    </message>
    <picture>zzzzz9</picture>
    <script>
      (not SecurityRisk) | (SecurityRisk &amp; (LoanAmount &lt; 1000))
    </script>
  </validate>
</field>

<field name="SecurityRisk" y="26.46mm" 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. The validate element’s attributes specify how the XFA processing application should respond in the event of an error. For example:

<validate formatTest="error" scriptTest="error">

In the case of the datatype test, the error response is implied, as described later in this section.

The following sections describe the error response levels for each type of test.
nullTest

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.

Datatype test

This test is implied by the existence of a datatype element (i.e. `integer` and `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.

formatTest

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.

scriptTest

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.

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 script associated with the event is automatically invoked. Thus an XFA template can contain sophisticated logic that transforms the data, the presentation of the data, or even the template itself, in response to circumstances.

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.
A script binds to an application event by expressing a `ref` property whose value is `xfa.host`, or the alias `$host`.

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 ($form) it fires after the Template and Data DOMs have been merged to create the Form DOM, and the calculations and validations have fired. In addition, the ready event fires when the current data record advances. See the Data Handling Specification [@@Data-Handling] for more information about processing data as records.
In the case of the Layout DOM ($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:

```xml
<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 217.

**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 dropdown 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.

exit
This event triggers when the field loses keyboard focus.

full
This event triggers when the user has entered the maximum allowed amount of content into the field.

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.

**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:

```xml
<subform name="root">
  <subform name="sub1">
    <field name="field1" ...>
      <event ref="$" activity="click">
        <execute connection="service1" ...>
        </execute>
      </event>
    </field>
    <field name="field2" ...>
      <event ref="$" activity="click">
        <execute connection="service1" ...>
        </execute>
      </event>
    </field>
  </subform>
  <event ref="$connectionSet.service1" activity="postExecute">
    <script>...</script>
  </event>
</subform>
```

The accompanying connection set is as follows:

```xml
<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.

**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:

<table>
<thead>
<tr>
<th>Target Type</th>
<th>Event Name</th>
<th>Sets Primary Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>subform</td>
<td>enter</td>
<td>name, target</td>
</tr>
<tr>
<td></td>
<td>exit</td>
<td></td>
</tr>
<tr>
<td>field</td>
<td>click</td>
<td>name, modifier, shift,</td>
</tr>
<tr>
<td></td>
<td>enter</td>
<td>target</td>
</tr>
<tr>
<td></td>
<td>mouseEnter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mouseExit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mouseDown</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mouseUp</td>
<td></td>
</tr>
<tr>
<td>field</td>
<td>exit</td>
<td>commitKey, name, modifier, shift, target</td>
</tr>
<tr>
<td>field</td>
<td>change</td>
<td>change, keyDown, fullText, modifier, name, newContentType, newT, prevContentType, prevText, selEnd, selStart, shift, start, target</td>
</tr>
</tbody>
</table>
In addition the following secondary events set secondary properties but don’t change any primary properties:

<table>
<thead>
<tr>
<th>Target Type</th>
<th>Event Name</th>
<th>Sets Secondary Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>connection</td>
<td>preExecute</td>
<td>soapFaultCode</td>
</tr>
<tr>
<td></td>
<td>postExecute</td>
<td>soapFaultString</td>
</tr>
</tbody>
</table>

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.

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.

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
```

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:

```
<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
Forms" on page 35.

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 Forms"
on page 35.
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.

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 for Calculations, Validations, and Events 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.

```xml
<subform name="outer">
  <subform name="sub1">
    <event name="ready" ref="$form">
      <script>...</script>
    </event>
  </subform>
  <subform name="sub2">
    <event name="ready" ref="$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 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, validation precedes the exit event
2. validations and exit events for nested elements occur in order from inner to outer element
3. validation and 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, given the following template fragment:

```xml
<subform name="outer">
  <subform name="X">
    <validate ... />
    <event name="exit" ref="$" ... />
    <field name="A">
      <validate ... />
      <event name="exit" ref="$" ... />
    </field>
  </subform>
  <subform name="Y">
    <validate ... />
    <subform name="enter" ref="$" ... />
    <field name="B">
      <validate ... />
      <event name="enter" ref="$" ... />
    </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, given the following template fragment:

```xml
<field name="A">
  <event name="full" ref="$" ... />
  <event name="change" ref="$" ... />
</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 occur in order of depth-first traversal of the Form DOM
- 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: events causing events**

A script may cause changes of state that in turn trigger 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):

```xml
<subform name="root">
  <event name="ready" ref="$form">
    <script contentType="application/x-ecmascript">
      ...
      $form.root.A.enter();
      ...
    </script>
  </event>
  <field name="A">
    <event name="enter" ref="$" />
  </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 order of execution of the field enter event script and the portion of the form ready script after this line is undefined.

The following template fragment illustrates a script performing an action (changing keyboard focus) which indirectly triggers another event.

```xml
<subform name="root">
  <event name="ready" ref="$form">
    <script contentType="application/x-ecmascript">
      ...
      $host.setFocus($form.root.A);
      ...
    </script>
  </event>
</subform>
```
<field name="A">
  <event name="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). 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.
This chapter describes the role of scripting objects in templates. It describes how scripting languages 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 [XFA-FormCalc]. Often, the scripts attached to a form are similar to those attached to a spreadsheet. 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) [XFA-SOM]. A key feature of XFA-SOM is that it manages relative references. For example, when defining an invoice detail line in the Template Designer, the user might set up fields unitPrice, quantity and amount. The calculation for amount would simply be unitPrice*quantity. XFA-SOM would manage the scope in two ways: find the correct instances of unitPrice and quantity if the instances of those field names are in other subforms; and find the correct instances of unitPrice and quantity, 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 (e.g., saving the data).

Finally, scripts can be assigned to user actions, for example, onEnter, onExit, onClick, and so on.

**Script Language Selection**

*Note:* Information on this topic will be provided in a later release of this specification.

**Selecting a Scripting Environment**

*Note:* Information on this topic will be provided in a later release of this specification.
Setting up the scripting environment

**Note:** Information on this topic will be provided in a later release of this specification.

Exception handling

**Note:** More complete information on this topic will be provided in a later release of this specification.

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.

FormCalc and ECMAScript respond to exceptions as follows:

- **FormCalc.** The script stops as soon as an exception is thrown. FormCalc exceptions are described in [XFA-FormCalc](#).
- **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 [@@reference].

Picture clauses

**Note:** Information on this topic will be provided in a later release of this specification. In the mean time, please refer to the previous version of this reference Adobe XML Architecture: Picture Clause Specification, version 2.0. You can find that document using the search option on [http://partners.adobe.com/asn/](http://partners.adobe.com/asn/).

Naked references in JavaScript

**Note:** Information on this topic will be provided in a later release of this specification.

Writing Scripts for calculate Elements

**Note:** Information on this topic will be provided in a later release of this specification.

Calculation scripts must not make alterations to the structure of any of the DOMs, such as adding, deleting, or moving nodes.
9 Web Service Integration

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 simpler Web-interactive form, event objects can cause part or all of the XFA form to be submitted to a server. 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.

This chapter contains the following sections:

- “Submission” on page 229
- “Data Description and Connection Set” on page 231
- “Schema and WSDL” on page 254
- “Null handling” on page 254

Submission

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, and packaging of that content.

Submitting events 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.

Submitting events provide a uniform method used for the following purposes:

- 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

This chapter defines one other mechanism for submitting data, which are events having an execute action. Such event actions use web services to submit and process data. See “Schema and WSDL” on page 254 for a description of data submission using web services.

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.

```
<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.MyURL.com"
      textEncoding="UTF-16"
      xdpContent="pdf datasets xfdf template"/>
  </event>
</field>
```

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 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 and how content should be packaged (`format`, `embedPDF` and `xdpContent`) and by specifying how the content should be encoded (`textEncoding`).

The following sections discuss how the `submit` syntax supports various form-oriented workflows.

#### Workflows That Use XFA Forms Without PDF Content

**Note:** To be supplied in a subsequent release of this specification.

#### Workflows That Use XFA-Based PDF Forms

In situations where the data portion of the XDP must be consistent XFA-based PDFs, PDF content is submitted using the XDP format rather than the current XFDF grammar for form data.

The `submit` syntax supports content interchange by allowing submission of the PDF document and its subassemblies (annotations and data) and formatting that content formatted as XDP or XML. An example of a submit object that supports such a workflow follows. Only the PDF form data is sent to the server.

```
<submit
  format="xdp"
  target="http://www.MyURL.com"
  textEncoding="UTF-16"
  xdpContent="xfdf"/>
```

The target value may be any valid URI. The following shows examples of such URIs.

```
target="mailto:john.doe@company.com"
target="ftp://ftp.is.co.za/rfc/rfc1808.txt"
target="file:///C:/dir/filename"
```
Processing Steps Initiated by Activation of a Submitting Event

When an event having a submit action is activated, the processing application performs the following steps.

1. Executes all validation scripts. If this step raises exceptions, the requested content should not be submitted to the server.

2. Executes all nullTest checks. If this step raises exceptions, the requested content should not be submitted to the server.

3. Executes 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 DOM changes state.

5. Executes all run-at-client preSubmit events. The scripts in preSubmit events that are triggered when the submitting event is activated 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. (See also @events.)

6. Package the data and any other content, as specified in the submit properties format, embedPDF and xdpContent.

7. Send the data to a server using an HTTP POST operation. If submission triggered any run-at-server preSubmit events, the content is submitted to the server with an indication that the server should run the associated script before performing the rest of the processing.

   The response to an HTTP POST failure is application dependent.

Data Description and Connection Set

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. Also, it is a design goal for XFA to accept existing data instance documents without alteration, but not all XML documents contain schema declarations. Therefore, XFA provides a mechanism to specify a single optional default data description and any number of optional context-specific data descriptions, via a mechanism outside of the data document.

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 232 and “Data Description Element Reference” on page 235 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 “XFA Template Element Reference” on page 257 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 Grammar” on page 240 and “Connection Set Element Reference” on page 241 sections of this specification. Web Services are described in “Web Services” on page 244. Submission via HTTP is described in @@ref.

Data Description Grammar

Data descriptions are contained in an XDP inside the dataSets packet. 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:

```xml
<po:order xmlns:po="http://www.abc/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:

```xml
<dd:dataDescription
  xmlns:dd="http://ns.adobe.com/data-description/"
  name="order">
```
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):

```xml
<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>
```

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:

```xml
<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:order>
```
The components of the dd: markup are described in detail in the section "Data Description Element Reference" on page 235.

**Description of a Web Service Message**

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:

```xml
<dd:dataDescription
    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":

```xml
<dd:dataDescription
    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.
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.

**dd:name attribute**

This attribute supplies a name for the data description. The attribute must be supplied and the name 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.

The value of this attribute must match a subform name, so it must be a valid XFA name as defined in @@ref.

**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:

```xml
<location>
  <x>1.234</x>
  <y>5.678</y>
</location>
```

But the following is also valid:

```xml
<location>
  <r>5.432</r>
  <theta>31.97</theta>
</location>
```

This can be declared in a data description as follows:

```xml
<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:

```xml
<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 "Basic Forms" on page 35.

**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 value 0 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 to 0.

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.

```xml
<po:customer>
  <po:lastname/>
  <po:firstname dd:minOccur="0"/>
</po:customer>
```

Hence the following fragment of data is valid:

```xml
<po:customer>
  <po:lastname>Smith</po:lastname>
</po:customer>
```

The following fragment declares that a meeting must be attended by at least two people:

```xml
<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 than 1. 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:

```xml
<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:

```xml
<pizza dd:model="choice">
  <houseSpecial/>
  <topping dd:maxOccur="-1">
    <pizza/>
  </topping>
</pizza>
```

Hence the following fragment of data is valid:

```xml
<pizza>
  <houseSpecial/>
</pizza>
```

But the following is also valid:

```xml
<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.

```xml
<address dd:model="ordered">
  <streetNumber/>
  <streetName/>
  <city/>
  <postalCode/>
</address>
```

Hence the following fragment of data is valid:

```xml
<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.

```xml
<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.

```xml
<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.

```xml
<animal name="" vet:species=""
   dd:reqattrs="name vet:species"/>
```

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 \texttt{wsdlConnection} element. Invoking the connection causes the web service to be invoked using the sequence of operations listed under "Web Services" on page 244.

The following skeleton summarizes the structure of the connection set element and its descendants:

```xml
<connectionSet
xmlns="http://www.xfa.org/schema/xfa-connection-set/2.1/">
  <!-- 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 either this... -->
  <xmlConnection dataDescription="ddName" name="cxnName">
    <uri>sampleDataURI</uri>
  </xmlConnection>
  <!-- ...or this... -->
  <xsdConnection dataDescription="ddName" name="cxnName">
    <rootElement>elementName</rootElement>
    <uri>schemaURI</uri>
  </xsdConnection>
</connectionSet>
```

The corresponding \texttt{[XMLSchema]} schema is available at \texttt{@@ref Connection Set Element Reference}

### Connection Set Element Reference

All of the elements and attributes described in this section must belong to the following namespace:

http://www.xfa.org/schema/xfa-connection-set/2.1/

\textbf{Note:} The trailing "/" is required.

The "2.1" represents this version of this specification. 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 "2.1" it will know that it may have to make fixups to bring the connection set up to date with the later specification.

\texttt{connectionSet Element}

This element is the container for the set of connections. There must be at most one \texttt{connectionSet} element in an XDP.

\texttt{content}

Any number of \texttt{wsdlConnection} elements and at most one of either \texttt{xmlConnection} or \texttt{xsdConnection} elements.
**operation Element**

This element declares the SOAP operation that is associated with its enclosing `wsdlConnection` element. 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.

**input Attribute**

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.

**output Attribute**

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.

**content**

The name of the selected operation.

**rootElement Element**

This element declares the starting point (root) within the associated W3C [XMLSchema] schema.

**content**

The name of the outermost element that was used when generating the data description from the associated W3C [XMLSchema] schema.

**soapAction Element**

This element declares the SOAP action for its parent `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].

**soapAddress Element**

This element declares the host location for its parent `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].

**uri Element**

This element declares the location of the sample document or schema for its parent `xmlConnection` or `xsdConnection`.

**content**

The URI for the sample document or schema.
wsdlAddress Element
This element identifies the location of the service description to which the enclosing wsdlConnection element and its contents correspond.

ccontent
The URI for the service description.

wsdlConnection Element
This element represents one connection to a web service. This connection corresponds to a particular action requested from a particular service with data going in a particular direction or directions.

dataDescription Attribute
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.

name Attribute
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. See “XFA Template Element Reference” on page 257.

ccontent
An operation element, a soapAction element, a soapAddress element, and optionally a wsdlAddress element.

xmlConnection Element
This element represents a connection to sample data. 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.

dataDescription Attribute
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.

name Attribute
The name of the connection. The name must be unique among connections.

ccontent
A uri element.

xsdConnection Element
This element represents a connection to a schema. 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.
**dataDescription Attribute**

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.

**name Attribute**

The name of the connection. The name must be unique among connections.

**content**

A uri element.

---

**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 2.1 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 descriptions and connection sets before proceeding with this section.

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.

It is important to clarify the terminology “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.

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.

In a web service transaction the cycle of operation is:

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 then 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.
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]. The web service description file for the service follows:

```xml
<?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"/>
    </operation>
  </binding>
</definitions>
```
This definition file tells potential clients how to access the service. The types element defines XML components for use in the other sections. The two message elements define the two messages that are used. The portType element defines the operations and how they use messages. The binding element defines how the messages bind to the SOAP protocol (GetLastTradePriceInput as the input message and GetLastTradePriceOutput as the output message). The service element defines the URL of the server and the name of the service (port). For more information about the meaning of WSDL definition files see [WSDL1.1].

The input message defined by the above WSDL definition carries a ticker symbol for a publicly-listed corporation. The message 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>
```

If the query succeeds (that is, if a share quotation can be obtained for the given ticker symbol), the output message 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>
```

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>
```

```xml
<soap:Fault>
  <faultcode>...<faultcode>
  <faultstring>...<faultstring>
</soap:Fault>
```
The soap:Fault element can also contain a faultactor element but this is uncommon.

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. However the simple service in this example supports only one operation, so in this case only wsdlConnection element is needed. The complete connection set packet (with white space added for clarity) follows:

```xml
<connectionSet
  xmlns="http://www.xfa.org/schema/xfa-connection-set/2.1/"
>
  <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 two attributes which link it to other parts of the XDP. The dataDescription attribute points to a data description, which must be created. The name attribute supplies a name for this connection which will be used in connect elements to indicate that the content of a field, exclusion group, or subform takes part in the transaction.

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 soap Action 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. In the example the data description packet is:

```xml
<dd:dataDescription
 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.

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”). Fields and exclusion groups can have connect children which 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:

```xml
<subform name="input" ...>
 <subform name="query">
   <field name="stockTickerSymbol" ...>
   </field>
   <connect
    connection="TradePriceWS"
    ref="!connectionData.TickerPriceWS.Body.TradePriceRequest.tickerSymbol"
    usage="exportOnly"/>
   ...
 </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 identifying 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. Body 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:

```xml
<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” @@ref). 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.

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:

```
<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 Mapping Rules” @ref.

After the preExecute event finishes, the input message in the Connection Data DOM is converted to XML and sent to the web server. The 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:

```
<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:

```
<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 $event object, 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:

```xml
<subform name="output">
  <field name="sharePrice">
    ...
    <connect
      connection="TickerPriceWS"
      ref="!connectionData.TickerPriceWS.Body.TradePrice.price"
      usage="importOnly"/>
  </field>
  ...
</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. For example,

```xml
<field name="getQuoteBtn">
  ...
  <ui>
    <button />
  </ui>
  <event activity="click" runAt="client">
    <execute
      connection="TickerPriceWS"
      executeType="import"/>
  </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**

*Note:* Information on this topic will be provided in a later release of this specification.

**Null handling**

*Note:* Information on this topic will be provided in a later release of this specification.
10 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.

Detailed description of all widgets

Note: Information on this topic will be provided in a later release of this specification.

Sequencing (tabbing orders)

Note: Information on this topic will be provided in a later release of this specification.

Accessibility

Note: Information on this topic will be provided in a later release of this specification.
11 Transformations

XSLT

Note: Information on this topic will be provided in a later release of this specification.

Built-in XFA-Data DOM Transformations

Note: Information on this topic will be provided in a later release of this specification.
This chapter is the language reference for the XFA template syntax. The syntax descriptions begin on page 263.

Guide to the Template Element Reference

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

The Template Syntax Reference 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 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:

```xml
<Draw>
  <Fill>
    <Color Value="255,255,255">
      <Solid/>
    </Color>
    <Value>
      <Rectangle>
        <Corner Join="Round"/>
        <Edge Stroke="Solid"/>
        <Edge Stroke="Dashed"/>
      </Rectangle>
    </Value>
  </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 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:

```xml
<Draw>
  <Fill>
    <!-- A white color fill -->
    <Color Value="255,255,255">
      <Solid/>
    </Color>
    <Value>
      <Rectangle>
        <Corner Join="Round"/>
        <Edge Stroke="Solid"/>
        <Edge Stroke="Dashed"/>
      </Rectangle>
    </Value>
  </Fill>
  <Fill>
    <!-- A black color fill -->
    <Color Value="0,0,0">
      <Solid/>
    </Color>
  </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:

```xml
<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:

```xml
<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.
Element Reference

The arc element

A curve that can be used for describing either an arc or an ellipse.

```xml
<arc

Properties:
  circular="0 | 1"
  hand="even | left | right"
  id="xml-id"
  lock="0 | 1"
  startAngle="0 | angle"
  sweepAngle="360 | angle"
  use="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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
Allow changes to properties and content.

1
Block changes to properties and content.

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 a prototype. The value of this property is a '#' character followed by the prototype’s identifier.
The area element

A container representing a geographical grouping of other containers.

<area

Properties:
- colSpan="1 | integer"
- id="xml-id"
- lock="0 | 1"
- name="xml-id"
- relevant="cdata"
- use="cdata"
- x="0in | measurement"
- y="0in | measurement"

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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.

The name property
An identifier that may be used to identify this element in script expressions.

The relevant property
Controls which views of the form include the enclosing object.

@@tbd - The interpretation of this attribute is not yet determined: see ViewsXFALangProposal and XFALangMinutes20030731.

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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.

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.

```xml
<assist

Properties:
    lock="0 | 1"
>
    <speak> [0..1]
    <toolTip> [0..1]
</assist>
```

The assist element is used within the following other elements:

- draw
- exclGroup
- field
- proto

The assist element provides a means to specify the tool tip and behavior for a spoken prompt.

Note that the assist element appears as a property of both the field and draw elements. It is a property of the latter, because an interactive application may choose to improve a form’s accessibility by speaking the static content of the form.

**The lock property**

A permission flag for allowing or blocking attempted changes to the element.

- 0
  
  Allow changes to properties and content.

- 1
  
  Block changes to properties and content.

**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 barcode element

An element that represents a bar code.

<barcode

Properties:

- `checksum` (none | auto | 1mod10 | 2mod10 | 1mod10_1mod11)
- `dataColumnCount` (cdata)
- `dataLength` (cdata)
- `dataPrep` (none | flatCompress)
- `dataRowCount` (cdata)
- `endChar` (cdata)
- `errorCorrectionLevel` (0 | cdata)
- `id` (xml-id)
- `lock` (0 | 1)
- `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)
- `wideNarrowRatio` (3:1 | cdata)

>  

The barcode element is used within the following other elements:

proto ui

The barcode element supplies the information required to display a bar code. This includes the type of the bar code and a set of options which varies from one type of bar code to another.

There are many types of bar codes. An XFA application is not required to support any particular set of bar codes. Futhermore the set of supported bar codes may vary depending on the display device, because some printers have built-in support for particular bar codes. This is known as a hardware bar code. A bar code may also be drawn stroke by stroke by the XFA application itself; this is known as a software bar code. When displaying on glass, which is not accessible to bar code readers, an XFA application may also revert to displaying just a placeholder rather than an accurate bar code.

Bar codes are complicated. For each type of bar code there are usually two separate specifications, one for the bar code itself and one for the bar code’s placement in relation to the physical page and to surrounding printed matter. The creator of the template is responsible for ensuring that the bar code is placed correctly upon the page. The XFA application is responsible for correctly rendering the bar code using the user data. The user data must be compatible with the bar code, that is, it must conform to the allowed character set and string length.
The checksum property

Algorithm for the checksum to insert into the bar code. The checksum is calculated based upon the supplied bar code data. The template schema allows any one of the choices listed below. Some bar code formats, however, either require a particular checksum or never allow a checksum; for such bar codes the XFA processor must ignore this property. Some of the remaining bar code formats support only a limited subset of these choices; for such bar codes the template must not specify a non-supported choice.

none

Do not insert a checksum. This is the default.

auto

Insert the default checksum for the bar code 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 bar code standards documents for the bar codes to which they apply.

The dataColumnCount property

(2-d bar codes only.) Optional number of data columns to encode for supported bar codes. The template must supply this property in conjunction with dataRowCount to specify a fixed row and column bar code, otherwise the XFA processor must use rowColumnRatio to determine the row and column count. 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 bar code it is padded out with padding symbols.

The dataLength property

(1-d bar codes only.) The expected maximum number of characters for this instance of the bar code.

For software bar codes, 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 bar code 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 bar codes this parameter is ignored. Because the XFA processor does not know the details of the bar code format, it cannot use this information to determine the bar width.

The data being displayed is not validated against this parameter. For software bar codes the XFA processor allows the data to overflow the assigned region of the field. For hardware bar codes the result of an overflow depends upon the printer.

Note that there is no corresponding minimum length restriction. Some bar code formats have a fixed number of symbols and must be filled out with padding characters. Others allow a variable number of symbols and must terminate after the last symbol.
The dataPrep property
{error: no property description found for barcode.dataPrep}

The dataRowCount property
(2-d bar codes only.) Optional number of data rows to encode for supported bar codes.

The template may supply this property in conjunction with dataColumnCount in order to specify a fixed row and column bar code. Otherwise the XFA processor uses rowColumnRatio plus the actual length of the data being inserted to determine the row and column count. The dataRowCount property must not be present unless the dataColumnCount property is also present. When these properties are used the size of the bar code is fixed. If the supplied data does not fill the bar code the remaining cells are padded out with padding symbols.

The endChar property
Optional ending control character to append to bar code data. This property is ignored by the XFA processor if the bar code pattern does not allow it.

The errorCorrectionLevel property
(2-d bar codes only.) Optional error correction level to apply to supported bar codes. For PDF417 the valid values are integers in the range 0 through 8, inclusive.

For bar code 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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
    Allow changes to properties and content.

1
    Block changes to properties and content.

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 bar code 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 bar code. 1-D bar codes 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 bar codes 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 bar code 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 bar codes moduleWidth either has no effect or has the same effect as for a software bar code, depending upon the printer and bar code. However for hardware bar codes 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 bar codes.

For 2-d bar codes 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 bar code 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 bar codes.

**The printCheckDigit property**

Specifies whether the check digit(s) is/are printed in the human-readable text.

The XFA processor ignores this property if the checksum property has a value of no, or if the checksum property has a value of yes and the standard behavior for the bar code type is to not include a checksum.

0  
Do not print the check digit in the human-readable text, only in the bar code itself. This is the default.

1  
Append the check digit to the end of the human-readable text.

**The rowColumnRatio property**

(2-d bar codes only.) Optional ratio of rows to columns for supported 2-D bar codes.

The XFA processor ignores this property if dataRowCount and dataColumnCount are specified.

When rowColumnRatio is supplied the XFA processor allows the bar code 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 bar code data.

This property is ignored by the XFA processor if the bar code 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 bar code. This is the default.

**above**
Text is placed above the bar code.

**belowEmbedded**
Text is partially embedded in the bottom of the bar code. The baseline of the text is aligned with the bottom of the bars.

**aboveEmbedded**
Text is partially embedded at the top of the bar code. The top of the text is aligned with the top of the bars.

**none**
No text is displayed.

The region available for embedded text, if any, is determined by the bar code format. For most bar code formats it is a single contiguous region, but for EAN series bar codes 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 template specifies `belowEmbedded` and there is no embedded text region at the bottom of the bar code, 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 bar code, the XFA processor may interpret the property as `above`.

The `truncate` property
Truncates the right edge of the bar code for supported formats. Of the bar codes in the standard types list, this applies only to PDF417. The XFA processor ignores this property for bar code 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 bar code 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 bar code 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

code128A

code128B

code128C

code128SSCC

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 bar code 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 bar codes. The bar code(s) are marked so that the bar code reader knows when it still has additional bar codes 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 bar code (Zip+4), as defined in DMM C840 Barcoding Standards for Letters and Flats [USPS-C840].

postUSZip

United States Postal Service POSTNET bar code (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


ucc128random


ucc128sscc


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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.

The wideNarrowRatio property
Ratio of wide bar to narrow bar in supported bar codes.

The allowable range of ratios varies between bar code formats and also, for hardware bar codes, the output device. The template must not specify a value outside the allowable range. The XFA processor ignores this property for bar code 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.
The bind element

An element that controls the behavior during merge operations of its enclosing element.

```xml
<bind

Properties:
  lock="0 | 1"
  match="once | none | global | dataRef"
  ref="cdata"
>
  <picture> [0..1]
</bind>
```

The bind element is used within the following other elements:
`exclGroupfield proto subform`

The lock property

A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.

The match property

Controls the role played by the 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 the XFA-Data Binding Specification [XFA-Data-Binding] for more information about, and an authoritative definition of, the effects of this property.

The picture property

A rendering element that describes input mask 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 boolean element

A content element describing single unit of data content representing a Boolean logical value.

```xml
<boolean

Properties:
  id="xml-id"
  lock="0 | 1"
  name="xml-id"
  use="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 bind.nullType property.

**The id property**

A unique identifier that may be used to identify this element as a target.

**The lock property**

A permission flag for allowing or blocking attempted changes to the element.

0

Allow changes to properties and content.

1

Block changes to properties and content.

**The name property**

An identifier that may be used to identify this element in script expressions.

**The use property**

Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The border element

A box model element that describes the border surrounding an object.

```xml
<border

Properties:
  break="close | open"
  hand="even | left | right"
  id="xml-id"
  lock="0 | 1"
  presence="visible | invisible | hidden"
  relevant="cdata"
  use="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:
`checkButtonchoiceListdateTimeEditdraw 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**

An element that describes the constraints on moving to a new page or content area 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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
Allow changes to properties and content.

1
Block changes to properties and content.

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
Controls which views of the form include the enclosing object.

@@tbd - The interpretation of this attribute is not yet determined: see ViewsXFALangProposal and XFALangMinutes20030731.

The use property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The break element

An element that describes the constraints on moving to a new page or content area after rendering an object.

<break

Properties:
- after="auto | contentArea | pageArea | pageBack |
  pageEven | pageFront | pageOdd"
- afterTarget="cdata"
- before="auto | contentArea | pageArea | pageBack |
  pageEven | pageFront | pageOdd"
- beforeTarget="cdata"
- bookendLeader="cdata"
- bookendTrailer="cdata"
- id="xml-id"
- lock="0 | 1"
- overflowLeader="cdata"
- overflowTarget="cdata"
- overflowTrailer="cdata"
- startNew="0 | 1"
- use="cdata"

>  
  <extras> [0..1]

</break>

The break element is used within the following other elements: proto subform subformSet

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.

currentArea

Rendering will transition the next available content area.

pageArea

Rendering will transition to a new page.

pageBack

When duplexing, rendering will transition to the next available back surface, potentially causing an intervening page surface to be printed. If duplexing is not in effect, rendering will transition to a new page.
**pageEven**
Rendering will transition to the next available even numbered page, potentially causing intervening numbered and/or unnumbered page(s) to be printed. This behavior does not require duplexing.

**pageFront**
When duplexing, rendering will transition to the next available front surface, potentially causing an intervening page surface to be printed. If duplexing is not in effect, rendering will transition to a new page.

**pageOdd**
Rendering will transition to the next available odd numbered page, potentially causing intervening numbered and/or unnumbered page(s) to be printed. This behavior does not require duplexing.

**The afterTarget property**
Specifies the explicit destination page or content area for the after property.

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.

**pageBack**
When duplexing, rendering will transition to the next available back surface, potentially causing an intervening page surface to be printed. If duplexing is not in effect, rendering will transition to a new page. Note that pageBack, unlike pageEven, is not affected by page numbering.

**pageEven**
Rendering will transition to the next available even numbered page, potentially causing intervening numbered and/or unnumbered page(s) to be printed. This behavior does not require duplexing.

**pageFront**
When duplexing, rendering will transition to the next available front surface, potentially causing an intervening page surface to be printed. If duplexing is not in effect, rendering will transition to a new page. Note that `pageFront`, unlike `pageOdd`, is not affected by page numbering.

`pageOdd`
Rendering will transition to the next available odd numbered page, potentially causing intervening numbered and/or unnumbered page(s) to be printed. This behavior does not require duplexing.

**The beforeTarget property**
This property specifies the explicit destination page or contentArea for the before property. 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. 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. 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 lock property**
A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.

**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 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 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 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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The button element

A user interface element that describes a push-button widget.

```xml
<button
    id="xml-id"
    lock="0 | 1"
    use="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 id property**
A unique identifier that may be used to identify this element as a target.

**The lock property**
A permission flag for allowing or blocking attempted changes to the element.

0
    Allow changes to properties and content.

1
    Block changes to properties and content.

**The use property**
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The calculate element

An automation element that controls the calculation of its container's value.

<calculate

Properties:

  id="xml:id"
  lock="0 | 1"
  override="disabled | warning | error | ignore"
  use="cdata"
>
  <extras> [0..1]
  <message> [0..1]
  <script> [0..1]
</calculate>

The calculate element is used within the following other elements:

  exclGroupfield 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 lock property

A permission flag for allowing or blocking attempted changes to the element.

  0  
    Allow changes to properties and content.

  1  
    Block changes to properties and content.

The message property

A automation element that holds one or more sub-elements containing validation failure messages.

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.

When there is no accompanying calculate element this property has no effect and the user is free to enter a value into the container. Otherwise the behavior is as described below.
disabled
   The calculation is disabled. In an interactive context the user is free to enter data into the field.

error
   The calculation is enabled. The user is not allowed to override the calculated value.

ignore
   The calculation is enabled. The user is free to override the calculated value. The calculated value is merely a default.

warning
   The calculation is enabled. The user can override the calculated value, but when the container gains input focus the application presents a warning message and prompt. The warning message indicates that the calculated value is recommended. The prompt gives the user two choices: leave the value alone or go ahead and alter it. Once the user has opted to go ahead the application does not issue any warning or prompt 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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The caption element

A box model element that describes a descriptive label associated with an object.

<caption

Properties:
    id="xml:id"
    lock="0 | 1"
    placement="left | top | right | bottom | inline"
    presence="visible | invisible | hidden"
    reserve="-1 | measurement"
    use="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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.

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.
There is no meaningful default for this attribute. Rather, if this attribute is not supplied the height or width (as appropriate) is determined by the content of the caption and text auto-wrapping does not occur.

**The use property**
Invokes a prototype. The value of this property is a '#' character followed by the prototype’s identifier.

**The value property**
A content element that encloses a single unit of data content.

For more information see "The value element".
The certificate element

{error: missing description for certificate element}

<certificate

Properties:
   id="xml:id"
   lock="0 | 1"
   name="xml:id"
   use="cdata"
>
   ...pcdata...
</certificate>

The certificate element is used within the following other elements:
issuers proto signing

Content
{error: no property description found for certificate.#text}

The id property
A unique identifier that may be used to identify this element as a target.

The lock property
A permission flag for allowing or blocking attempted changes to the element.

0
   Allow changes to properties and content.

1
   Block changes to properties and content.

The name property
An identifier that may be used to identify this element in script expressions.

The use property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The certificates element

{error: missing description for certificates element}

<certificates

Properties:
   id="xml:id"
   lock="0 | 1"
   url="cdata"
   use="cdata"
>
   <issuers> [0..1]
   <oids> [0..1]
   <signing> [0..1]
</certificates

The certificates 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 issuers property
{error: no property description found for certificates.issuers}
For more information see "The issuers element".

The lock property
A permission flag for allowing or blocking attempted changes to the element.

0
   Allow changes to properties and content.

1
   Block changes to properties and content.

The oids property
{error: no property description found for certificates.oids}
For more information see "The oids element".

The signing property
{error: no property description found for certificates.signing}
For more information see "The signing element".

The url property
{error: no property description found for certificates.url}
The use property

Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The checkButton element

A user interface element that describes either a checkbox or radio-button widget.

```xml
<checkButton

Properties:
  allowNeutral="0 | 1"
  id="xml-id"
  lock="0 | 1"
  shape="square | round"
  size="10pt | measurement"
  use="cdata"
>
  <border> [0..1]
  <extras> [0..1]
  <margin> [0..1]
</checkButton>
```

The checkButton element is used within the following other elements:
proto ui

**The allowNeutral property**

This property specifies whether the checkbox or radio-button can support an additional third-state that represents a neutral value.

0

The checkbox or radio-button supports two states representing true or false.

1

The checkbox or radio-button supports three states representing true, false, or neutral.

**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 lock property**

A permission flag for allowing or blocking attempted changes to the element.

0
Allow changes to properties and content.

1

Block changes to properties and content.

**The margin property**

A box model element that specifies one or more insets for an object.

For more information see "The margin element".

**The shape property**

This property specifies whether the checkbox or radio-button will appear, with a square or round outline, respectively.

- **square**
  - The check button appears with a square outline.

- **round**
  - The check button appears with a round outline.

**The size property**

A measurement specifying the size of the checkbox or radio-button outline representing either the height/width for a checkbox or the diameter for a radio-button. The default is 10pt.

**The use property**

Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
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"
- lock="0 | 1"
- open="userControl | onEntry | always | multiSelect"
- textEntry="0 | 1"
- use="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

{error: no property description found for choiceList.commitOn}

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 lock property

A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.
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.

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.

The use property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The color element

An element that describes a color.

<color

Properties:
  cSpace="SRGB" | cdata
  id="xml-id"
  lock="0 | 1"
  use="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 lock property
A permission flag for allowing or blocking attempted changes to the element.

  0
  Allow changes to properties and content.

  1
  Block changes to properties and content.

The use property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.

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

{error: missing description for comb element}

<comb

Properties:
  id="xml:id"
  lock="0 | 1"
  use="cdata"
>
</comb>

The comb element is used within the following other elements:
proto textEdit

**The id property**
A unique identifier that may be used to identify this element as a target.

**The lock property**
A permission flag for allowing or blocking attempted changes to the element.

0
   Allow changes to properties and content.

1
   Block changes to properties and content.

**The use property**
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
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 version 2.1 [XFA-Connection-Set] for more information.

```
<connect
  Properties:
    connection="cdata"
    lock="0 | 1"
    ref="cdata"
    usage="exportAndImport | exportOnly | importOnly"
>
  <picture> [0..1]
</connect>
```

The connect element is used within the following other elements:
exclGroupfield proto subform

**The connection property**
The name of the associated connection element in the connection set.

**The lock property**
A permission flag for allowing or blocking attempted changes to the element.

0  
  Allow changes to properties and content.

1  
  Block changes to properties and content.

**The picture property**
A rendering element that describes input mask 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 reffor the same connection. For example if a subform has aref 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 contentArea element

An element that describes a region within a page area eligible for receiving content.

```
<contentArea

Properties:
  h="0in | measurement"
  id="xml-id"
  lock="0 | 1"
  name="xml-id"
  relevant="cdata"
  use="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: pageAreaproto

**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. When height is specified as a measurement, that value 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.

**The id property**

A unique identifier that may be used to identify this element as a target.

**The lock property**

A permission flag for allowing or blocking attempted changes to the element.

0

  Allow changes to properties and content.

1
Block changes to properties and content.

**The name property**
An identifier that may be used to identify this element in script expressions.

**The relevant property**
Controls which views of the form include the enclosing object.

@@tbd - The interpretation of this attribute is not yet determined: see ViewsXFALangProposal and XFALangMinutes20030731.

**The use property**
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.

**The w property**
Width for layout purposes. When width is specified as a measurement, that value 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.

**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"
  lock="0 | 1"
  presence="visible | invisible | hidden"
  radius="0in | measurement"
  stroke="solid | dashed | dotted | lowered | raised | etched | embossed | dashDot | dashDotDot"
  thickness="0.5pt | measurement"
  use="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).
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 lock property**

A permission flag for allowing or blocking attempted changes to the element.

0

Allow changes to properties and content.

1

Block changes to properties and 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 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 if 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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The date element

A content element that describes a single unit of data content representing a date.

```xml
<date
Properties:
   id="xml-id"
   lock="0 | 1"
   name="xml-id"
   use="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 [ISO8601], as specified in [XFA-Date-Time]. 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 [ISO8601] as specified in XFA Date and Time Data Formats[XFA-Date-Time].

When no content is present, the content shall be interpreted as representing a null value, irrespective of the value of the `bind.nullType` property.

**The id property**

A unique identifier that may be used to identify this element as a target.

**The lock property**

A permission flag for allowing or blocking attempted changes to the element.

0

Allow changes to properties and content.

1

Block changes to properties and content.

**The name property**

An identifier that may be used to identify this element in script expressions.

**The use property**

Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
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"
  lock="0 | 1"
  name="xml-id"
  use="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 [ISO8601] as specified in XFA Date and Time Data Formats [XFA-Date-Time].

When no content is present, the content shall be interpreted as representing a null value, irrespective of the value of the bind.nullType property.

The id property

A unique identifier that may be used to identify this element as a target.

The lock property

A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The dateTimeEdit element

A user interface element describing a widget intended to aid in the selection of date and/or time.

<dateTimeEdit
  Properties:
    id="xml-id"
    lock="0 | 1"
    use="cdata"
  >
    <border> [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 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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.

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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The decimal element

A content element that describes a single unit of data content representing a number with a fixed number of digits after the decimal.

<decimal

Properties:
- fracDigits="\text{integer}"
- id="xml-id"
- leadDigits="\text{integer}"
- lock="0 | 1"
- name="xml-id"
- use="cdata"

> ...pcdata...
</decimal>

The decimal element is used within the following other elements: desc exObject extras items proto value variables

Content

This element may enclose decimal-data which is 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 bind.nullType property.

The fracDigits property

The maximum number of digits (inclusively) following the decimal point to capture and store. The default is 2.

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 is 0.

The lock property

A permission flag for allowing or blocking attempted changes to the element.

0

Allow changes to properties and content.

1
Block changes to properties and content.

**The name property**
An identifier that may be used to identify this element in script expressions.

**The use property**
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The `defaultUi` element

An element for widgets whose depiction is delegated to the XFA application.

```xml
<defaultUi

Properties:
  id="xml-id"
  lock="0 | 1"
  use="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 lock property**

A permission flag for allowing or blocking attempted changes to the element.

- 0
  - Allow changes to properties and content.

- 1
  - Block changes to properties and content.

**The use property**

Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The desc element

An element to hold human-readable metadata.

```xml
<desc

Properties:
  id="xml:id"
  lock="0 | 1"
  use="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 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 lock property**
A permission flag for allowing or blocking attempted changes to the element.

0
   Allow changes to properties and content.

1
   Block changes to properties and content.

**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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The draw element

A container element that contains non-interactive data content.

```xml
<draw

Properties:
  anchorType="topLeft | topCenter | topRight | middleLeft |  
  middleCenter | middleRight | bottomLeft | bottomCenter |  
  bottomRight"
  colSpan="1 | integer"  
  h="0in | measurement"  
  hAlign="left | center | right | justify |  
  justifyAll | radix"
  id="xml-id"
  lock="0 | 1"  
  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"
  vAlign="top | middle | bottom"
  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]
</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. When height is specified as a measurement, that value 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.

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 assume that the last character represents 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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
   Allow changes to properties and content.

1
   Block changes to properties and content.

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. If this attribute is not supplied there is no limit.
If an h attribute is supplied the container is not vertically growable and this attribute is ignored.

The maxW property
Maximum width for layout purposes. If this attribute is not supplied there is no limit.
If a w attribute is supplied the container is not horizontally growable and this attribute is ignored.

The minH property
Minimum height for layout purposes. The default is 0.
If an h attribute is supplied the container is not vertically growable and this attribute is ignored.

The minW property
Minimum width for layout purposes. The default is 0.
If a w attribute is supplied the container is not horizontally growable and this attribute is ignored.

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
Controls which views of the form include the enclosing object.

@@tbd - The interpretation of this attribute is not yet determined: see ViewsXFALangProposal and XFALangMinutes20030731.

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 non-negative multiple of 90. The default is 0.

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 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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.

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 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. When width is specified as a *measurement*, that value 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.

**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 PositionedLayout. 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 PositionedLayout. 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"
lock="0 | 1"
presence="visible | invisible | hidden"
stroke="solid | dashed | dotted | lowered | raised | etched | embossed | dashDot | dashDotDot"
thickness="0.5pt | measurement"
use="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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
   Allow changes to properties and content.

1
   Block changes to properties and 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 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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The event element

An automation element that causes a script to be executed or data to be submitted whenever a particular event occurs.

```xml
<event
  activity="click | initialize | enter | exit |
          mouseEnter | mouseExit | change | preSave | postSave | prePrint | postPrint | ready |
          docReady | docClose | mouseUp | mouseDown |
          full | preSubmit | preExecute | postExecute | preOpen"
  id="xml-id"
  lock="0 | 1"
  name="xml-id"
  ref="\$ | cdata"
  use="cdata"
>
  <extras> [0..1]
</event>
```

One-of properties:

- `<execute>` [0..1]
- `<script>` [0..1]
- `<signData>` [0..1]
- `<submit>` [0..1]

The event element is used within the following other elements:

- `exclGroupfield`
- `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`
  @@tbd
docReady
@@tbd

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.

initialize
@@tbd

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 connectionDataelement 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.

**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 lock property**

A permission flag for allowing or blocking attempted changes to the element.

0

Allow changes to properties and content.

1

Block changes to properties and content.

**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 @@tbd 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**
{error: no property description found for event.signData}

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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The exclGroup element

A container element that describes a mutual exclusion relationship between a set of containers.

```xml
<exclGroup

Properties:
  access="open | protected | readOnly | nonInteractive"
  anchorType="topLeft | topCenter | topRight | middleLeft |
  middleCenter | middleRight | bottomLeft | bottomCenter |
  bottomRight"
  colSpan="1 | integer"
  h="0in | measurement"
  hAlign="left | center | right | justify | justifyAll | radix"
  id="xml-id"
  layout="position | lr-tb | rl-tb | tb |
  table | row"
  lock="0 | 1"
  maxH="0in | measurement"
  maxW="0in | measurement"
  minH="0in | measurement"
  minW="0in | measurement"
  name="xml-id"
  presence="visible | invisible | hidden"
  relevant="cdata"
  transient="0 | 1"
  use="cdata"
  vAlign="top | middle | bottom"
  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]
</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 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 anchorType property**
Location of the container's anchor point when placed with positioned layout strategy.

- **topLeft**
  Top left 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 it's 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 version 2.1 [XFA-Connection-Set] 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. When height is specified as a measurement, that value 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.

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 assume that the last character represents 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 layout property**
Layout strategy to be used within this element.

position
  The content of the element is positioned according 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**
  Reserved for future use. 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 lock property**
A permission flag for allowing or blocking attempted changes to the element.
Allow changes to properties and content.

1

Block changes to properties and content.

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. If this attribute is not supplied there is no limit. If an h attribute is supplied the container is not vertically growable and this attribute is ignored.

The maxW property
Maximum width for layout purposes. If this attribute is not supplied there is no limit. If a w attribute is supplied the container is not horizontally growable and this attribute is ignored.

The minH property
Minimum height for layout purposes. The default is 0. If an h attribute is supplied the container is not vertically growable and this attribute is ignored.

The minW property
Minimum width for layout purposes. The default is 0. If a w attribute is supplied the container is not horizontally growable and this attribute is ignored.

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**
Controls which views of the form include the enclosing object.

`@@tbd` - The interpretation of this attribute is not yet determined: see ViewsXFALangProposal and XFALangMinutes20030731.

**The transient property**
This property specifies whether the processing application must save the value of the exclusion group as part of a form submission or save operation.

```
0
  The exclusion group value must be saved.
1
  The exclusion group must not be saved.
```

**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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.

**The validate property**
A automation element that controls validation of user-supplied data.

For more information see "The validate element".

**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 **w** property
.Width for layout purposes. When width is specified as a measurement, that value 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.

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"
    lock="0 | 1"
    maxLength="-1 | integer"
    name="xml-id"
    transferEncoding="none | base64"
    use="cdata"

>          

One-of properties:

...pcdata...

      
      <#xml> [0..1]

</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 its bind.nullType property.

The #xml property

{error: no property description found for exData.#xml}

For more information see "The #xml element".

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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.

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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
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"
  lock="0 | 1"
  runAt="client | server | both"
  use="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 connectProperties.

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. If this property is missing or empty the connection name defaults to the name of the containing - @@tbd?? subform.

If the value of therunAt 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

{error: no property description found for execute.executeType}

The id property

A unique identifier that may be used to identify this element as a target.
The lock property
A permission flag for allowing or blocking attempted changes to the element.

0
   Allow changes to properties and content.

1
   Block changes to properties and content.

The runAt property
Specifies where the script is to be invoked from.

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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The `exObject` element

An element that describes a single program or implementation-dependent foreign object.

```xml
<exObject
  Properties:
    archive="cdata"
    classId="cdata"
    codeBase="cdata"
    codeType="cdata"
    id="xml:id"
    lock="0 | 1"
    name="xml:id"
    use="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 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 lock property
A permission flag for allowing or blocking attempted changes to the element.
0
  Allow changes to properties and content.
1
  Block changes to properties and content.

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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
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"
  lock="0 | 1"
  name="xml-id"
  use="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 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 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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0

   Allow changes to properties and content.

1

   Block changes to properties and content.

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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The field element

A container element that describes a single interactive container capable of capturing and presenting data content.

```xml
[field

Properties:

access="open | protected | readOnly | nonInteractive"
anchorType="topLeft | topCenter | topRight | middleLeft | middleCenter | middleRight | bottomLeft | bottomCenter | bottomRight"
colSpan="1 | integer"
h="0in | measurement"
hAlign="left | center | right | justify | justifyAll | radix"
id="xml-id"
locale="cdata"
lock="0 | 1"
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"
vAlign="top | middle | bottom"
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:
<connect> [0..n]
<event> [0..n]
<items> [0..n]
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 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 version 2.1 [XFA-Connection-Set] 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. When height is specified as a measurement, that value 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.

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 assume that the last character represents 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 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
Language, currency, and time/date formatting to use for the content of this element.

The locale affects the representation of data formatted, validated, or normalized by picture clauses.

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, for example en_US. For a complete list of valid locale values, refer to the IETF RFC 1766 [RFC1766] and ISO 639 [ISO639] /ISO 3166 [ISO3166] 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 behaviour 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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
Allow changes to properties and content.

1
Block changes to properties and content.
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. If this attribute is not supplied there is no limit.
If an h attribute is supplied the container is not vertically growable and this attribute is ignored.

The maxW property
Maximum width for layout purposes. If this attribute is not supplied there is no limit.
If a w attribute is supplied the container is not horizontally growable and this attribute is ignored.

The minH property
Minimum height for layout purposes. The default is 0.
If an h attribute is supplied the container is not vertically growable and this attribute is ignored.

The minW property
Minimum width for layout purposes. The default is 0.
If a w attribute is supplied the container is not horizontally growable and this attribute is ignored.

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
Controls which views of the form include the enclosing object.
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 non-negative multiple of 90. The default is 0.

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 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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.

The validate property
A automation element that controls validation of user-supplied data.

For more information see "The validate element".

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 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. When width is specified as a measurement, that value 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.

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.

\[
<\text{fill}
\]

**Properties:**

- \text{id}="\text{xml-id}"
- \text{lock}="0 | 1"
- \text{presence}="\text{visible} | \text{invisible} | \text{hidden}"
- \text{use}="\text{cdata}"

\[
<\text{color} \ [0..1]
\]

One-of properties:

- \text{linear} \ [0..1]
- \text{pattern} \ [0..1]
- \text{radial} \ [0..1]
- \text{solid} \ [0..1]
- \text{stipple} \ [0..1]

**Children:**

- \text{extras} \ [0..n]

\[
</\text{fill}>
\]

The fill element is used within the following other elements:

\text{arc} \ \text{border} \ \text{font} \ \text{proto} \ \text{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 \text{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 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 linear property
A fill type element that describes a linear gradient fill.
For more information see "The linear element".

The lock property
A permission flag for allowing or blocking attempted changes to the element.

0
   Allow changes to properties and content.

1
   Block changes to properties and content.

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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The filter element

{error: missing description for filter element}

<filter

Properties:
  id="xml-id"
  lock="0 | 1"
  name="xml-id"
  use="cdata"
>
  <certificates> [0..1]
  <handler> [0..1]
  <reasons> [0..1]
</filter>

The filter element is used within the following other elements:
proto signData

The certificates property
{error: no property description found for filter.certificates}
For more information see "The certificates element".

The handler property
{error: no property description found for filter.handler}
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 lock property
A permission flag for allowing or blocking attempted changes to the element.

  0
  Allow changes to properties and content.

  1
  Block changes to properties and content.

The name property
An identifier that may be used to identify this element in script expressions.

The reasons property
{error: no property description found for filter.reasons}
For more information see "The reasons element".
The use property

Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The float element

A content element that describes a single unit of data content representing a floating point value.

```xml
<float

Properties:
  id="xml:id"
  lock="0 | 1"
  name="xml:id"
  use="cdata"
>
  ...pcdata...
</float>
```

The float element is used within the following other elements:
```xml
desc  exObject  extras  items  proto  value  variables
```

**Content**

This element may enclose float-data which is 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.

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 bind.nullTyproperty.

**The id property**

A unique identifier that may be used to identify this element as a target.

**The lock property**

A permission flag for allowing or blocking attempted changes to the element.

0

Allow changes to properties and content.

1

Block changes to properties and content.

**The name property**

An identifier that may be used to identify this element in script expressions.

**The use property**

Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
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"
- lock="0 | 1"
- overline="0 | 1 | 2"
- overlinePeriod="all | word"
- posture="normal | italic"
- size="10pt | measurement"
- typeface="Courier | cdata"
- underline="0 | 1 | 2"
- underlinePeriod="all | word"
- use="cdata"
- weight="normal | bold"

>  

<fill> [0..1]

Children:
- <extras> [0..n]

</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 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 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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
Allow changes to properties and content.

1
Block changes to properties and content.

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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.

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.

```xml
<format
  id="xml-id"
  lock="0 | 1"
  use="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 lock property

A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.

The picture property

A rendering element that describes input mask and output formatting information.

For more information see "The picture element".

The use property

Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The handler element

{error: missing description for handler element}

<handler

Properties:
  id="xml:id"
  lock="0 | 1"
  type="optional | required"
  use="cdata"
  version="cdata"
>
  ...pcdata...
</handler>

The handler element is used within the following other elements:
filter proto

Content

{error: no property description found for handler.#text}

The id property
A unique identifier that may be used to identify this element as a target.

The lock property
A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.

The type property
{error: no property description found for handler.type}

The use property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.

The version property
{error: no property description found for handler.version}
The image element

A content element that describes a single image.

```xml
<image
Properties:
    aspect="fit | none | actual | width | height"
    contentType="cdata"
    href="cdata"
    id="xml-id"
    lock="0 | 1"
    name="xml-id"
    transferEncoding="base64 | none"
    use="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 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 its bind.nullType property.

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. If this attribute has a value and the element encloses inline image content, the href attribute shall define the source of image content.

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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
Allow changes to properties and content.

1
Block changes to properties and content.

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 specified in [RFC2045].

The use property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The imageEdit element

{error: missing description for imageEdit element}

<imageEdit

Properties:
  data="link | embed"
  id="xml-id"
  lock="0 | 1"
  use="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
{error: no property description found for imageEdit.data}

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 lock property
A permission flag for allowing or blocking attempted changes to the element.

  0
  
  Allow changes to properties and content.

  1
  
  Block changes to properties and content.

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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The integer element

A content element that describes a single unit of data content representing an integer value.

```xml
<integer

Properties:
  id="xml-id"
  lock="0 | 1"
  name="xml-id"
  use="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 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.

When no content is present, the content shall be interpreted as representing a null value, irrespective of the value of the bind.nullType property.

The id property

A unique identifier that may be used to identify this element as a target.

The lock property

A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.

The name property

An identifier that may be used to identify this element in script expressions.

The use property

Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The issuers element

{error: missing description for issuers element}

<issuers>

Properties:
  id="xml-id"
  lock="0 | 1"
  type="optional | required"
  use="cdata"

Children:
  <certificate> [0..n]
</issuers>

The issuers element is used within the following other elements:
certificatesproto

The certificate child
{error: no property description found for issuers.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 lock property
A permission flag for allowing or blocking attempted changes to the element.

  0
  Allow changes to properties and content.

  1
  Block changes to properties and content.

The type property
{error: no property description found for issuers.type}

The use property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The items element

An element that supplies a set of values for a choice list or a check button.

<items

Properties:
  id="xml:id"
  lock="0 | 1"
  name="xml:id"
  presence="visible | invisible | hidden"
  ref="cdata"
  save="0 | 1"
  use="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 checkBox 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 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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
Allow changes to properties and content.

1
Block changes to properties and content.

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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
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"
  lock="0 | 1"
  next="none | contentArea | pageArea"
  previous="none | contentArea | pageArea"
  use="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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
   Allow changes to properties and content.

1
   Block changes to properties and content.

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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The line element

A content element that describes a single rendered line.

```xml
<line
  Properties:
    hand="even | left | right"
    id="xml-id"
    lock="0 | 1"
    slope="\ | /"
    use="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 lock property

A permission flag for allowing or blocking attempted changes to the element.

- **0**
  Allow changes to properties and content.

- **1**
  Block changes to properties and content.
The slope property
This property specifies the orientation of the line.

\n    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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The linear element

A fill type element that describes a linear gradient fill.

<linear

Properties:
  id="xml:id"
  lock="0 | 1"
  type="toRight | toBottom | toLeft | toTop"
  use="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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.

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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The manifest element

An element which contains a list of references to all the nodes that must be included in a signature.

```xml
<manifest
  Properties:
    id="xml-id"
    lock="0  |  1"
    use="cdata"
>
Children:
  <ref>  [0..n]
</manifest>
```

The manifest element is used within the following other elements:
proto signData

**The id property**
A unique identifier that may be used to identify this element as a target.

**The lock property**
A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.

**The ref child**
{error: no property description found for manifest.ref}

For more information see "The ref element".

**The use property**
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The margin element

A box model element that specifies one or more insets for an object.

```xml
<margin
    bottomInset="0in | measurement"
    id="xml-id"
    leftInset="0in | measurement"
    lock="0 | 1"
    rightInset="0in | measurement"
    topInset="0in | measurement"
    use="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 lock property**
A permission flag for allowing or blocking attempted changes to the element.

0
   Allow changes to properties and content.

1
   Block changes to properties and content.

**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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
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"
lock="0 | 1"
long="0in | measurement"
orientation="portrait | landscape"
short="0in | measurement"
stock="cdata"
trayIn="auto | delegate | pageFront"
trayOut="auto | delegate"
use="cdata"

>
</medium>

The medium element is used within the following other elements:
pageAreaproto

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 content of the subform is not available for manipulation by the user. A user-agent should treat the subform as a @@link pass-thru container in sequencing operations, and user must not be permitted to modify the content of the subform. The content of the subform is still modifiable via indirect means such as scripting operations and calculations.

The lock property
A permission flag for allowing or blocking attempted changes to the element.

0
Allow changes to properties and content.

1
Block changes to properties and content.
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
@@tbd

The trayOut property
@@tbd

The use property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The message element

A automation element that holds one or more sub-elements containing validation failure messages.

```xml
<message

Properties:
  id="xml-id"
  lock="0 | 1"
  use="cdata"

> 

Children:
  <text> [0..n]
</message>
```

The message element is used within the following other elements:
- `calculateproto`
- `validate`

If there is only one child text element, the message it contains is used for all messages issued by the enclosing `validate` element.

If there are multiple child elements, their name attributes are used to distinguish them. The child element named `scriptTest` is used for the script validation, `nullTest` for the null validation, and `formatTest` for the format validation. It is erroneous to have more than one child element with the same name or with no name.

**The id property**

A unique identifier that may be used to identify this element as a target.

**The lock property**

A permission flag for allowing or blocking attempted changes to the element.

- 0
  - Allow changes to properties and content.

- 1
  - Block changes to properties and content.

**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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The numericEdit element

A user interface element that describes a widget intended to aid in the manipulation of numeric content.

```xml
<numericEdit
Properties:
  id="xml:id"
  lock="0 | 1"
  use="cdata"
>
  <border> [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 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 lock property**
A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.

**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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The occur element

An element that describes the constraints over the number of allowable instances for its enclosing container.

```xml
<occur
    Properties:
    initial="1 | integer"
    lock="0 | 1"
    max="1 | integer"
    min="1 | integer"
>
    <extras> [0..1]
    <script> [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 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 lock property

A permission flag for allowing or blocking attempted changes to the element.

0

Allow changes to properties and content.

1

Block changes to properties and content.

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 script property
An automation element that contains a script.

For more information see "The script element".
The **oid** element

{error: missing description for oid element}

```xml
<oid

Properties:
  id="xml-id"
  lock="0 | 1"
  name="xml-id"
  use="cdata"
>
  ...pcdata...
</oid>
```

The `oid` element is used within the following other elements:
oids proto

**Content**

{error: no property description found for oid.#text}

**The id property**

A unique identifier that may be used to identify this element as a target.

**The lock property**

A permission flag for allowing or blocking attempted changes to the element.

0

Allow changes to properties and content.

1

Block changes to properties and content.

**The name property**

An identifier that may be used to identify this element in script expressions.

**The use property**

Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The oids element

{error: missing description for oids element}

<oids

Properties:
  id="xml-id"
  lock="0 | 1"
  type="optional | required"
  use="cdata"
>

Children:
  <oid> [0..n]
</oids>

The oids element is used within the following other elements:
  certificatesproto

The id property
A unique identifier that may be used to identify this element as a target.

The lock property
A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.

The oid child
{error: no property description found for oids.oid}
For more information see "The oid element".

The type property
{error: no property description found for oids.type}

The use property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The pageArea element

An element that describes a rendering surface.

<pageArea

Properties:

- blank="0 | integer"
- id="xml-id"
- initialNumber="1 | integer"
- lock="0 | 1"
- name="xml-id"
- numbered="1 | integer"
- relevant="cdata"
- use="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 blank property

This property specifies whether the page area is intended to be blank and therefore may result in special treatment by the output device.

0

The page area is not intended to be blank, and any content will be rendered.

1

The page area is intended to be blank, and may be subject to special treatment by the output device.

For instance, a printer may charge the user on a per-printed-page basis. The user does not wish to be charged for blank backsides of printed pages on a duplexed job. This attribute permits the blank backsides
The document to be marked blank with the result that the processing application must not render any content on the backside and the printer may receive special instructions to ensure that the blank backside is not counted towards the user's charges.

**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 initial page number to the first consecutive occurrence of the page, if it is a numbered page area. @tbd - how do you just continue the current page numbering?? does it distinguish not supplied at all from supplied with an empty string??

**The lock property**
A permission flag for allowing or blocking attempted changes to the element.

0

Allow changes to properties and content.
Block changes to properties and content.

**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 area is considered a numbered page area.

Numbered page areas contribute to the normal incrementing of page numbers, whereas un-numbered pages occur without incrementing page numbering.

- **auto**
  The page area represents a numbered page area. Therefore the instantiation of the page area contributes to the incrementing of the current page area number.

- **none**
  The page area represents a un-numbered page area. Therefore the instantiation of the page area does not contribute to the incrementing of the current page area 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 relevant property**
Controls which views of the form include the enclosing object.

@@tbd - The interpretation of this attribute is not yet determined: see ViewsXFALangProposal and XFALangMinutes20030731.

**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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The pageSet element

An element that describes a set of related page area objects.

```xml
<pageSet

Properties:
  id="xml:id"
  lock="0 | 1"
  name="xml:id"
  relevant="cdata"
  use="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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0

  Allow changes to properties and content.

1

  Block changes to properties and content.

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 relevant property
Controls which views of the form include the enclosing object.
@@tbd - The interpretation of this attribute is not yet determined: see ViewsXFALangProposal and XFALangMinutes20030731.

The use property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
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="opt | measurement"
  lock="0 | 1"
  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"
  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 assume that the last character represents 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 lock property**
A permission flag for allowing or blocking attempted changes to the element.

0
   Allow changes to properties and content.

1
   Block changes to properties and content.

**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 behaviour 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**

@@tbd
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
@@tbd

The tabStops property
@@tbd

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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.

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:
   id="xml-id"
   lock="0 | 1"
   passwordChar="* | cdata"
   use="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 id property**

A unique identifier that may be used to identify this element as a target.

**The lock property**

A permission flag for allowing or blocking attempted changes to the element.

0

Allow changes to properties and content.

1

Block changes to properties and content.

**The margin property**

A box model element that specifies one or more insets for an object.

For more information see "The margin element".

DRAFT
The `passwordChar` property
A single character to be echoed in place of each entered password character. The default is `***` (asterisk).

The `use` property
Invokes a prototype. The value of this property is a `#` character followed by the prototype's identifier.
The pattern element

A fill type element that describes a hatching pattern.

```xml
<pattern

Properties:
  id="xml-id"
  lock="0 | 1"
  type="crossHatch | horizontal | vertical | diagonalLeft | diagonalRight | crossDiagonal"
  use="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 lock property**

A permission flag for allowing or blocking attempted changes to the element.

  0

  Allow changes to properties and content.

  1

  Block changes to properties and content.
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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The picture element

A rendering element that describes input mask and output formatting information.

```xml
<picture
    id="xml-id"
    lock="0 | 1"
    use="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 [XFA-Picture-Clause].

**The id property**

A unique identifier that may be used to identify this element as a target.

**The lock property**

A permission flag for allowing or blocking attempted changes to the element.

0

Allow changes to properties and content.

1

Block changes to properties and content.

**The use property**

Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The proto element

An element that describes a set of reusable element definitions, as described in the section Prototypes above.

<proto

Properties:
  lock="0 | 1"
>

Children:
  <arc> [0..n]
  <area> [0..n]
  <assist> [0..n]
  <barcode> [0..n]
  <bind> [0..n]
  <boolean> [0..n]
  <border> [0..n]
  <break> [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]
  <draw> [0..n]
  <edge> [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]
<line> [0..n]
<linear> [0..n]
<manifest> [0..n]
<margin> [0..n]
<medium> [0..n]
<message> [0..n]
<numERICedit> [0..n]
<occur> [0..n]
<oid> [0..n]
<oids> [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]
<signature> [0..n]
<signatureProperties> [0..n]
<signData> [0..n]
<signedState> [0..n]
<signing> [0..n]
<solid> [0..n]
<Speak> [0..n]
<stipple> [0..n]
<subform> [0..n]
<subformSet> [0..n]
<submit> [0..n]
<template> [0..n]
<text> [0..n]
<textEdit> [0..n]
<time> [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 bar code.

For more information see "The barcode element".

**The bind child**
An element that controls the behavior during merge operations of its enclosing element.

For more information see "The bind 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**
An element that describes the constraints on moving to a new page or content area after rendering an object.

For more information see "The break 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
{error: no property description found for proto.certificate}
For more information see "The certificate element".

The certificates child
{error: no property description found for proto.certificates}
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
{error: no property description found for proto.comb}
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 version 2.1 [XFA-Connection-Set] 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 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 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 aborder or rectangle.
For more information see "The edge 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**

{error: no property description found for proto.filter}

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
{error: no property description found for proto.handler}
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
{error: no property description found for proto.imageEdit}
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
{error: no property description found for proto.issuers}
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 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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
   Allow changes to properties and content.

1
   Block changes to properties and content.

The manifest child
An element which contains a list of references to all the nodes that must be included in a signature.
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 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 validation failure messages.
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
{error: no property description found for proto.oid)
For more information see "The oid element".

The oids child
{error: no property description found for proto.oids)
For more information see "The oids 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 mask 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
{error: no property description found for proto.reason}
For more information see "The reason element".

**The reasons child**
{error: no property description found for proto.reasons}
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**
{error: no property description found for proto.ref}
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 signature child**
An element determining which other elements are signed by a signature.
For more information see "The signature element".

**The signatureProperties child**
{error: no property description found for proto.signatureProperties}
For more information see "The signatureProperties element".

**The signData child**
{error: no property description found for proto.signData}
For more information see "The signData element".

**The signedState child**
{error: no property description found for proto.signedState}
For more information see "The signedState element".

**The signing child**
{error: no property description found for proto.signing}
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 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 template child**
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.

For more information see "The template 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 `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.

```xml
<radial
   id="xml-id"
   lock="0 | 1"
   type="toEdge | toCenter"
   use="cdata">
   <color> [0..1]
   <extras> [0..1]
</radial>
```

The radial element is used within the following other elements:
filt 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 lock property**

A permission flag for allowing or blocking attempted changes to the element.

0

Allow changes to properties and content.

1

Block changes to properties and content.
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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The reason element

{error: missing description for reason element}

<reason

Properties:
   id="xml:id"
   lock="0 | 1"
   name="xml:id"
   use="cdata"
>
   ...pcdata...
</reason>

The reason element is used within the following other elements:
proto reasons

Content
{error: no property description found for reason.#text}

The id property
A unique identifier that may be used to identify this element as a target.

The lock property
A permission flag for allowing or blocking attempted changes to the element.

0
   Allow changes to properties and content.

1
   Block changes to properties and content.

The name property
An identifier that may be used to identify this element in script expressions.

The use property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The reasons element

{error: missing description for reasons element}

<reasons

Properties:
  id="xml:id"
  lock="0 | 1"
  type="optional | required"
  use="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 lock property
A permission flag for allowing or blocking attempted changes to the element.

  0
  Allow changes to properties and content.

  1
  Block changes to properties and content.

The reason child
{error: no property description found for reasons.reason}

For more information see "The reason element".

The type property
{error: no property description found for reasons.type}

The use property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The rectangle element

A content element that describes a single rendered rectangle.

<rectangle

Properties:

hand="even | left | right"
id="xml:id"
lock="0 | 1"
use="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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
Allow changes to properties and content.

1
Block changes to properties and content.

The use property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The ref element

{error: missing description for ref element}

<ref

Properties:
  id="xml-id"
  lock="0  |  1"
  use="cdata"
>
  ...pcdata...
</ref>

The ref element is used within the following other elements:
manifestproto

Content
{error: no property description found for ref.#text}

The id property
A unique identifier that may be used to identify this element as a target.

The lock property
A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.

The use property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The script element

An automation element that contains a script.

```xml
<script

Properties:
  binding="XFA | cdata"
  contentType="application/x-formcalc | cdata"
  id="xml-id"
  lock="0 | 1"
  name="xml-id"
  runAt="client | server | both"
  stateless="0 | 1"
  use="cdata"
>
  ...pcdata...
</script>
```

The script element is used within the following other elements:
calculateevent occur 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 [XFA-FormCalc].

cdata

Support for other script types, such as application/x-ecmascript is implementation-defined.

The id property

A unique identifier that may be used to identify this element as a target.
The lock property
A permission flag for allowing or blocking attempted changes to the element.

0
   Allow changes to properties and content.

1
   Block changes to properties and content.

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 the @@ref for a full discussion of security issues.

The stateless property
Determines whether the a script's variables persist from one invocation to the next.

0
   The script's variables do persist (it is stateful).

1
   The script's variables do not persist (it is stateless).

The use property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The signature element

An element determining which other elements are signed by a signature.

```
<signature

Properties:
id="xml:id"
lock="0 | 1"
type="PDF1.3 | xmlsig"
use="cdata"
>
  <border> [0..1]
  <extras> [0..1]
  <margin> [0..1]
</signature>
```

The signature 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 lock property**

A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.

**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 use property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The signatureProperties element

{error: missing description for signatureProperties element}

<signatureProperties

Properties:
   id="xml:id"
   lock="0 | 1"
   use="cdata"

>   <signedState> [0..1]
</signatureProperties>

The signatureProperties element is used within the following other elements:
proto signData

The id property
A unique identifier that may be used to identify this element as a target.

The lock property
A permission flag for allowing or blocking attempted changes to the element.

0
   Allow changes to properties and content.

1
   Block changes to properties and content.

The signedState property
{error: no property description found for signatureProperties.signedState}

For more information see "The signedState element".

The use property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The signData element

{error: missing description for signData element}

<signData

Properties:
  id="xml:id"
  lock="0 | 1"
  operation="sign | verify | clear"
  ref="cdata"
  target="cdata"
  use="cdata"
>
  <filter> [0..1]
  <manifest> [0..1]
  <signatureProperties> [0..1]
</signData>

The signData element is used within the following other elements:
  event proto

The filter property
{error: no property description found for signData.filter}

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 lock property
A permission flag for allowing or blocking attempted changes to the element.

  0
    Allow changes to properties and content.

  1
    Block changes to properties and content.

The manifest property
An element which contains a list of references to all the nodes that must be included in a signature.

For more information see "The manifest element".

The operation property
{error: no property description found for signData.operation}

The ref property
{error: no property description found for signData.ref}
The `signatureProperties` property
{error: no property description found for signData.signatureProperties}

For more information see "The `signatureProperties` element".

The `target` property
{error: no property description found for signData.target}

The `use` property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The signedState element

{error: missing description for signedState element}

<signedState

Properties:
   id="xml:id"
   lock="0 | 1"
   use="cdata"
>
  ...pcdata...
</signedState>

The signedState element is used within the following other elements:
proto signatureProperties

Content

{error: no property description found for signedState.#text}

The id property
A unique identifier that may be used to identify this element as a target.

The lock property
A permission flag for allowing or blocking attempted changes to the element.

0
   Allow changes to properties and content.

1
   Block changes to properties and content.

The use property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The signing element

{error: missing description for signing element}

<signing

Properties:
  id="xml-id"
  lock="0 | 1"
  type="optional | required"
  use="cdata"
>

Children:
  <certificate> [0..n]
</signing>

The signing element is used within the following other elements:
certificatesproto

The certificate child
{error: no property description found for signing.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 lock property
A permission flag for allowing or blocking attempted changes to the element.

  0
    Allow changes to properties and content.

  1
    Block changes to properties and content.

The type property
{error: no property description found for signing.type}

The use property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The solid element

A fill type element that describes a solid fill.

<solid

Properties:
    id="xml:id"
    lock="0 | 1"
    use="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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
   Allow changes to properties and content.

1
   Block changes to properties and content.

The use property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
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"
  lock="0 | 1"
  priority="custom | toolTip | caption | name"
>
  ...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 lock property

A permission flag for allowing or blocking attempted changes to the element.

0

Allow changes to properties and content.

1

Block changes to properties and content.

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 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 stipple element

A fill type element that describes a stippling effect.

```
<stipple
  id="xml-id"
  lock="0 | 1"
  rate="50 | integer"
  use="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 lock property**
A permission flag for allowing or blocking attempted changes to the element.

0

  Allow changes to properties and content.

1

  Block changes to properties and content.

**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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The subform element

A container element that describes a single subform capable of enclosing other containers.

```
<subform

Properties:
  allowMacro="0 | 1"
  anchorType="topLeft | topLeft | topCenter | topCenter | topRight | topRight | middleLeft |
    middleLeft | middleCenter | middleCenter | middleRight | middleRight | bottomLeft | bottomLeft | bottomCenter | bottomCenter | bottomRight |
  colSpan="1 | integer"
  columnWidths="cdata"
  h="0in | measurement"
  hAlign="left | center | right | justify | justifyAll | radix"
  id="xml-id"
  layout="position | lr-tb | rl-tb | tb | table | row"
  locale="cdata"
  lock="0 | 1"
  maxH="0in | measurement"
  maxW="0in | measurement"
  minH="0in | measurement"
  minW="0in | measurement"
  name="xml-id"
  presence="visible | invisible | hidden"
  relevant="cdata"
  scope="name | none"
  use="cdata"
  vAlign="top | middle | bottom"
  w="0in | measurement"
  x="0in | measurement"
  y="0in | measurement"
>
  <bind> [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]
  <pageSet> [0..1]
  <para> [0..1]
  <traversal> [0..1]
  <validate> [0..1]
  <variables> [0..1]

Children:
  <area> [0..n]
  <connect> [0..n]
```
<draw> [0..n]
<event> [0..n]
<exclGroup> [0..n]
<exObject> [0..n]
<field> [0..n]
<proto> [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.

\texttt{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 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 break property**

An element that describes the constraints on moving to a new page or content area after rendering an object.

For more information see “The break element”.

**The calculate property**

An automation element that controls the calculation of it's 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 version 2.1 [XFA-Connection-Set] 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. When height is specified as a measurement, that value 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.

**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 assume that the last character represents 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 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 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
   Reserved for future use. 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.
The content of the element is flowed in a direction proceeding from top to bottom.

**The locale property**
Language, currency, and time/date formatting to use for the content of this element.
The locale affects the representation of data formatted, validated, or normalized by picture clauses.
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, for example en_US. For a complete list of valid locale values, refer to the IETF RFC 1766 [RFC1766] and ISO 639 [ISO639], ISO 3166 [ISO3166] specifications. Note that this is the same set of locale names used by the xml:langattribute defined in [XML1.0].

When this property is absent or empty the default behaviour 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 on en_US.

**The lock property**
A permission flag for allowing or blocking attempted changes to the element.

- **0**
  Allow changes to properties and content.

- **1**
  Block changes to properties and content.

**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. If this attribute is not supplied there is no limit.
If an h attribute is supplied the container is not vertically growable and this attribute is ignored.

**The maxW property**
Maximum width for layout purposes. If this attribute is not supplied there is no limit.
If a w attribute is supplied the container is not horizontally growable and this attribute is ignored.

**The minH property**
Minimum height for layout purposes. The default is 0.
If an h attribute is supplied the container is not vertically growable and this attribute is ignored.
The **minW property**
Minimum width for layout purposes. The default is 0.
If a `w` attribute is supplied the container is not horizontally growable and this attribute is ignored.

**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 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 above.
For more information see "The proto element".

**The relevant property**
Controls which views of the form include the enclosing object.

@@tbd - The interpretation of this attribute is not yet determined: see ViewsXFALangProposal and XFALangMinutes20030731.
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 **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 a prototype. The value of this property is a '#' character followed by the prototype’s identifier.

The **validate property**
A automation element that controls validation of user-supplied data.

For more information see "The validate element".

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 variables property**
An element to hold document variables.

For more information see "The variables element".

**The w property**
Width for layout purposes. When width is specified as a measurement, that value 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.

**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 positionedlayout. 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 positionedlayout. Defaults to 0.
The subformSet element

An element that describes a set of related subform objects.

```xml
<subformSet

Properties:
  id="xml-id"
  lock="0 | 1"
  name="xml-id"
  relation="ordered | unordered | choice"
  relevant="cdata"
  use="cdata"
>
  <break> [0..1]
  <desc> [0..1]
  <extras> [0..1]
  <occur> [0..1]
</subformSet>
```

The subformSet element is used within the following other elements:
area proto subform subformSet

The break property
An element that describes the constraints on moving to a new page or content area after rendering an object.

For more information see "The break 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 lock property
A permission flag for allowing or blocking attempted changes to the element.
0

 Allow changes to properties and content.

1

 Block changes to properties and content.

**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 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**
Controls which views of the form include the enclosing object.

@@tbd - The interpretation of this attribute is not yet determined: see ViewsXFALangProposal and XFALangMinutes20030731.

**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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
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"
id="xml-id"
lock="0 | 1"
target="cdata"
textEncoding="cdata"
use="cdata"
xdpContent="cdata"

</submit>

The submit element is used within the following other elements:
event proto

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 format property
Determines the format in which the data will be submitted.

xdp
The data is packaged in XDP format, as described in @ref [XDP].

formdata
The data is packaged in-URL encoded format as described in Uniform Resource Locators (URL) [RFC1738]. When packaged in this format the textEncoding property has no effect.

**pdf**
The data is packaged in PDF format as described in the Adobe PDF Specifications [PDF].

**xfd**
The data is packaged in XFD format, as described in [XFD].

**xml**
The data is packaged in XML format as described in the XML Specification version 1.0 [XML]. The schema is determined according to the same rules used for a save operation as described in [XML].

**The id property**
A unique identifier that may be used to identify this element as a target.

**The lock property**
A permission flag for allowing or blocking attempted changes to the element.

0
Allow changes to properties and content.

1
Block changes to properties and content.

**The target property**
The URL to which the data will be submitted.

**The textEncoding property**
The encoding of text content in the referenced document.

Note that the value of this property must be matched against the following values 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], and UTF-8 serialization as defined by ISO/IEC 10646 [ISO10646].

**UTF-16**

The characters are encoded using Unicode code points as defined by [Unicode], and UTF-16 serialization as defined by ISO/IEC 10646 [ISO10646].

**UCS-2**

The characters are encoded using Unicode code points as defined by [Unicode], and UCS-2 serialization as defined by ISO/IEC 10646 [ISO10646].

**fontSpecific**

The characters are encoded in a font-specific way. Each character is represented by one 8-bit byte.

**The use property**

Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.

**The xdpContent property**

Controls what subset of the data is submitted. This property is used only when the format property is xdp.

**datasets pdf xdf**

Elements with the tags datasets, pdf, and xdf 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:
  lock="0 | 1"
>
  <extras> [0..1]

Children:
  <subform> [0..n]
</template>

The template element is used within the following other elements:
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 lock property
A permission flag for allowing or blocking attempted changes to the element.

  0
   Allow changes to properties and content.

  1
   Block changes to properties and content.

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.

```xml
<text

Properties:
   id="xml-id"
   lock="0 | 1"
   maxChars="0 | integer"
   name="xml-id"
   use="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 @@ref.

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.

The id property

A unique identifier that may be used to identify this element as a target.

The lock property

A permission flag for allowing or blocking attempted changes to the element.

0

   Allow changes to properties and content.

1

   Block changes to properties and content.

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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The textEdit element

A user interface element that encloses a widget intended to aid in the manipulation of textual content.

```xml
<textEdit

Properties:
  allowRichText="0 | 1"
  id="xml-id"
  lock="0 | 1"
  multiLine="1 | 0"
  use="cdata"
>
  <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 XFA Data Text Handling Specification version 2.1 [XFA-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

(error: no property description found for textEdit.comb)

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 id property
A unique identifier that may be used to identify this element as a target.

The lock property
A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.

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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The time element

A content element that describes a single unit of data content representing a time value.

```xml
<time

Properties:
  id="xml-id"
  lock="0 | 1"
  name="xml-id"
  use="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 [ISO8601]. This element is intended to hold only the time portion of an ISO8601 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 [ISO8601] as specified in XFA Date and Time Data Formats [XFA-Date-Time].

When no content is present, the content shall be interpreted as representing a null value, irrespective of the value of the bind.nullType property.

**The id property**

A unique identifier that may be used to identify this element as a target.

**The lock property**

A permission flag for allowing or blocking attempted changes to the element.

0

  Allow changes to properties and content.

1

  Block changes to properties and content.

**The name property**

An identifier that may be used to identify this element in script expressions.

**The use property**

Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The `toolTip` element

An element that supplies text for a tool tip. This element is ignored by non-interactive applications.

```
<toolTip
  Properties:
    lock="0 | 1"
>
  ...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 lock property**

A permission flag for allowing or blocking attempted changes to the element.

- 0
  
  Allow changes to properties and content.

- 1
  
  Block changes to properties and content.
The traversal element

An element that links its container to other objects in sequence.

<traversal

Properties:
  id="xml:id"
  lock="0 | 1"
  passThrough="0 | 1"
  use="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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.

The passThrough property
@@omit - not used??

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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The traverse element

An element that declares a single link from its container to another object in a unidirectional chain of links.

```xml
<traverse
  Properties:
    delegate="0 | 1"
    id="xml-id"
    lock="0 | 1"
    operation="next | up | down | left | right | back | first"
    ref="cdata"
    use="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 are not reversible.

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 delegate property
@@omit - not used??

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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.
The operation property
This property determines when this link is used.

next
Used when the user presses the Tab key or enters the final character in a fixed-width field. However the same chain of next links is also traversed by the speech tool when enunciating the form. 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
Used 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 used only when the container is a subform or subform set. The link points to the object that gains focus when the container is entered. In effect the container delegates focus via this link. Defaults to the first container that is a child of this container, in top-to-bottom left-to-right order.

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 script property
An automation element that contains a script.

For more information see "The script element".

The use property
Invokes a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The ui element

A user-interface element that encloses the actual user interface widget element.

<ui

Properties:
  id="xml:id"
  lock="0 | 1"
  use="cdata"
>
  <extras> [0..1]
  <picture> [0..1]

One-of properties:
  <barcode> [0..1]
  <button> [0..1]
  <checkBox> [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 bar code.

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
{error: no property description found for ui.imageEdit}

For more information see "The imageEdit element".

The lock property
A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.
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 mask and output formatting information.

For more information see "The picture element".

The signature property
An element determining which other elements are signed by a signature.

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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The validate element

A automation element that controls validation of user-supplied data.

<validate

Properties:
- formatTest="warning | disabled | error"
- id="xml-id"
- lock="0 | 1"
- nullTest="disabled | warning | error"
- scriptTest="error | disabled | warning"
- use="cdata"

> <extras> [0..1]
- <message> [0..1]
- <picture> [0..1]
- <script> [0..1]
</validate>

The validate element is used within the following other elements:
- exclGroupfield
- 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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
- Allow changes to properties and content.
1
Block changes to properties and content.

**The message property**
A automation element that holds one or more sub-elements containing validation failure messages.
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 mask 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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The value element

A content element that encloses a single unit of data content.

```xml
<value
Properties:
    id="xml:id"
    lock="0 | 1"
    override="0 | 1"
    use="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 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 lock property**
A permission flag for allowing or blocking attempted changes to the element.

0  
Allow changes to properties and content.

1  
Block changes to properties and content.

**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.
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 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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
The variables element

An element to hold document variables.

<variables

Properties:
  id="xml:id"
  lock="0 | 1"
  use="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]
  <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 addressible 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 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 lock property
A permission flag for allowing or blocking attempted changes to the element.

0
  Allow changes to properties and content.

1
  Block changes to properties and content.
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 a prototype. The value of this property is a '#' character followed by the prototype's identifier.
Note: Information on this topic will be provided in a later release of this specification.
Note: Information on this topic will be provided in a later release of this specification.
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.

The monthNames child
Each child describes the (abbreviated) names of the months of the year.

The eraNames child
Each child describes the names of the eras of the calendar.

The meridiemNames child
Each child describes the names of the meridiem.
The currencySymbol element

An element to describe currency symbols.

```xml
<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> [4]

</currencySymbols>

The currencySymbol child

Each child describes a name of currency symbol.
The **datePattern** element

An element to describe the format of a 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 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.

```xml
<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.

```xml
<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.

```xml
<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]
</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 date patterns. Date patterns (date pictures) are described in the “Picture clauses” on page 228.

The timePatterns child
Each child describes the locale's time patterns. Time patterns (time pictures) are described in the “Picture clauses” on page 228.

The dateTimeSymbols child
Each child describes the locale's date time symbols.

The numberPatterns child
Each child describes the locale's number patterns. Number patterns (numeric pictures) are also described in the “Picture clauses” on page 228.

The numberSymbols child
Each child describes the locale's numeric symbols.

The currencySymbols child
Each child describes the locale's currency symbols.
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.

```xml
<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.

```xml
<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 numberSymbol element

An element to describe a number symbol.

```xml
<numberSymbol

Properties:
  name="decimal|grouping|percent|minus" >
  ...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.
The numberSymbols element

An element to describe number symbols.

<numberSymbols>

Children:
  <numberSymbol> [4]
</numberSymbols>

The numberSymbol child

Each child describes a kind of number symbol.
The **timePattern** element

An element to describe the format of a 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 time patterns.

```xml
<timePatterns>
</timePatterns>
```

Children:
- `<timePattern>` [4]

The `timePattern` child

Each child describes the format of a time pattern.
Note: Information on this topic will be provided in a later release of this specification.
Note: Information on this topic will be provided in a later release of this specification. In the mean time, please refer to the previous version of this reference Adobe XML Architecture: FormCalc Specification, version 2.0. You can find that document using the search option on http://partners.adobe.com/asn/.
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 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 Element

The XDP format is comprised of only a single element, known as the xdp element, as follows:

```xml
<xdp:xdp

Properties:
xmns:xdp="http://ns.adobe.com/xdp/"
>
Children
  <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]
</xdp:xdp>
```

This element contains options governing the output document when it is in XDP format.

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 509. Because the XDP format itself is comprised only of the xdp element, the functionality and behaviour 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.

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:

```xml
<xfa:datasets xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
  <xfa:data>
    <book>
```
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:

```xml
<!-- 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>
        <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 the above 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:

```xml
<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>
        <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+PgpzdHJlYW0KeJylWEtv3DYvutX8FKgPZj... 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 pdf element

An XDF pdf element encloses a PDF packet (MIME-type application/pdf). The PDF packet has the following format:

```xml
<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].

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.

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.

```xml
```

The embedPDF property of the submit element specifies whether the PDF packet is embedded within an XDP packet submitted as a result of the event activation.

The datasets element

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.

```xml
<xfa:datasets
  xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
  <xfa:data>
    ...XML form-data content...
  </xfa:data>
</xfa:datasets>
```

“Basic Forms” on page 35 provides more information on the datasets and data elements. The datasets MIME Type is text/xml.
Other Grammars Used with XFA

The following describes other XFA-based grammars that may be included as XDP packets. This list is not exhaustive. That is, other XFA-based grammars may be included in XDP, as needed.

<table>
<thead>
<tr>
<th>Packet name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>config</td>
<td>This packet encloses the configuration settings (XCI). For more information, see “Config Syntax Reference” on page 480. The following shows the format of a config packet: &lt;xfa:config xmlns:xfa=&quot;<a href="http://www.xfa.org/schema/xci/2.1/%22%3E">http://www.xfa.org/schema/xci/2.1/&quot;&gt;</a> ...XML configuration specifications ... &lt;/xfa:config&gt; The config MIME-type is text/xml.</td>
</tr>
<tr>
<td>connectionSet</td>
<td>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). For more information, see @@tbd. The following shows the format of a connectionSet packet: &lt;connectionSet xmlns=&quot;http://www.xfa.org/schema/xfa-connection-set/2.1/&quot;&gt; &lt;wsdlConnection name=&quot;ShoppingCart&quot; ... &gt; ... &lt;/wsdlConnection&gt; &lt;wsdlConnection name=&quot;Catalogue&quot; ... &gt; ... &lt;/wsdlConnection&gt; &lt;wsdlConnection name=&quot;Shipping&quot; ... &gt; ... &lt;/wsdlConnection&gt; &lt;xmlConnection name=&quot;TsAndCs&quot; ... &gt; ... &lt;/xmlConnection&gt; &lt;/connectionSet&gt; &lt;/xfd&gt; The connectionSet MIME-type is text/xml.</td>
</tr>
<tr>
<td>localeSet</td>
<td>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 “Locale Handling” on page 35 and “LocaleSet Reference” on page 481. &lt;xdp xmlns=&quot;http://ns.adobe.com/xdp/&quot;&gt; &lt;localeSet xmlns=&quot;http://www.xfa.org/schema/locale-set/2.1/&quot;&gt; ... &lt;/localeSet&gt; ... &lt;/xdp&gt; The localeSet MIME-type is text/xml.</td>
</tr>
<tr>
<td>sourceSet</td>
<td>This packet contains ADO database queries, used to describe data binding to ADO data sources. The ADO grammar is defined @@tbd. &lt;sourceSet xmlns=&quot;http://www.xfa.org/schema/xfa-source-set/2.1/&quot;/&gt; The sourceSet MIME-type is text/xml.</td>
</tr>
</tbody>
</table>
### Packet name | Description
--- | ---
**template** | This packet contains the form template, as defined by “XFA Template Element Reference” on page 257.

The following shows the format of an XDP packet:

```xml
<xfa:template
  xmlns:xfa="http://www.xfa.org/schema/xfa-template/2.1/"
  ...XML form-template specifications...
</xfa:template>
```

The template MIME-type is `application/x-xfa-template`.

**stylesheet** | This 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 more than one XSLT packet.

```xml
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  ...XSL stylesheet elements...
</xsl:stylesheet>
```

The stylesheet MIME-type is `text/css`.

**xdc** | 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.

```xml
<xsl:xdc
  xmlns:xdc="http://www.xfa.org/schema/xdc/2.1/"
</xsl:xdc>
```

**xfdf** | The xfdf (annotations) packet encloses collaboration annotations placed upon a PDF document and is expressed via a subset (???) of the Adobe XFDF format [XFDF].

```xml
<xfdf xmlns="http://ns.adobe.com/xfdf/"
  xml:space="preserve">
  ...
</xfdf>
```

The xfdf MIME-type is `@@tbd`.

**xmpmeta** | 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. [@@ref]

```xml
<xmpmeta
  xmlns="http://ns.adobe.com/xmpmeta/"
  xml:space="preserve">
  ...
</xmpmeta>
```

The xmpmeta MIME-type is `@@tbd`. 
The references in this section are grouped in the categories: “General”, “Fonts and Character Encoding”, and “Bar Codes”.

General

[Adobe Patent Clarification Notice]

[ECMAScript]
ECMAScript Language Specification, ECMA International

[ISO639]

[ISO3166]

[ISO-4217]
Codes for the representation of currencies and funds”, Available for purchase at: http://www.iso.ch/

[ISO-8601]

[MIMETYPES]
List of registered content types (MIME types) Download list from ftp://ftp.isi.edu/in-notes/iana/assignments/media-types.

[PDF]

[POSIX.1]


[XFA Specification]

[ RFC 1738 ]

Available at: http://www.ietf.org/rfc/rfc1738.txt.

[ RFC 1766 ]

Tags for the Identification of Languages, Internet Engineering Task Force, March 1995

[ RFC 2045 ]

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.

[ RFC 2046 ]

Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. N. Freed, N. Borenstein,
1996.
Available at: http://www.ietf.org/rfc/rfc2046.txt.

[ RFC 2119 ]

RFC2119: Key words for use in RFCs to Indicate Requirement Levels, S. Bradner, March 1997. Available

[ RFC 2376 ]

To be supplied in a later release of this document.

[ RFC 2396 ]


[ SOAP 1.1 ]

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 [ WSDL 1.1 ].

[ WSDL 1.1 ]

Available at: http://www.w3.org/TR/2001/NOTE-wsdl-20010315

[ XPDF ]

[ XML 1.0 ]

Available at: http://www.w3.org/TR/REC-xml

[ XMLBASE ]

Available at http://www.w3.org/TR/xmlbase/.
[XMLSchema]

Available at: [http://www.w3.org/TR/xmlschema-1/](http://www.w3.org/TR/xmlschema-1/) and [http://www.w3.org/TR/xmlschema-2/](http://www.w3.org/TR/xmlschema-2/), respectively.

[XSLT]

Available at [http://www.w3.org/TR/xslt](http://www.w3.org/TR/xslt).

**Fonts and Character Encoding**

[Adobe-Fonts]

Available at: [https://partners.adobe.com/asn/tech/type/ftechnotes.jsp](https://partners.adobe.com/asn/tech/type/ftechnotes.jsp).

[Code-Page-950]


[GB2312]


[ISO-8859-1]


[ISO-8859-2]


[ISO-8859-7]


[ISO-10646]


[KSC5601]

[OpenType]


[Shift-JIS]


[Unicode]


[Unicode-Preface]


[Unicode-Ch6]


Bar Codes

[APO-Barcode]


[Codabar]


[Code2Of5Interleaved]


[Code39]


[Code49]

[Code93]

[Code128-1995]

[Code128-1999]

[ISO-15420]
Available for purchase at: http://www.iso.ch/.

[LOGMAS]
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]

[PDF417]

[QRCode]

[RM4SCC]
How to Use Mailsort Guide, sections "Mailsort 700" and "Mailsort 120". Royal Mail (United Kingdom), 2003.
Available at: http://www.royalmail.com/mailsort.
[Telepen]

[USPS-C100]

[USPS-C840]
Glossary

**application processor**
See “XFA application processor”

**annotation**
Additional content added to a PDF document. Such content includes comments.

**canonical format**

**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**
An object into which content may be placed during the layout operation. Containers include pageAreas, contentAreas, areas, subforms, fields, and exclusion groups. Some containers may also be placed into other containers, however this is restricted to certain combinations of outer and inner container.

**current record**
In XFA 2.1 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 “Basic Forms” on page 35 for more information about record processing.

**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.

**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.

**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.

**form creator**
The person and/or software that creates a form template, possibly along with other information such as a data description.

**form data**

**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.
**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).

**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.

**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 set**

Information about locales. A locale set includes predefined set of conventions for representing dates, times, numbers, and currency.

**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.

**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” on page 93.

**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.

**re-normalizing the Data DOM**

See “normalizing the Data DOM”

**SOM (XFA Script Object Model)**

A model for referencing values, properties and methods within a particular Document Object Model (DOM).

**SOM resolver**

**source set**

ADO database queries, used to describe data binding to ADO data sources

**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.

**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".

**XCI (XML Configuration Information)**

Configuration information for a Presentation Agent output driver. The root element in the XCI grammar is the config element.

**XDC (XML Device Control)**

**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)**

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 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.

**SOM (XFA Script Object Model)**

See “SOM (XFA Script Object Model)".

**XFA name**

A string suitable for identifying an object in an XFA DOM, using the XFA Scripting Object Model [XFA-SOM] syntax. A valid XFA name must be a valid XML name, as defined in [XML1.0], with the additional restriction that it must not contain a colon (:) character. Note that this is less restrictive than XFA 2.0.

**XFD (XML Form Data)**

**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)**

**XML Data Document**

Well-formed XML document containing data that is processed by XFA processing applications.

**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.