Developing Applications Using Mars

Version 0.8

Draft

November 2007

Developing Applications Using Mars, Version 0.8

Edition 1.1, November 2007

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Contents

List of Examples **7**

Preface **9**

What's in this guide? 9
Who should read this guide? 9
We want to hear from you 9
Related documentation 9

1 Introduction **10**

About Mars 10
Mars and XML-based workflows 11
Document creation 11
Document processing 11
Display 12
Document creation process 12
Instructions for creating a simple document 12
Instructions for adding metadata and navigational features 13
Instructions for creating Mars documents that include forms, logical structure and private data 13

Part I: Creating a Basic Mars Document

1 Developing Mars Documents **15**

Understanding the structure of a Mars document 15
Packaging the Mars files into a ZIP file 16

2 Developing the Backbone **18**

Understanding the backbone 18
Creating the Mars backbone file 19
Specifying the Mars namespace and other identifiers 19
Specifying page references 20
Specifying layout and viewing characteristics 21
Specifying navigation features 24

3 Developing Pages **27**

Understanding pages 27
SVG represents page content 28
Backbone file specifies page placement 28
SVG coordinates place content in the current user space 29
Creating pages 30
Creating the page infrastructure 30
Creating the SVG file and the basic SVG elements 31
Creating SVG expressions that describe graphic content and text 31

4 Referencing and Embedding Fonts **35**

Understanding fonts 35
Supported fonts 35
Font-related structures 36
Referencing and embedding fonts 37

Planning 38
Defining font face specifiers 38
Establishing the font characteristics to apply to text 39
Specifying a font descriptor 40
Embedding OpenType fonts 42
Converting an existing font to the OpenType font format 45

5 Adding Images to SVG Content 46
Understanding images and color profiles 46
About images 46
Image information 46
Supported image formats 46
About color profiles 47
Using images 47
Specifying color profile definitions for images 47
Adding an image to a page 48

6 Specifying Color Spaces 50
Understanding color spaces 50
Formats used with color space type 50
Defining and using a color space for text and graphic content 57
Defining color profiles in SVG 58
Using ICC color spaces to specify color for text or graphic objects 58
Associating non-ICC color spaces with text or graphic objects 59

Part II: Adding Interactive Features to a Mars Document

8 Creating Bookmarks 62
Understanding bookmarks 62
Specifying bookmarks 63
Referencing the bookmarks file from the backbone 63
Creating the bookmarks file 63
Specifying destinations 65
Explicit destinations 66
Named destinations 67

9 Representing Actions 74
Understanding actions 74
Predefined actions 74
Specifying actions 76
Go-to action 76
URI action 77
Sound action 77
JavaScript action 78

10 Adding Annotations 82
Understanding annotations 82
Content annotations 82
Markup annotations 83
Display properties of annotations: basic properties, appearances and default appearances 85
Specifying content and markup annotations 87
Creating content annotations 87
Creating markup annotations 89

11 Creating Articles 100
Understanding articles 100
Creating articles 100
Planning 100
Referencing the threads file from the backbone file 100
Creating the articles file 101
12 Specifying Optional Content 103
Understanding optional content 103
Specifying optional content 103
Planning 103
Establishing the optional content groups 104
Marking SVG graphic content and text for inclusion in optional content groups 108
Marking non-SVG graphic content and text for inclusion in optional content groups 109
13 Creating Portable Collections 111
Understanding portable collections 111
Specifying portable collections 111
Planning 112
Embedding the files in the Mars document 112
Providing information about the embedded files 112
Describing the appearance of the information in the collection user interface 112

Part III: Adding Application Interchange to a Mars Document

15 Specifying Metadata 117
Understanding XMP metadata 117
Specifying document-level metadata 118
16 Specifying Marked Content, Logical Structure and Tagging 121
Understanding marked content 121
Understanding logical structure 122
Logical representation 123
Paths describe structure 124
Connection between the structure tree and the content 125
Node naming convention 127
Understanding tagging 129
Creating a Mars document with marked content 130
Planning for marked content 130
Indicating the Mars document contains marked content 130
Creating SVG that includes marked-content tags 130
Creating a Mars document with logical structure and tagging 131
Planning the logical structure 132
Specifying the structuring state and class mapping 133
Identifying SVG content groups 134
Creating a page-level structure file 135
Referencing the page-level structure file in the page information file 136
Creating a document-level structure cache 137
17 Creating Forms 140
Understanding forms 140
Form fields 142
Representing XML forms in a Mars document 142

Display properties of annotations: basic properties, appearances and default appearances ... 146
Creating XML forms ... 148
Adding the form files to the /form directory ... 149
Referencing the form files ... 150
Specifying containers and fields that mirror the template ... 151
Creating buttons, text fields and choice fields ... 152
Specifying XML form field actions ... 158
Specifying XML form predefined appearances ... 159
Creating XML form data ... 159
Creating Acrobat forms ... 160
Planning ... 160
Creating Acrobat field elements in the backbone file ... 161
Creating the widget annotations that provide the field behavior ... 162
Specifying Acrobat form field actions ... 163
Specifying a submit-form action ... 164
Specifying Acrobat form predefined appearances ... 165
Creating Acrobat form data ... 166

18 Adding Foreign Data (Page Piece Dictionaries) ... 168

Understanding foreign data ... 168
About private data specifically allowed by the Mars schema ... 169
Format for private data ... 169
Adding private data to application datasets ... 171
Registering with Adobe ... 172
Adding an application dataset ... 172
Adding private data to logical-structure ... 173
Indicating the logical structure contains private data ... 173
Adding private data to the structure elements ... 173

A Tasks Enabled by Mars ... 175

Index ... 177

List of Examples

Example 1.1 Implicit associations between files (container.xml) 17
Example 2.1 Specifying version information 19
Example 2.2 Specifying page mode and page labels (backbone.xml) 22
Example 2.3 Specifying viewer preferences 23
Example 2.4 Specifying an open action 23
Example 2.5 Specifying page labels 25
Example 3.1 A simple backbone file (/backbone.xml) 28
Example 3.2 A simple SVG page file (filename: /page/0/pg.svg) 28
Example 3.3 Backbone file specifies the base user space for the page 29
Example 3.4 SVG coordinates place content in the current user space 30
Example 3.5 Page information file (/page/0/info.xml) 31
Example 3.6 Specifying a header (path: /page/page-number/pg.svg) 33
Example 3.7 Specifying a watermark (path: /page/page-number/pg.svg) 33
Example 3.8 Specifying the page content file (/page/page-number/pg.svg) 33
Example 4.1 Font specifier for a non-embedded font 39
Example 4.2 Font specifier for an embedded font 39
Example 4.3 Font descriptor file contents (/font/*name*.fd or /page/page_number/name.fd) 42
Example 4.4 Applying obfuscation to a font 45
Example 5.1 Associating a color profile with an image (/page/page-number/pg.svg) 49
Example 6.1 Specifying color profiles (/page/page-number/pg.svg) 58
Example 6.2 Associating a color space with a graphic element (/page/page-number/pg.svg) 59
Example 6.3 Using a device color space (/page/page-number/pg.svg) 60
Example 8.1 Specifying the location of the bookmarks file (backbone.xml) 63
Example 8.2 Specifying bookmarks with actions (path name: /bookmarks.xml) 64
Example 8.3 Specifying named destinations (bookmarks.xml) 70
Example 8.4 Specifying named destinations (/page/0/dests.xml) 71
Example 8.5 Specifying a named destination cache (/cache/names.xml) 73
Example 9.1 Specifying a GoTo action (bookmarks.xml) 76
Example 9.2 Specifying a URI action 77
Example 9.3 Specifying a sound action (bookmarks.xml) 78
Example 9.4 A widget annotation that specifies JavaScript code (/page/page_number/pg.can) 79
Example 9.5 A document action that references a named JavaScript segment (/backbone.xml) 79
Example 9.6 Specifying the location of the JavaScript files (backbone.xml) 80
Example 9.7 Defining the named JavaScript segments (/script/javascripts.js) 81
Example 10.1 Specifying a default appearance 86
Example 10.2 Page information file (/page/page_number/info.xml) provides location of content annotations 87
Example 10.3 Specifying a link content annotation (/page/page-number/pg.can) 89

Example 10.4 Specifying a text markup annotation (/page/page-number/pg.svg.ann) 92
Example 10.5 Specifying a highlight markup annotation (/page/page-number/pg.svg.ann) 93
Example 10.6 Specifying a line markup annotation (path name: /page/page-number/pg.svg.ann) 94
Example 10.7 Specifying an embedded file (/file/embedded_files.xml) 97
Example 10.8 Specifying a file attachment annotation (/page/page-number/pg.svg.ann) 98
Example 10.9 SVG that draws the graphic referenced from the Down appearance 98
Example 10.10 SVG that draws the graphic referenced from the Normal appearance 99
Example 11.1 Specifying threads and beads 102
Example 12.1 Backbone defines document-level optional content settings 106
Example 12.2 SVG content associated with an optional content group 108
Example 12.3 109
Example 13.1 Backbone file from a collection 114
Example 13.2 Embedded files for a collection 114
Example 15.1 Specifying XMP metadata (/META-INF/metadata.xml) 119
Example 16.1 Descriptive structure tree (does not conform to Mars node-naming conventions) 125
Example 16.2 Structure tree using numeric node names (conforms to the Mars node-naming conventions) 125
Example 16.3 Marked content (/page/page_number/pg.svg) 131
Example 16.4 Node associated with attributes contained in the class map 132
Example 16.5 Role Map (/backbone.xml) 134
Example 16.6 Marked content identifiers added to an existing SVG file (/page/page_number/pg.svg) 135
Example 16.7 Page-level structure file (/page/page_number/struct.xml) 136
Example 16.8 Page information file references page-level structure file 137
Example 16.9 Predefined document-level structure cache (/cache/struct.xml) 138
Example 17.1 Template subform and field size and placement (template.xft) 146
Example 17.2 Annotations created for the XML form template (/page/0/pg.can) 146
Example 17.3 Specifying a default appearance 148
Example 17.4 XML form parts (backbone.xml) 150
Example 17.5 AcroForm/Fields element reflects the XFA template structure 152
Example 17.6 Specifying a button field for an XML form 153
Example 17.7 A text field connecting the template field with the annotation (backbone.xml) 154
Example 17.8 Choice list with display values that differ from export values 155
Example 17.9 Specifying a text, button, and choice fields for an XML form 156
Example 17.10 Saving data from an XML form 159
Example 17.11 Specifying a button and text fields for an Acrobat form 162
Example 17.12 Specifying a submit-form action (/page/page_number/pg.can) 164
Example 17.13 Specifying a submit-form action (/page/page_number/pg.can) 165
Example 17.14 SVG content that provides the content for the appearance (/page/0/form_613-2.svg) 166
Example 17.15 Saving data in XFDF format 167
Example 18.1 Specifying private application data 173
Example 18.2 Specifying UserProperties 174

Preface

This guide provides information and examples on how to develop documents and applications using Mars, an XML-based representation of Adobe® PDF documents.

What's in this guide?

This guide is a companion to the *Mars Reference* and provides information about developing documents and custom applications using Mars. Each chapter includes brief examples with explanations.

Who should read this guide?

This document is for developers who want to leverage their XML tools and knowledge to create, manipulate, and extract information from Mars documents.

This document assumes that you understand XML and are familiar with programming on your development platform.

We want to hear from you

Mars is a work in progress. We want to hear from you about which Mars features you find most valuable and what enhancements you would like to see added to Mars.

You can share your comments on the Adobe Labs Mars web site, available at http://www.adobe.com/go/mars. Select the "Community" tab and then the "Mars General Discussion Forum" link.

Related documentation

The resources listed in this table can help you learn more about the product.

For information about	**See**
Mars syntax and semantics	*Mars Reference* at the following location: http://www.adobe.com/go/mars
PDF file format	*PDF Reference, sixth edition, version 1.7* at the following location: http://www.adobe.com/go/acrobat_developer
OEBPS Container Format	http://idpf.org/specs.htm
XML schema data types	http://www.w3.org/TR/2004/REC-xml-20040204/
XML Forms Architecture (XFA)	Select "XML" from the "Technology Centers" panel at the following location: http://www.adobe.com/devnet
Scalable Vector Graphics (SVG)	http://www.w3.org/Graphics/SVG/

1 Introduction

This chapter introduces the capabilities and development workflows available through Mars. It summarizes the capabilities inherent to Mars; it can help you understand the overall approach to developing applications for Mars; and it provides a guide to understanding and approaching the remaining chapters in this book.

About Mars

Mars provides an XML-based representation of PDF documents. It represents document information by combining XML, images, fonts, and color profiles within a Zip-based package.

Many of the technologies used in Mars are managed by external standards organizations. For example, Mars uses Scalable Vector Graphics (SVG) to describe non-varying content that should be drawn on a page. SVG is an open standard maintained by the W3C (http://w3.org). The Mars Zip-based package is based on the specification for Open eBook Publication Structure Container Format (OCF) 1.0. OCF is an open standard maintained by the International Digital Publishing Forum (http://idpf.org).

Mars documents, like PDF documents, can be displayed within a viewer application. When viewed, Mars documents have the same appearance and behavior as a PDF document with the same content. Other than having a different file suffix, a user would not see any difference between a PDF and Mars document.

Installing the Mars plug-in for Adobe Acrobat® 8, enables users to open Mars documents and to save Mars documents as PDF and visa versa. The Mars plug-in for Adobe Acrobat 8 is available from http://adobe.com/go/mars.

The information in a Mars document is organized similar to the information in a PDF document. In Mars, the pages, images, fonts, bookmarks, and other document components appear as separate files within the Zip package. The Mars components such as bookmarks are more independent and easier to manipulate by virtue of their XML representation. Thus, Mars enables developers to interoperate with XML and standards-based formats.

Mars addresses the following goals:

- It provides an XML representation of PDF.
- It supports developers who want to leverage their XML tools and knowledge to create, manipulate, and extract information from PDF documents.
- It provides an XML document solution for organizations that have chosen to unify their infrastructure using XML as the base representation.
- It defines and implements a representation of PDF information based on reusable XML components.

These reusable XML components represent self-contained pieces of document information that can be used in a variety of contexts, some of which do not involve PDF documents.

Mars provides the following advantages:

- Leverage from existing standards: Use of public standards such as XML, SVG and XPath enable you to take advantage of existing knowledge and tools, which can simplify the development of programs that create, manipulate and examine Mars documents.

- Ease of use: Annotations, bookmarks, JavaScript™, fonts, metadata, external references, specialized processing data, and attachments are separated from the content of the Mars document. This allows them to be easily created and manipulated.
- Efficiency: The page-level assembly of documents allows for selected pages to be read independently of the other pages.

Mars and XML-based workflows

You can create, process and display documents. These capabilities are explained below.

Document creation

You can use Acrobat to convert PDF into Mars documents and vice versa. You can also create XML components representing pages and the overall document structure and then combine them in a Zip package. For more information on creating the Zip package, see "Understanding the structure of a Mars document" on page 15.

These are some of the many workflows:

- Creation of Mars using XML tools and then converting back to PDF for distribution and processing
- Conversion of PDF to Mars for use as a template, modification of the Mars template using XML tools, and converting back to PDF for distribution and processing
- Conversion of PDF to Mars, addition of standard components to PDFs such as watermarks and cover pages, and then conversion back to PDF.

Creation and processing of Mars documents is made easy by the use of XML and HTML-like references between document components.

See the appendix "Tasks Enabled by Mars" on page 175 for a list of individual tasks that can be used in document processing.

Document processing

Many document applications do not necessarily involve creation of documents but rather involve the processing or modification of information in existing documents. Mars documents are composed of separate files in a Zip container which simplifies processing. In many cases, the needed information is in a specific file in the document package and therefore is easy to locate and process.

Examples of document processing tasks include:

- Manipulation of auxiliary content: Creation and manipulation of areas of the document, including annotations, bookmarks, JavaScript, fonts, metadata, external references, specialized processing data, and attachments.
- Document assembly and disassembly: Assembling documents from the page level.
- Extracting document information: Extracting information about a document's fonts, images or form data.
- Page content creation and manipulation: Creating and manipulating page content represented as SVG.

Breaking the document into separate files also provides a mechanism for efficiently accessing parts of the document. Each file represents a separate entry point into the overall document that can be read independently of other content.

See the appendix "Tasks Enabled by Mars" on page 175 for a list of individual tasks that can be used in document processing.

Display

PDF and Mars documents contain various kinds of information, including page content, images, fonts, color definitions, form data, scripts, annotations, font information, and named destinations. Not all of this information is required to display only a part of the document; PDF documents are processed by reading only the required objects, resulting in increased performance when accessing large documents. Similarly, Mars documents are structured so that it is not necessary to read all of the files in the Zip package into memory ahead of time. In general, only the information required to perform a particular operation is read.

See the appendix "Tasks Enabled by Mars" on page 175 for a list of individual tasks that can be used in document processing.

Document creation process

The steps for creating a Mars document are defined in detail in the chapters throughout this book.

Instructions for creating a simple document

The following table describes the chapters that cover creation of a simple document.

Chapter	Description
Developing Mars Documents	Explains the physical structure of Mars documents, including the file structure packaging hierarchy, as well as directory and file naming conventions.
Developing the Backbone	Explains what the Mars document backbone elements are and how to create them.
Developing Pages	Explains how to specify content for pages within a Mars document.
Referencing and Embedding Fonts	Explains how to define fonts, and how to specify embedded fonts.
Adding Images to SVG Content	Explains how to specify and reference images in a Mars document.
Specifying Color Spaces	Explains what color spaces are and how to use ICC and PDF color spaces in Mars documents.

Instructions for adding metadata and navigational features

The following chapters explain how to add metadata and navigational features to an existing Mars document.

Chapter	Description
Specifying Metadata	Explains how to specify metadata using the Extensible Metadata Platform (XMP) standard and how to associate metadata with images.
Representing Actions	Explains how to represent predefined actions and JavaScript actions in a Mars document.
Adding Annotations	Explains how to create and use content and markup annotations in Mars documents.
Creating Bookmarks	Explains how bookmarks are used and how to create them. It also describes how bookmarks provide document-level navigation through the use of explicit and named destinations.
Creating Articles	Explains how to create article threads which allow users to navigate through logically-related information located on different pages within a document.

Instructions for creating Mars documents that include forms, logical structure and private data

The following chapters explain how to create mars document that include forms, logical structure and private data. These steps are done in conjunction with creating a simple Mars document.

Chapter	Description
Creating Forms	Explains how to create static XML forms that comply with the Adobe XML Form Architecture (XFA) and Acrobat forms. Mars also supports dynamic XML forms; however, this guide does not yet address this topic.
Specifying Marked Content, Logical Structure and Tagging	Explains how to specify logical structure in a document to enable reflow, accessibility, and content reuse.
Adding Foreign Data (Page Piece Dictionaries)	Explains how to add private application data and custom object attributes.

Part I: Creating a Basic Mars Document

1 Developing Mars Documents

This chapter describes the physical structure of Mars documents including the recommended directory and file naming conventions.

This chapter contains the following information.

Topic	Description	See
Understanding the structure of a Mars document	Explains the structure of Mars documents and certain rules that apply across all parts of a Mars document.	page 15
Packaging the Mars files into a ZIP file	Explains how to create the Mars package.	page 16

Understanding the structure of a Mars document

A *Mars document* (also called a *Mars package*) consists of a number of files that, when interpreted, represent a document that can have the same characteristics as a PDF document. The grammar of each type of file in a Mars document is specified in the *Mars Reference* and the Mars schemas.

The structure of the files in a Mars document is a hierarchy that starts with a single root file. The root file has references to other files in the Mars document, which in turn can have references to still other files. These explicit references are specified in the *Mars Reference* and the Mars schemas.

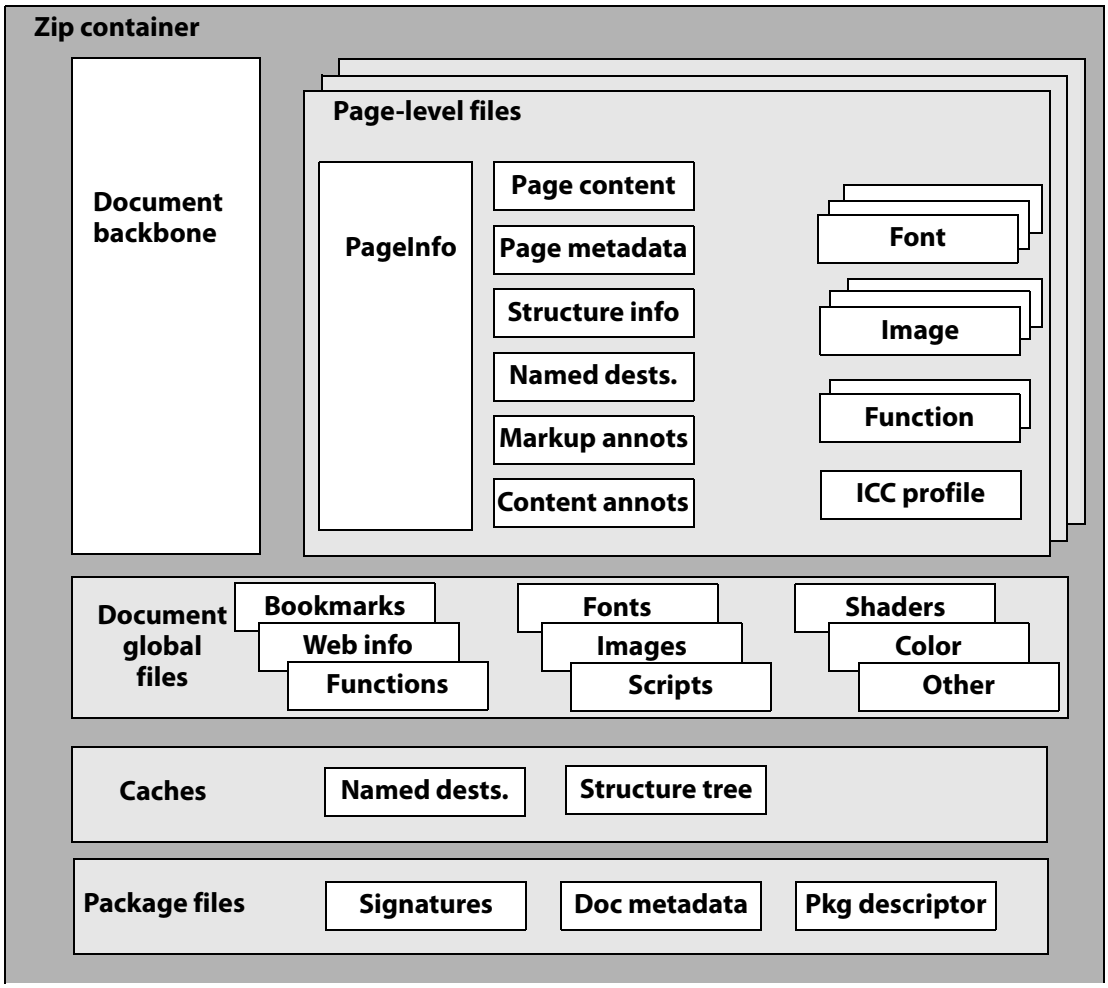
The structure of the files in a Mars document is also reflected in the directory and file naming conventions described here and used throughout this guide. Most of these conventions are optional; however, their use can improve consistency between documents. This guide prefaces non-required directory and file names with the term *conventionally* to indicate their use is suggested by convention rather than by requirement, as reflected in the following sentence:

> Conventionally, the directory containing the files that comprise a page is named /page/*page-number*, where *page-number* is the zero-based page number of the page.

The root file, called a backbone file, describes the overall document and references other files that provide subordinate aspects of the document. Examples of such subordinate files are the page information files (one per page), the cache files and the bookmarks file. Each page information file describes a single page in the document and references other files that provide the page content and other data unique to that page.

The following figure shows the structure of a Mars document.

Structure of a Mars document



For information on the structure of a Mars document, see the *Mars Reference*.

Packaging the Mars files into a ZIP file

To create a Mars package using an application interface or Zip, perform the following tasks:

1. Create a root directory with the name of the Mars document with the suffix `mars`.
2. In the root directory, create a file called mimetype. This file must be physically the first file in the package. The file's contents must be the Mars MIME type `application/vnd.adobe.x-mars`. For more information, see the specification for the Open eBook Publication Structure Container Format (OCF) 1.0.

   Note: This MIME type is a placeholder MIME type; the final MIME type will be chosen when the name for the format is finalized.
3. Add the backbone.xml file to the root of the package, as described in "Developing the Backbone" on page 18.

4. Create the META-INF directory at the root of the package.

5. Create the container.xml file in the META-INF directory, modifying its contents to represent metadata association rules, as described in *Universal Container Format*, located in the *Mars Specification*. The following example shows the contents of the container.xml file. This file content is the same for all Mars documents.

Example 1.1 ***Implicit associations between files (container.xml)***

```
<?xml version="1.0" encoding="UTF-8"?>
<container version="1.0"
   xmlns="urn:oasis:names:tc:opendocument:xmlns:container">
   <relationships xmlns:pdf="http://ns.adobe.com/pdf/2006">
      <relationship type="metadata" target="$path.xmp"/>
      <relationship type="pdf:annotation" target="$path.ann"/>
   </relationships>
</container>
```

6. Add the `page` directory and the nested page folders and files, as described in "Developing Pages" on page 27.

7. Add any additional directories and files based on the document's requirements. The remaining chapters in this guide discuss other features that can be added to a Mars document.

8. Compress the files in the package, using an application such as WinZip or the Java class `java.util.zip` For more information, see http://java.sun.com/reference/api/.

It is recommended to add the files in the order that they will used by the document. This is especially important for performance when viewing documents on the Internet.

2 Developing the Backbone

This chapter describes the Mars document backbone elements and how to define them.

This chapter contains the following information.

Topic	Description	See
Understanding the backbone	Explains the requirements for developing a Mars backbone.	page 18
Specifying the Mars namespace and other identifiers	Explains how to specify the Mars namespace and the other identifiers that correlate the Mars document to the Mars schema and to the version of the PDF Reference.	page 19
Specifying page references	Explains how to add page references to the Mars backbone file. The page data resides in other files in the Mars document.	page 20
Specifying layout and viewing characteristics	Explains how to specify the layout and viewing characteristics of a Mars document.	page 21
Specifying navigation features	Explains how to specify the navigational characteristics of a Mars document.	page 24

Understanding the backbone

The root of a Mars document is known as the *backbone* and exists in a file called `backbone.xml`. This file is considered the *root* of the Mars document because it acts as the starting point for the hierarchy of Mars files that comprise a Mars document.

The backbone file identifies the version of Mars used in the document and the PDF version represented in the document. It also contains references to other objects defining the document's contents, outline (bookmarks), article threads, named destinations, and other attributes. In addition, it contains information about how the document should be displayed on the screen, such as whether its outline and thumbnail page images should be displayed automatically and whether some location other than the first page should be shown when the document is opened. Files referenced from the backbone can reference other files that are part of the document but are not directly referenced from the backbone.

The top-level element in a Mars document is `PDF`. The `PDF` element and its child elements are referred to as the Mars *backbone*. Each element in the `PDF` element describe a different Mars document feature. The following table describes some of the `PDF` child elements:

Element	Description
`Pages`	References the individual information files, each of which describes the files that comprise a page
`AcroForm`	Describes the fields used in the document.
`Bookmarks`	Describes the bookmarks (outline) for the document.

Creating the Mars backbone file

This section explains how to create a basic Mars backbone file. After you finish creating the files that comprise the Mars document, you must package the files as described in "Packaging the Mars files into a ZIP file" on page 16.

Specifying the Mars namespace and other identifiers

The following steps explain how to specify the XML declaration, Mars namespace, and other identifiers in the backbone.xml file. These attributes identify the Mars release to which the Mars package conforms and the version of PDF represented by the Mars document. The following example shows the expected result.

Example 2.1 ***Specifying version information***

```
<PDF PDFVersion="1.6" Version="0.8.0" Class="04-18-07"
   xmlns="http://ns.adobe.com/pdf/2006" DocumentID="AWC1mzaNiUmPDGVNM/z0Vg=="
   InstanceID="1RJssP8RsEi6i0ruarhYbw==">
```

1. Create an XML file named backbone.xml, located in the root directory. Ensure the standard XML declaration appears on the first line:

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

2. Create a `PDF` element as the root element in backbone.xml. This element corresponds to the top-level element in the *Mars Reference*.

3. Set the `PDF` element's default namespace as follows:

   ```
   xmlns="http://ns.adobe.com/pdf/2006"
   ```

4. Set the `PDF` element's `PDFVersion` attribute to indicate the PDF version that the Mars document represents. The naming convention for the `PDFVersion` attribute is in the form *major.minor* where *major* and *minor* are integers. The following are an incomplete set of supported values:

   1.6 indicates the Mars document is compatible with the *PDF Reference version 1.6*

   1.7 indicates the Mars document is compatible with the *PDF Reference, version 1.7*

5. Set the `PDF` element's `Version` attribute to identify the Mars schema version being used. The naming convention for the `Version` attribute is in the form *major.minor.micro* where *major*, *minor*, and *micro* are integers. The numbering convention is that smaller numbers are considered to be earlier than larger numbers. The `Version` attribute must be formatted correctly for the document to be considered valid. As of this guide's publication date, the current version was 0.8.0. See the release notes for more current information.

6. Set the `PDF` element's `Class` attribute to identify the document compatibility level. This attribute is used to prevent problems due to major format changes which cannot be handled in an upward compatible manner. It is used primarily when the Mars format is being developed and in beta test so that unsupported files can be detected. As of this guide's publication date, the current class value was `04-18-07`. See the release notes for more current information.

7. Set the PDF element's `DocumentID` attribute to a URI that specifies a unique identifier for the document. This identifier is subsequently used for obfuscating fonts embedded in the Mars document. The value is a string that represents a Base-64-encoded URI.

Specifying page references

This section explains how to add page references to a Mars backbone. Such references specify the pages that comprise a Mars document. The following figure illustrates the relationship between page references and page data. The page data is a hierarchy of files that specify an individual page, as described in "Developing Pages" on page 27.

File name: backbone.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<PDF … >
   <Pages>
      <Page src="/page/0/info.xml"
         x1="0" y1="0" x2="612" y2="792" ID="2" />
      <Page src="/page/1/info.xml"
         x1="0" y1="0" x2="612" y2="792" ID="0" />
      <Page src="/page/2/info.xml"
         x1="0" y1="0" x2="612" y2="792" ID="1" />
   </Pages>
   …
</PDF>
```

File name: page/0/info.xml

```
<Page xmlns="…">
   <Contents src="…"/>
</Page>
```

File name: page/1/info.xml

```
<Page xmlns="…">
   <Contents src="…"/>
</Page>
```

File name: page/2/info.xml

```
<Page xmlns="…">
   <Contents src="…"/>
</Page>
```

To add pages to a Mars document, perform the following tasks:

1. Create a `Pages` element as a child of the root `PDF` element. The `Pages` element specifies the pages in the document.

2. For each page in the document, add a `Page` element as a child of the root `Pages` element. The order of `Page` elements defines the order of pages in the document. Each individual `Page` element includes information such as the source file for the page, the size of the page, and an identifier to be used in the case that the page needs to be referenced. Set the following attributes for a `Page` element:
   - Set the `src` attribute to specify the location where the page information file can be found.
   - Set the `x1`, `x2`, `y1`, and `y2` attributes to specify the page size coordinates. In this example, the upper-left corner of the window for this page is located at `(0,0)` and the lower-right corner of the window is located at `(612,792)`.

     The coordinates are specified in default user space units, which are defined in the *PDF Reference*. For the page size coordinates, the default for the size of a unit in default user space (1/72 inch) is approximately the same as a point. Because coordinates in user space may be specified as either integers or real numbers, the unit size in default user space does not constrain positions to any arbitrary grid. The resolution of coordinates in user space is not related in any way to the resolution of pixels in device space.
   - Add an `ID` attribute, which is a unique identifier used if the page is to be referenced from elsewhere in this document. This attribute is not required.

Specifying layout and viewing characteristics

This section explains how to specify the layout and viewing characteristics of a Mars document.

Initial layout

The `PageMode` and `PageLayout` attributes of the `PDF` element advise the Mars viewer on which navigational tools to display and how to lay out the window when the document is opened. Depending on user preferences and other conditions, the document cannot be guaranteed to be displayed as requested by setting these attributes. The `PageMode` attribute specifies which tools should be displayed when opened. The `PageLayout` specifies the page layout to be used when the document is opened.

To set the page mode and page layout for a Mars document, set the following attributes in the `PDF` element:

1. Set the `PageMode` attribute to specify how the document should be displayed when opened. To see the defined values for this attribute, see the `PageMode` entry in Table 3.25 of the *PDF Reference, version 1.7*.

2. Set the `PageLayout` attribute to specify the page layout to be used when the document is opened. To see the defined values for this attribute, see the `PageLayout` entry in Table 3.25 of the *PDF Reference, version 1.7*.

In the example below, the `PageLayout` attribute is set to `TwoPageRight`, which means that the pages are displayed two at a time, with odd-numbered pages on the right when the document is opened. The `PageMode` attribute is set to `UseOutlines`, which means that the document's bookmarks will be visible when the document is opened.

Example 2.2 ***Specifying page mode and page labels (backbone.xml)***

```
<PDF PDFVersion="1.6" Version="0.8.0" Class="04-18-07"
   xmlns="http://ns.adobe.com/pdf/2006" PageMode="UseOutlines"
   PageLayout="TwoPageRight" >
   <Pages>
      <Page src="/page/0/info.xml" x1="0" y1="0" x2="612" y2="792" ID="0"/>
      <Page src="/page/1/info.xml" x1="0" y1="0" x2="612" y2="792" ID="1"/>
      <Page src="/page/2/info.xml" x1="0" y1="0" x2="612" y2="792" ID="2"/>
   </Pages>
</PDF>
```

For more information, see the *Mars Reference*.

Viewer preferences

The `ViewerPreferences` element controls the way the document is to be presented on the screen or in print. If no viewer preferences are specified, viewing and printing applications behave in accordance with the current user preference settings.

To set viewer preferences for a Mars document, perform the following tasks:

1. Create the `ViewerPreferences` element as a child of the root `PDF` element.
2. Set the `ViewerPreferences` element `CenterWindow` attribute to `true` to specify whether the document should be centered in the window. The default value is `false`.
3. Set the `ViewerPreferences` element `DisplayDocTitle` attribute to specify whether the document title should appear in the window title bar. If false, the title bar should instead display the name of the Mars document. The default value is `false`.
4. Set the `ViewerPreferences` element `HideToolTitle` attribute to `true` to specify whether to hide the viewer application's tool bars when the document is active. The default value is `false`.
5. Set the `ViewerPreferences` element `NonFullScreenPageMode` attribute to `true` to specify how to display the document on exiting full-screen mode. The `NonFullScreenPageMode` entry in Table 8.1 of the *PDF Reference, version 1.7* describes the supported values.
6. Set the other `ViewerPreferences` element attributes as needed. These attributes all correspond to the entries in Table 8.1 of the *PDF Reference, version 1.7*.

The following example shows how to set the more common viewer preferences. In this example, the `ViewerPreferences` element specifies the following characteristics:

- The document window is centered in the screen
- The window title bar displays the document title taken from the document description
- The toolbar is hidden
- The menubar is displayed
- The document outline is displayed on exiting full-screen mode

Example 2.3 ***Specifying viewer preferences***

```
<PDF PDFVersion="1.7" Version="0.8.0" Class="04-18-07"
   xmlns="http://ns.adobe.com/pdf/2006">
   <ViewerPreferences
      CenterWindow="true"
      DisplayDocTitle="true"
      HideToolbar="true"
      NonFullScreenPageMode="UseOutlines"
   / >
   …
</PDF>
```

For more information on viewer preferences, see Table 8.1 in the *PDF Reference, version 1.7*.

Open actions

Normally, when a document is opened, the first page is displayed using the default magnification factor. You can specify that a different page be shown when the document is opened, however, by setting the `AfterOpen` element. The `AfterOpen` element sets a value that specifies a destination to be displayed or an action to be performed when the document is opened.

The following example specifies an `AfterOpen` element that opens `/backbone.xml#1` at the upper-left corner of the window at `(72, 587)` with no magnification applied.

Example 2.4 ***Specifying an open action***

```
<PDF PDFVersion="1.6" Version="0.8.0" Class="04-18-07"
   xmlns="http://ns.adobe.com/pdf/2006" >
   <Pages>
      <Page src="/page/0/info.xml" x1="0" y1="0" x2="612" y2="792" ID="0"/>
      <Page src="/page/1/info.xml" x1="0" y1="0" x2="612" y2="792" ID="1"/>
      <Page src="/page/2/info.xml" x1="0" y1="0" x2="612" y2="792" ID="2"/>
   </Pages>
   <AfterOpen>
      <GoTo><Dest><XYZ Page_ref="/backbone.xml#1" Left="72" Top="587"
         Zoom=""/></Dest></GoTo>
   </AfterOpen>
</PDF>
```

To set an open action for a Mars document, perform the following tasks:

1. Decide what action should be performed on document open. This is typically a `Goto` action which causes a page other than the first page to be displayed, but it can be any action including execution of an embedded JavaScript. For this example, we will assume it is a `Goto` action.

2. Add an `AfterOpen` element to the root `PDF` element in the document backbone.

3. Create a `GoTo` action to change the view to a specified destination (page, location, and magnification factor). For more information, see "Understanding actions" on page 74. In this case, the `Goto` action consists of a `GoTo` element containing a destination element.

4. Create the `Dest` element to define the view of the document.
    - Set the `Page_ref` attribute that references the page to be displayed. For this example, the page being referenced is the page with the value ID=1. The value used for `Page_ref` is a URI style reference. For this example, `/backbone.xml#1` represents an object with an `ID` of `1` stored in the `backbone.xml` file.
    - Set the `Left` and `Top` coordinates that specify the upper-left corner of the window to be displayed.
    - Set the `Zoom` element to the magnification for the contents of the page. In this example, the zoom value is `null`, which means that the zoom factor will not be changed for this page. For more information, see "Specifying destinations" on page 65.

For more information on actions, see "Specifying actions" on page 76.

For more information on named destinations, see "Named destinations" on page 67.

Specifying navigation features

The Mars backbone can include elements that enable the user to interact with a global overview of the document or to navigate from page to page within a document. This section explains how to add elements that specify the more common navigation features: page labels, bookmarks, and articles.

Page labels

The `PageLabels` element specifies additional content added to each page of a viewed or printed document. Page labels can specify formatted strings that optionally include the current page number. The `PageLabels` element is a child of the root `PDF` element in the `backbone.xml` file.

A `PageLabel` element is created for each set of page ranges and label formats within the `PageLabels` element in the document backbone. A page range is a series of consecutive pages, where the first and last page in the range are identified.

A page's label consists of a numeric portion based on its position within its labeling range, optionally preceded by a label prefix denoting the range itself. For example, the pages in an appendix might be labeled with decimal numeric portions prefixed with the string *A.*; the resulting page labels would be *A.1*, *A.2*, and so on. Page labels and page indices need not coincide: the indices are fixed, running consecutively through the document starting from 0 for the first page, but the labels can be specified in any way that is appropriate for the particular document.

Each page in a Mars document is identified by an integer page index that expresses the page's relative position within the document, and is specified via the `Start` attribute of the `PageLabel` element. The `Start` attribute specifies the first page in a labeling range. In addition to the `Start` attribute, the `Style` attribute defines the labeling characteristics for the pages in that range. This is what allows for distinct preface, appendix, and page numbers within a document.

The `Style` attribute sets the numbering style to be used for the numeric portion of each page label. In Example 3.4, the document begins with 21 pages of content numbered in `Roman_Lowercase` and the

remainder of the document is numbered in the `Numeric` style. For a list of possible `Style` attribute values, see the table below.

Style	Description
`Numeric`	Decimal arabic numerals
`Roman_Uppercase`	Uppercase roman numerals
`Roman_Lowercase`	Lowercase roman numerals
`Letter_Uppercase`	Uppercase letters (A to Z for the first 26 pages, AA to ZZ for the next 26, and so on)
`Letter_Lowercase`	Lowercase letters (a to z for the first 26 pages, aa to zz for the next 26, and so on)

The following example shows how to specify a `Roman_Lowercase` style for the first page through the twenty-second page of the document and a `Numeric` style for the twenty-third and subsequent pages. The example sets the first page with a page label of "i" (Roman lower-case), and the twenty-second page with a label of *"23"*.

Example 2.5 ***Specifying page labels***

```
<PDF PDFVersion="1.6" Version="0.8.0" Class="04-18-07"
   xmlns="http://ns.adobe.com/pdf/2006" >
   <Pages>
      <Page src="/page/0/info.xml" x1="0" y1="0" x2="612" y2="792" ID="0"/>
      <Page src="/page/1/info.xml" x1="0" y1="0" x2="612" y2="792" ID="1"/>
      <Page src="/page/2/info.xml" x1="0" y1="0" x2="612" y2="792" ID="2"/>
      …
      <Page src="/page/99/info.xml" x1="0" y1="0" x2="612" y2="792" ID="99"/>
   </Pages>
   <PageLabels>
      <PageLabel FirstInRange="0" Style="Roman_Lowercase"></PageLabel>
      <PageLabel FirstInRange="22" Style="Numeric"></PageLabel>
   </PageLabels>
</PDF>
```

To create page labels for a Mars document, perform the following tasks:

1. Analyze the document to determine what kinds of page numbering are used and in what order. For each range, select the most appropriate style from the table above. For example, a document may have front matter numbered i, ii, iii, iv, and so on up to xxii, followed by pages numbered 1 to the end of the document.

2. In the document backbone file, /backbone.xml, add a `PageLabels` element as a child of the root `PDF` element.

3. Create `PageLabel` elements and set the `FirstInRange` and `Style` attributes.

For more information on page labels, see the `PageLabels` entry in Table 3.25, and section 8.3.1 in the *PDF Reference, version 1.7*.

Bookmarks

Bookmarks allow the user to navigate interactively from one part of the document to another. For Mars documents, bookmark definitions are in a separate file referenced from the backbone. They can reference page ordinals, the structure tree, or they can contain arbitrary actions.

For more information on bookmarks and how to create them, see "Understanding bookmarks" on page 62.

Articles

Articles enable the user to navigate from page to page within a document. They chain together items of content within the document that are logically connected but not physically sequential.

For more information on articles and how to create them, see "Understanding articles" on page 100.

3 Developing Pages

This chapter describes how to develop and specify the content for a page within a Mars document. The information for a page is spread across a number of files and the names of these files and their purpose are also explained in this chapter.

This chapter contains the following information.

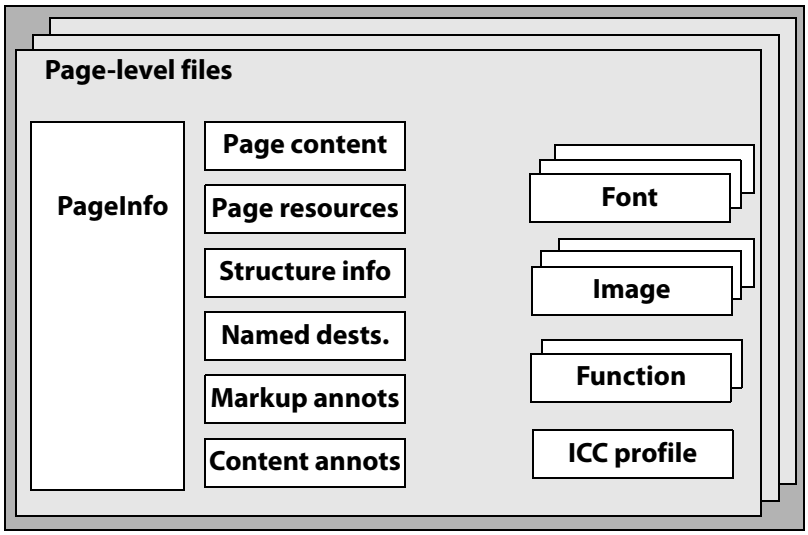
Topic	Description	See
Understanding pages	Explains the basic page information needed to develop a page.	page 27
Creating pages	Explains how to create a page for a Mars document.	page 30

Understanding pages

Each page in a Mars document is represented by a collection of files contained in a single directory. The directory is referenced from the document backbone, as described in "Developing the Backbone" on page 18. Conventionally, the directory containing the files that comprise a page is named /page/*page-number*, where *page-number* is the zero-based page number of the page.

The figure below shows a graphical representation of page structure for a Mars document.

Page structure for a Mars document



The files required to define a page are the page information and page content file. The page information file is the *root* of a page and includes the page size, orientation, and a reference to the page content file. The page content file contains all the declaration information about the page.

For information on the files that comprise a page of a Mars document, see the *Mars Reference*.

SVG represents page content

Page content is represented in SVG format, which is a W3C standard for representing text and graphics. SVG is a language used to describe two-dimensional graphics in XML and it allows the use of three different types of graphic objects.

- Vector graphic shapes
- Images
- Text

The SVG references for images are in a format similar to HTML image references. Other resources such as fonts, color spaces and shaders are also referenced in the page content file. The color spaces are used to describe colors. Some color spaces are related to device color representation (grayscale, RGB, CMYK), others to human visual perception (CIE-based). The shaders provide a smooth transition between colors across an area to be painted, independent of the resolution of any particular output device and without specifying the number of steps in the color transition.

Pages in a Mars document can share these resources by having the shared resources in separate files and referencing those files from the required pages.

In addition, the page content itself can be broken into multiple files using the SVG `symbol` mechanism. For more information, see the *SVG Specification*.

Backbone file specifies page placement

Below is an example that shows the XML for the backbone file, page information file, and corresponding page content file example. In the `backbone.xml` file, three pages are created, each having an upper-left corner of `(0,0)` and a lower-right corner of `(612, 792)`.

Example 3.1 ***A simple backbone file (/backbone.xml)***

```
<PDF PDFVersion="1.6" Version="0.8.0" Class="04-18-07" >
   <Pages>
      <Page src="/page/0/info.xml" x1="0" y1="0" x2="612" y2="792" ID="0"/>
      <Page src="/page/1/info.xml" x1="0" y1="0" x2="612" y2="792" ID="1"/>
      <Page src="/page/2/info.xml" x1="0" y1="0" x2="612" y2="792" ID="2"/>
   </Pages>
</PDF>
```

In the SVG file, the font is specified, and the text is set to Hello World in a 32 point font. For an explanation of the SVG elements, and a more detailed example, see "Creating pages" on page 30.

Example 3.2 ***A simple SVG page file (filename: /page/0/pg.svg)***

```
<svg xmlns="http://www.w3.org/2000/svg"
   xml:space="preserve" fill="none" stroke="none" width="612" height="792">
   <defs>
      <font-face font-family="F3">
         <font-face-src>
            <font-face-uri xlink:href="/font/054C66D4._sfnt"/>
            <font-face-name name="Interstate"/>
         </font-face-src>
      </font-face>
   </defs>
```

```
   <g>
      <text transform="matrix(1 0 0 1 200 100)" font-size="32"
         font-family="F3" fill="rgb(0,0,0)
         device-color(DeviceGray,0)" fill-rule="evenodd">
         <tspan >Hello World</tspan>
      </text>
   </g>
</svg>
```

SVG coordinates place content in the current user space

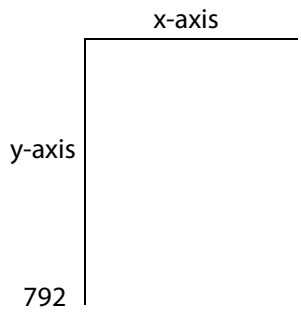
SVG page coordinates are set via the `x`, `y`, `width` and `height` attributes. The `x` attribute specifies the x-axis coordinate of the left corner of the rectangular region where an embedded `svg` element is placed. The `y` attribute specifies the y-axis coordinate of the top corner of the rectangular region into which an embedded `svg` element is placed. The `width` attribute sets the width of the SVG document fragment. For embedded `svg` elements, this is the width of the rectangular region into which the `svg` element is placed. A `height` attribute sets the height of the SVG document fragment. For embedded `svg` elements, this is the height of the rectangular region into which the `svg` element is placed. A negative value is an error. A value of zero disables rendering of the element.

Example 3.3 specifies the user space for the page.

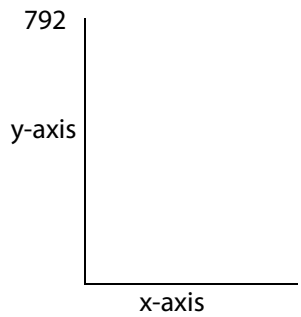
Example 3.3 ***Backbone file specifies the base user space for the page***

```
<PDF PDFVersion="1.6" Version="0.8.0" Class="04-18-07"
   xmlns="http://ns.adobe.com/pdf/2006">
   <Pages>
      <Page src="/page/0/info.xml" x1="0" y1="0" x2="567" y2="756"/>
   </Pages>
</PDF>
```

In Example 3.4, the `g` element's `transform` attribute specifies a matrix that flips the current user space upside down and shifts the y-axis down by 792 user units, as shown in the following illustration.



User space before the transformation



User space after the transformation

The image element uses the `width`, `height`, `x`, and `y` attributes to place an image on the current user space (as modified by the transform attribute in the `g` element).

Example 3.4 ***SVG coordinates place content in the current user space***

```
<svg
   xmlns="http://www.w3.org/2000/svg"
   xml:space="preserve"
   xmlns:pdf="http://ns.adobe.com/pdf/2006"
   xmlns:xlink="http://www.w3.org/1999/xlink" …>
   <defs/>
   <!-- The transform matrix flips and then shifts the Y-axis. -->
      and shifts the y-axis.-->
   <g transform="matrix(1 0 0 -1 0 792)">
      <!-- The transform matrix re-flips the Y-axis.-->
      <image xlink:href="/page/0/im_2030206195-1.png"
         <!-- The image is placed in user space in the rectangle defined by: -->
         width="297.4031" height="39.54" x="72" y="708.06"
         <!-- The transform matrix re-flips the Y-axis.-->
         transform="matrix(1 0 0 -1 0 0)" … />
   </g>
</svg>
```

Creating pages

This section explains how to create pages for a Mars document. After you finish creating the files that comprise the Mars document, you must package the files as described in "Packaging the Mars files into a ZIP file" on page 16.

Creating the page infrastructure

The following steps explain how to create the directory and page information that establish the infrastructure for each page in the document. These steps assume you have created a backbone file that contains one `Page` element for each of the pages in the document, as described in "Creating the Mars backbone file" on page 19.

1. Create a directory called /page/*page-number*, where *page-number* corresponds to the zero-based page number of the page. For example, create the directory /page/0/ for the first page in the document.

2. In the directory you created for the page, create the page information file called `info.xml`. Ensure the standard XML declaration appears as the first line in this file:
   ```
   <?xml version="1.0" encoding="UTF-8"?>
   ```

3. Create the `Page` element as the root element, setting the default namespace, as follows:
   ```
   xmlns="http://ns.adobe.com/pdf/2006"
   ```

4. Create a `Contents` element as a child of the `Page` element, setting its `src` attribute to the location where the page content file can be found.

5. In the `Page` element, add other attributes and elements, as needed.

Below is an example of a simple page information file that specifies the page content file location as /page/0/pg.svg.

Example 3.5 ***Page information file (/page/0/info.xml)***

```
<Page xmlns="http://ns.adobe.com/pdf/2006">
   <Contents src="/page/0/pg.svg"/>
</Page>
```

Creating the SVG file and the basic SVG elements

The following steps explain how to create the skeleton of an SVG file. These steps assume you have completed the steps described in "Creating the page infrastructure" on page 30.

1. Create the page content file called pg.svg, ensuring the standard XML declaration appears as the first line in the file: `<?xml version="1.0" encoding="utf-8"?>`

   By convention, this file is located in the /page/page_number directory

2. Create the `svg` element as the root element, setting namespaces for svg, pdf and xlink, as follows:

   ```
   xmlns="http://www.w3.org/2000/svg"
   xmlns:pdf="http://ns.adobe.com/pdf/2006"
   xmlns:xlink="http://www.w3.org/1999/xlink"
   ```

3. Create the `defs` element as a child of the `svg` element. This element defines fonts, color spaces, and other resources to be used by the page. The `defs` element acts as a container element for URI-referenced elements.

The World Wide Web Consortium provides specifications and tutorials for SVG (http://w3.org).

Creating SVG expressions that describe graphic content and text

The following steps explain how to add simple SVG graphic content (a rectangle, a polyline, a watermark, and text) with corresponding attributes and values that can be found on the SVG Graphic Content image shown further in this chapter. These steps assume you have completed the steps in "Creating the SVG file and the basic SVG elements" on page 31.

To create rectangles, perform the following tasks:

1. Create a `rect` element following the `defs` element, and set its `x` and `y` to the coordinates on the page where the rectangle should appear. For information on SVG coordinates, see "SVG coordinates place content in the current user space" on page 29.

2. Set the `rect` element's `width` and `height` attributes to specify the size of the rectangle.

3. If the rectangle should be filled (solid), set the `rect` element's `fill` attribute to specify the color with which to fill the rectangle. By default, the rectangle is unfilled.

   The solid blue rectangle specified below and rendered on page 34 specifies placement in the upper-right corner of the page (`475,40`) and specifies a width of `100` and a height of `65`. The blue color for the rectangle is specified by `rgb(0,0,240)`.

   ```
   <!--A solid blue rectangle. -->
   <rect x="475" y="40" width="100" height="65" fill="rgb(0,0,240)"/>
   ```

4. If the rectangle should have an outline, set the `rect` element's `stroke` attribute to specify the color to use for the outline, and the `stroke-width` attribute to specify the stroke width. The default stroke width is 1 user unit.

   The red rectangle outline specified below and rendered on page 34 specifies the same placement and size as the solid blue rectangle. Rather than specify a solid fill, the rectangle specifies a stroke outline that is red.

```
<!--An unfilled rectangle with a red border
   that overlays the solid blue box-->
<rect x="475" y="40" fill="none" stroke="rgb(255,0,0)"
   width="100" height="65"/>
```

To create a page border, perform the following tasks:

1. Create a `polyline` element as a child of the `rect` element. The `polyline` element defines a set of connected straight line segments. Typically, `polyline` elements define open shapes.

2. Set the `polyline` element's attributes as follows:

   - Set the `points` attribute to specify an array of numbers representing the alternating x and y coordinates used to create the polyline.
   - Set the color and width to be used by the `stroke` attribute for the line. The `stroke` attribute paints a line along the current path. The stroked line follows each straight or curved segment in the path, centered on the segment with sides parallel to it. Each of the path's subpaths is treated separately. The color or pattern of the line is determined by the current color and colorspace for stroking operations.
   - Set the `stroke-width` element that sets the line width thickness of the line used to stroke a path. It is a non-negative number expressed in user space units.

     To create the polyline around the outline of the page with the corners cut out as shown in the Mars document, a `polyline` element was specified and the stroke attribute was set to a width of `5` and the color to `rgb(255,0,255)` which is a pinkish magenta color.

```
<polyline points="20,10 590,10 590,20 605,20 605,770 590,770
   590, 785 20,785 0,770 5,770 5,20 20,20 20,5"
   stroke="rgb(255,0,255)" stroke-width="5"/>
```

To create a watermark and a header on the page, perform the following tasks:

1. Create a `text` element that defines a graphics element consisting of text. The glyphs to be rendered are defined by the XML character data within the `text` element, along with relevant properties and Unicode-character-to-glyph mapping tables within the font itself. The `text` element properties indicate such things as the writing direction, font specification and painting attributes which describe how to render the characters. Since `text` elements are rendered using the same rendering methods as other graphics elements, all of the same coordinate system transformations, painting, clipping and masking features that apply to shapes such as paths and rectangles also apply to `text` elements. Set the following attributes for a `text` element:

   - Set the `x` and `y` attributes to the position for the text in the current user coordinate system.
   - Set the `font-size`, and `font-family` attributes to specify the size, and type of font. For more information, see "Referencing and Embedding Fonts" on page 35.
   - Set the `fill` element which uses the current nonstroking color to paint the entire region enclosed by the current path. If the path consists of several disconnected subpaths, `fill` paints the insides of all subpaths, considered together. Any subpaths that are open are implicitly closed before being filled.

- Set the `tspan` element to adjust the text and font properties. The current text position can also be adjusted using the absolute or relative coordinate values.

  The following example creates a header that displays `Ad agency preliminary copy` at the top of the page. A red fill is used for the text.

Example 3.6 ***Specifying a header (path: /page/page-number/pg.svg)***

```
<!--A header that appears at the top of the page.-->
<text x="225" y="30" font-size="12" font-family="F2" fill="rgb(255,0,0)">
   <tspan>Ad agency preliminary copy</tspan>
</text>
```

The following example specifies a watermark-style text, creates a `text` element and uses the `transform` element combined with the `rotate(45)` function. This rotates the text 45 degrees. See the `PRELIMINARY` text in the figure below for an example of what this looks like.

Note: You can also use watermark annotations to apply a watermark that is separate from the SVG content. "Adding Annotations" on page 82 provides more information.

Example 3.7 ***Specifying a watermark (path: /page/page-number/pg.svg)***

```
<!-- A watermark that appears at an angle on the page.-->
<text x="275" y="60" transform="rotate(45)" font-size="36"
   font-family="F2" fill="rgb(255,0,255)" opacity=".5">
   <tspan>PRELIMINARY</tspan>
</text>
```

The following example combines the preceding examples to illustrate the order of SVG elements within the `pg.svg` file for the SVG graphic content example. It shows how to create two rectangles, text, a polyline a watermark, and a header.

Example 3.8 ***Specifying the page content file (/page/page-number/pg.svg)***

```
<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg"
   xml:space="preserve" fill="none" stroke="none" width="612" height="792">
   <defs>
      <font-face font-family="F2">
         <font-face-src>
            <font-face-name name="Times New Roman"/>
         </font-face-src>
      </font-face>
      <color-profile name="cs-0" xlink:href="/color/cs-0.icc" pdf:numC="3"/>
   </defs>
   <!-- Other definitions used by normal page content go here.-->
   <g>
      <!--Normal page content goes here.-->
   </g>
   <!-- Watermarks and additional graphics.-->
   <g>
      <!--A solid blue box-->
      <rect x="475" y="40" width="100" height="65" fill="rgb(0,0,240)"/>
      <!--An unfilled box with a red border that overlays the solid blue box-->
      <rect x="475" y="40" fill="none" stroke="rgb(255,0,0)"
         width="100" height="65"/>
      <!-- A border that appears at the outer edges of the page.-->
```

```
      <polyline points="20,10 590,10 590,20 605,20 605,770 590,770 590,785
        20,785 20,770 5,770 5,20 20,20 20,5" stroke="rgb(255,0,255)"
        stroke-width="5"/>
      <!--A watermark that appears at the top of the page.-->
      <text x="225" y="30" font-size="12" font-family="F2" fill="rgb(255,0,0)">
        <tspan>Ad agency preliminary copy</tspan>
      </text>
      <!-- A watermark that appears at an angle on the page.-->
      <text x="275" y="60" transform="rotate(45)" font-size="36"
        font-family="F2" fill="rgb(255,0,255)" opacity=".5">
        <tspan>PRELIMINARY</tspan>
      </text>
    </g>
</svg>
```

For more information on how to represent text and graphic content, see the *SVG Reference*.

The image below represents the SVG graphic content example. It displays the rectangles, polyline, and watermark graphics that were added to the first page of the Mars document. All the XML examples in this section correspond to this figure.

SVG graphic content

4 Referencing and Embedding Fonts

This chapter describes how to reference and embed fonts in the SVG files of a Mars document. It does not describe font specifications in rich text expressions, which can appear in non-SVG files.

This chapter contains the following information.

Topic	Description	See
Understanding fonts	The supported font types and their representation in a Mars document.	page 35
Referencing and embedding fonts	Explains how to create Mars and SVG expression that specify the fonts used to render specific text	page 38

Understanding fonts

This section describes the font types that Mars supports and the XML and SVG expressions that identify a specific font face.

Supported fonts

Mars supports embedded and non-embedded OpenType fonts and embedded SVG fonts.

Note: Currently, Mars supports only OpenType fonts.

OpenType fonts

The OpenType font format is an extension of the TrueType font format, adding support for PostScript® font data. The OpenType font format was developed jointly by Microsoft and Adobe. OpenType fonts and the operating system services which support OpenType fonts provide users with a simple way to install and use fonts, whether the fonts contain TrueType outlines or CFF (PostScript) outlines. For more information on OpenType fonts, see http://www.adobe.com/go/opentype.

Note: The Adobe Font Developer Kit can be used to convert certain types of fonts to OpenType fonts. See www.adobe.com/go/opentype.

SVG fonts

SVG fonts can be embedded in the SVG page directly or referenced in a separate file that is included in the Mars package.

SVG fonts must not define any `glyph` elements with more than one Unicode character within the `unicode` attribute. This restriction ensures the ability to map the glyph to a unique Unicode character.

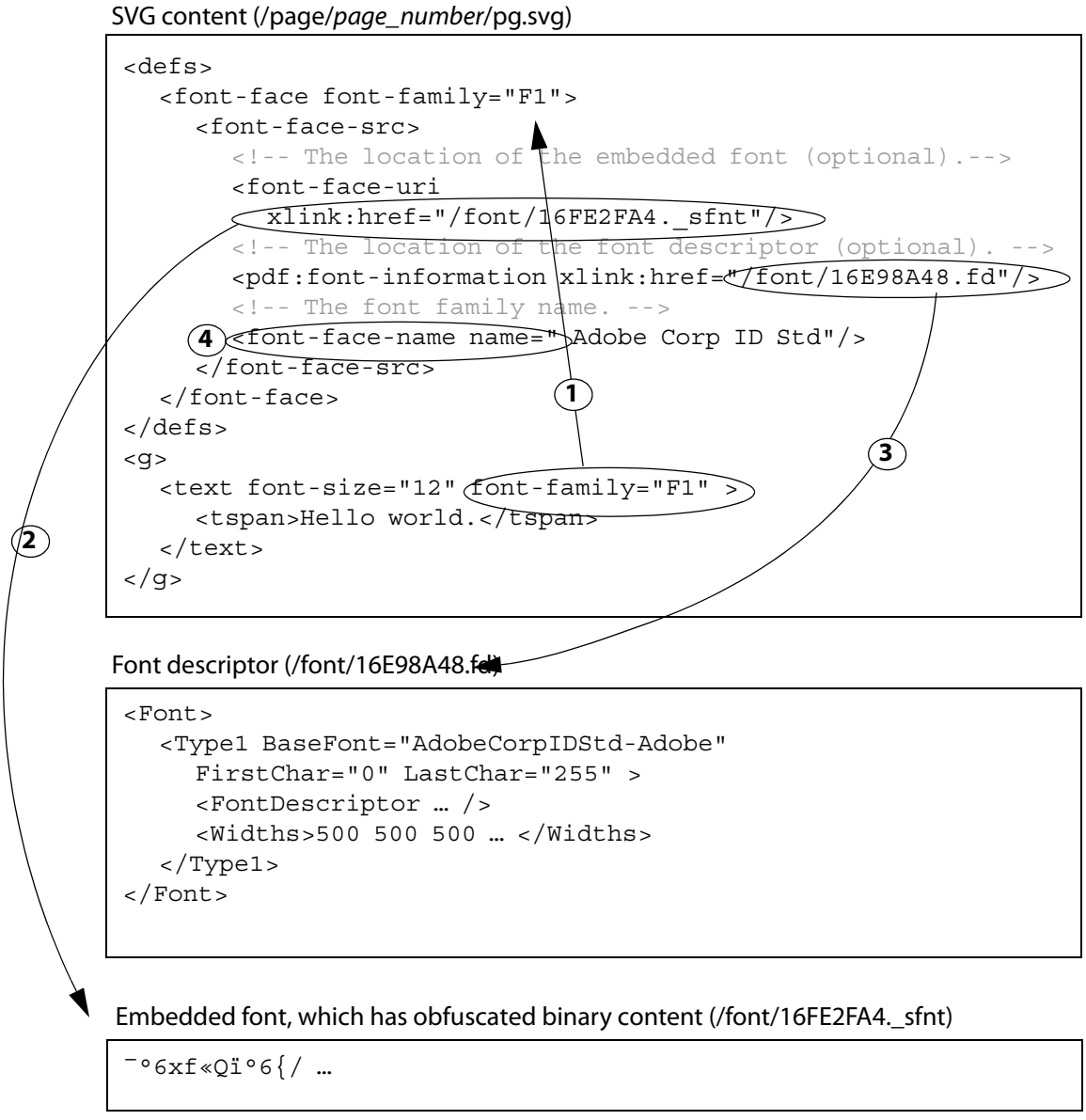
Note: Mars does not support fonts that use the Compact Embedded Font (CEF) format. Adobe Illustrator® can embed CEF fonts in SVG files. They use a compressed form of Type 1 and TrueType fonts that retain the hinting and kerning information from the original fonts.

Font-related structures

The font used to render text in a Mars document can be specified by the Mars document or can be left up to the default settings in the Mars viewer application. Fonts can be embedded in the Mars document, which guarantees the desired font is used.

The following illustration shows the relationship between the different structures used to specify and embed a font.

Structures used to represent font references and embedded fonts



1	References a `font-face` element within the SVG file's `defs` element.
2	References a file containing the embedded font. If this embedded font is present in the Mars document, the Mars viewer application uses it to render the text.

3	References a file containing a description of the font. If the font is not found on the system, the Mars viewer application uses the font description to find or synthesize a replacement font.
4	Provides the name of the font, which the Mars viewer application uses to determine whether the font is installed on the system. If it is, the application uses that font to render the text.
Else	If the font information described in 1-4 is not provided, the Mars viewer application uses an application-dependent replacement font to render the text, typically Times New Roman.

Font face specifiers

The SVG `font-face` element in the SVG definitions section contains a property that references the font descriptor. This property is present for embedded and non-embedded fonts.

Font descriptors

Font descriptors are XML files that are stored in separate package files. They are provided regardless of whether the font is embedded. By convention, these files are placed in the `/font` directory or the directory for the page in which the font descriptor is referenced. Font descriptors contain information about the font in general, but do not contain instructions on drawing the font's glyphs.

The font descriptor contains information about the font, including bounding box, the character widths and the height of the font above and below the baseline. If the font is not embedded, the information in the font descriptor is used to find a replacement font. This information allows the document to be rendered with a layout similar to the original.

Note: The only reliable way to determine which fonts are used to render text is to examine the font-face references in SVG `text` or `tspan` elements. The presence of a font or font descriptor in the package does not guarantee that the font is actually used.

Permissions in embedded fonts

Fonts embedded in the Mars file that can be used by forms during editing must have editable embedding permissions. In general, font embedding permissions and guidelines should be closely followed by all applications in order to avoid copyright infringement. For details, see *Font Embedding Guidelines for Adobe Third-Party Developers*, which is available at www.adobe.com/go/devnet.

Protection of embedded fonts

Embedded font files are obfuscated to discourage their use outside the document. Such fonts should not be removed from the package or used by other documents.

Referencing and embedding fonts

Referencing and embedding fonts involves the following general steps:

- Planning
- Defining font face specifiers, which includes creating font-face declarations on the SVG pages that use the font
- Establishing the font characteristics to apply to text

- Specifying a font descriptor
- Embedding OpenType fonts, which includes checking permissions and adding the obfuscated font to the Mars package

Planning

Decide which fonts you want to use and where they should appear in the document.

If space is not a consideration, embedding a font is the best way to ensure consistent results. If you decide to embed a font, ensure you have the rights to embed them.

Care should be taken to avoid multiple copies of the same information in a given document because font files and font descriptors tend to be large.

- Font descriptors. If multiple pages in your Mars document reference the same font, create a single font descriptor file located in the /font directory.
- Embedded SVG font files. If an SVG font is used on multiple pages, a single copy of that font should be saved in the Mars package as a separate file located in the /font directory. This placement avoids duplicating the same font in multiple SVG files.
- Embedded OpenType font files. If an OpenType font is used on multiple pages and the font is to be embedded, a single copy of that font should be located in the /font directory.

Defining font face specifiers

For each font used in the Mars document, the SVG `defs` element contains a font face specifier that provides the font family name and a link to a font descriptor file.

The following steps explain how to define a font face specifier in the SVG definitions:

1. In the SVG file, add a `font-face` element, as a child of the `defs` element.
2. In the `font-face` element, add a `font-family` attribute whose value is a string that uniquely identifies the font face specifier. This attribute and value are repeated in the text content elements that use the font described by this font face specifier. SVG text content elements include the `text` and `tspan` elements.
3. Add a `font-face-name` element as a child of the `font-face` element. In the `font-face-name` element, add a `name` attribute that provides the font family name. The Mars viewer application uses this name to search the system for correspondingly named fonts.
4. If the font is embedded, add a `font-face-src` element as a child of the `font-face` element. Then create a `font-face-uri` as a child of the `font-face-src`, and set its `xlink:href` attribute to specify the location of the embedded font.
5. Add a `pdf:font-information` element as a child of the `font-face-src` element. In the `pdf:font-information` element, add an `xlink:href` attribute that provides the location of the font descriptor. The Mars viewer application uses this definition to find or create a replacement font. See "Specifying a font descriptor" on page 40.

Note: The Mars document must contain embedded copies of any SVG fonts it uses.

Note: Identically named fonts can have different appearances. The only way to guarantee the appearance of a font is to embed that font in the Mars package. A font can be embedded only if the font's embedding permissions allow it.

The following is an example of a font specifier for a non-embedded font.

Example 4.1 ***Font specifier for a non-embedded font***

```
<svg>
   <defs>
      <font-face font-family="F1">
         <pdf:font-information xlink:href="/font/16E98A48.fd"/>
         <font-face-name name="Adobe Corp ID Std"/>
      </font-face>
   </defs>

   <g>
      <text font-size="12" font-family="F1">
         <tspan>Hello world.</tspan>
      </text>
   </g>
</svg>
```

The following is an example of a font specifier for an embedded font.

Example 4.2 ***Font specifier for an embedded font***

```
<svg>
   <defs>
      <font-face font-family="F1">
         <font-face-src>
            <font-face-uri xlink:href="/font/16FE2FA4._sfnt"/>
         </font-face-src>
      </font-face>
   </defs>

   <g>
      <text font-size="12" font-family="F1">
         <tspan>Hello world.</tspan>
      </text>
   </g>
</svg>
```

Establishing the font characteristics to apply to text

The following steps explain how to associate text with a font face and font characteristics. These steps apply equally to SVG and OpenType fonts.

1. In the `text` element that contains the characters you want to associate with the font, add the `font-family` attribute, setting its value to the unique identifier specified in the font face specifier created in "Defining font face specifiers" on page 38.

2. Also in the text element, set other font characteristics as needed, such as the following:

   - font-weight attribute, setting its value to the weight (thickness) component of the fully-qualified font name or font specifier. For example:

```
font-weight = "400"
```

- font-style attribute, setting its value to the style of the font. For example:

```
font-style = "normal"
```

See the *Mars Reference* for a more complete list of the SVG expressions supported in Mars.

Note: Mars viewer applications do not perform ligature formation or glyph substitutions beyond those specified in the SVG `altGlyphs` attribute. This attribute is a property of the `text` and `tspan` elements.

Specifying a font descriptor

Font descriptors contain information about the font, including the bounding box, the character widths and the height of the font above and below the baseline. If the font is not embedded and cannot be found on the system, the information in the font descriptor is used to find a replacement font.

To create a font descriptor, perform the following steps:

Note: The Adobe Font Developer Kit can be used to examine the tables in an OpenType font. See www.adobe.com/go/opentype.

1. Create an XML file with the suffix "`fd`", located in the /font directory or in the directory of the page on which they are referenced. Ensure the standard XML declaration appears on the first line:

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

2. Create a `Font` element as the root element in the font descriptor file.

3. Depending on the type of OpenType font being described, create a `Type0`, `Type1`, or `TrueType` element, as a child of the `Font` element. Only one of these elements may be present in the font descriptor file.

4. Add attributes to the `Type0`, `Type1`, `Type3`, or `TrueType` element, using the following table to determine the correlation between OpenType entries and attribute values:

Attribute name	Source of value for each type of font house in an OpenType font
`Name`	Omit this attribute. This attribute corresponds to an obsolete entry in the PDF font dictionaries.
`BaseFont`	All types: Use the OpenType `name` table's string 6 property.
`FirstChar`	All types: Use the OpenType `OS/2` table's `usFirstCharIndex`, converted to a decimal value
`LastChar`	All types: Use the OpenType `OS/2` table's `usLastCharIndex`, converted to a decimal value

5. Create a `Widths` element as a child of the `Type0`, `Type1`, `Type3`, or `TrueType` element. In this element, create one width entry for each of the characters described by the font. The widths correspond to the entries in the OpenType horizontal metrics (`hmtx`) table's `mtx/@width` entries. Each character has an `mtx` entry in this table.

6. Create a `FontDescriptor` element as a child of the `Type0`, `Type1`, `Type3`, or `TrueType` element. Use the following table to determine the correlation between OpenType entries and attribute values.

Attribute name	Source of value for each type of font house in an OpenType font
`FontName`	All types: Use the OpenType `name` table's string 6 property.
`FontFamily`	All types: Use the OpenType `name` table's string 16 property. If that property is missing, use that table's string 1 property.
`FontStretch`	All types: Use the OpenType `OS/2` table's `usWidthClass` property
`FontWeight`	All types: Use the OpenType `OS/2` table's `usWeightClass` property
`Flags`	All types: Use application-dependent values. Acrobat determines the `Symbolic` and `Serif` settings by measuring characteristics of rendered glyphs. **Note:** Acrobat ignores the flags `FixedPitch`, `Script`, `Italic`, `AllCap`, `SmallCap`, and `ForceBold`.
`FontBBox`	All types: Use the OpenType `head` table's properties: `xMin`, `yMin`, `xMax`, and `yMax`
`ItalicAngle`	• For Type 0, 1 and 3 fonts, use the OpenType `CFF` table's `CFFFont/Private/@ItalicAngle` property, which appears in the Type 1 font's `Private` dictionary. • TrueType fonts, use the OpenType name table's `post/ItalicAngle`
`Ascent`	All types: Use the OpenType `OS/2` table's `usWinAscent` property
`Descent`	All types: Use the OpenType `OS/2` table's `usWinDescent` property
`Leading`	All types: Use the OpenType `OS/2` table's `sTypoLineGap` property
`CapHeight`	All types: OpenType `OS/2` table's `sCapHeight` property
`XHeight`	All types: OpenType `OS/2` table's `sxHeight` property
`StemV`	• For Type 0, 1 and 3 fonts, Use the OpenType `CFF` table's `CFFFont/Private/@StdVW` property. • For TrueType, determine an application-dependent value. Acrobat determines this value by measuring characteristics of rendered glyphs.
`StemH`	• For Type 0, 1 and 3 fonts, use the OpenType CFF table's `CFFFont/Private/@StdHW` property. • For TrueType fonts, use 0 (zero). Such fonts omit this attribute.
`AvgWidth`	All types: OpenType `OS/2` table's `xAvgCharWidth` property
`MaxWidth`	All types: Use an application-dependent value. Acrobat does not use this attribute.

Attribute name	Source of value for each type of font house in an OpenType font
`MissingWidth`	All types: Use an application-dependent value. Acrobat does not use this attribute.
`CharSet`	A string listing the character names defined in a font subset. This attribute is used only for embedded subsets.

Note: The most important piece of information in the font descriptor is the value of the font's `Widths` array.

The following example creates a font descriptor for an OpenType version of a Type1 font. The base font name is `CourierStd`. This name omits a face type, which means the font represents the Roman face. The font family name is `Courier Std`. Each character width is the same, indicating the font is mono-spaced.

Example 4.3 ***Font descriptor file contents (/font/name.fd or /page/page_number/name.fd)***

```
<Font>
  <Type1 LastChar="255" BaseFont="CourierStd" FirstChar="0">
    <FontDescriptor StemV="0" FontName="CourierStd" FontStretch="Normal"
      FontWeight="500" Flags="Serif Nonsymbolic" Descent="-195"
      FontBBox="-56 -250 678 857" Ascent="631" FontFamily="Courier Std"
      CapHeight="576" XHeight="437" ItalicAngle="0"/>
    <Widths>600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600
600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600
600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600
600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600
600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600
600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600
600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600
600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600
600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600
600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600
600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600
600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600
600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600 600
600</Widths>
  </Type1>
</Font>
```

Embedding OpenType fonts

The section describes tasks related to embedding OpenType fonts.

Fonts embedded in a Mars document are for the exclusive use of the document. Such fonts cannot be removed from the Mars document and used in other documents or for other purposes.

Checking font permissions

Before embedding a font in a Mars document, you should ensure the font embedding permissions and guidelines are respected. Violating these permissions and guidelines may result in copyright infringement.

Fonts embedded in a Mars file that can be used by forms during editing must have editable embedding permissions.

For more information, see *Font Embedding Guidelines for Adobe Third-party Developers* available at http://www.adobe.com/go/acrobat_developer. (Click on "Working with Acrobat Distiller, fonts, and more" in the *Advanced user resources* panel.)

Determining the tables to include

An embedded OpenType font must contain certain tables depending on the font characteristics, as follows:

OpenType characteristics	Required OpenType tables
TrueType font dictionary or a CIDFontType2 CIDFont dictionary that contains a "glyf" table	"glyf", "head", "hhea", "hmtx", "loca", and "maxp". The "cvt " (notice the trailing space), "fpgm", and "prep" tables must also be included if they are required by the font instructions.
CIDFontType0 CIDFont dictionary that contains a "CFF " table (notice the trailing space) with a Top DICT that uses CIDFont operators (this is equivalent to subtype CIDFontType0C above)	"CFF ", "cmap"
Type1 font dictionary or CIDFontType0 CIDFont dictionary, if the embedded font program contains a "CFF " table without CIDFont operators	"CFF ", "cmap"

Note: The absence of some optional tables (such as those used for advanced line layout) may prevent editing of text containing the font.

Obfuscating the font

Mars requires all embedded font files to be obfuscated, using an algorithm that modifies the bits in the first 1024 bytes of the font file.

Font obfuscation involves the following steps:

1. Obtain the value of the `DocumentID` attribute from the document backbone. If this attribute is missing or has a null string value, fonts cannot be embedded in the Mars document.

2. Construct an obfuscation key, by converting the `DocumentID` attribute string into an array of bits. The string should be treated as a sequence of bytes, where each byte is interpreted as a sequence of eight bits with the high-order (most significant) bit listed first. Use only the first 512 bits of this value.

3. Exclusive OR the first 1024 bytes of the font data with the obfuscation key. The bytes that comprise the obfuscation key are repeated as many times as it takes to exclusive OR the first 1024 bytes of font data, as shown in Example 4.4. The font data is processed as a sequence of bytes in the same manner as the `DocumentID` attribute string in step 2.

4. The obfuscated data is written out to a package file. This file is referenced from font specifiers, as described in "Defining font face specifiers" on page 38

5. If the font data contains more than 1024 bytes, subsequent bytes are written out to the file, without modification.

This algorithm is symmetric. Applying steps 1-3 on obfuscated font data produces the original font data.

The C code segment (Example 4.4) illustrates the font obfuscation algorithm described above. The following table describes the variables used in this segment.

Variable	Type	Description
`data`	`unsigned char*`	Pointer to a buffer containing the original or obfuscated font data read
`dataSize`	`size_t`	Number of bytes of data
`fontKey`	`unsigned char *`	An array of bytes representing the obfuscation key. Step 2 on page 43 explains how to derive this key.
`fontKeySize`	`size_t`	Number of bytes of data in `fontKey`
`i`	`size_t`	Current position in the data buffer

The obfuscate function obfuscates or deobfuscates data in place. If the font data is larger than 1024 bytes, the remaining bytes are not modified.

Example 4.4 ***Applying obfuscation to a font***

```
void obfuscate(
       unsigned char* data,
       size_t dataSize,
       unsigned char* fontKey,
       size_t fontKeySize)
{
   for (size_t i = 0; i < dataSize && i < 1024; ++i)
      data[i] ^= fontKey[i % fontKeySize];
}
```

Converting an existing font to the OpenType font format

The Adobe Font Development Kit for OpenType (AFDKO) allows you to build an OpenType font from an existing font, verify the contents of the font, and proof the font on screen or on paper. This tool and information related to OpenType fonts is available from www.adobe.com/go/opentype.

5 Adding Images to SVG Content

This chapter describes how to specify images in Mars documents.

This chapter contains the following information.

Topic	Description	See
Understanding images and color profiles	Explains what images are and what image file types are supported.	page 46
Using images	Describes how to create color profiles and associate them with images.	page 47

Understanding images and color profiles

In Mars documents, images reside in separate files that are referenced from the individual pages that use them. If a page references an image that is color-managed, that page also defines the ICC color profile to apply to the image.

About images

A bitmap image is a sequence of samples obtained by scanning the image array in row or column order. Each sample in the array consists of as many color components as are needed for the color space in which they are specified.

Sampled images contain sequences of graphic operators that are used to paint the images for printing and display purposes. Each image is defined as a rectangular array of sample values, each representing a color. The image may approximate the appearance of some natural scene obtained through an input scanner or a video camera, or it may be generated synthetically.

The representation of the scanned information and the compression applied to the images are defined by the particular image format.

Image information

Mars represents images using standard formats. In general, all of the image information should be stored within the image file itself. The exceptions are use of “sidecar” files for XMP metadata and use of external color profiles.

Supported image formats

Images used in a Mars document must be included in the Mars package. That is, remote references to images are not supported.

A single image file in a Mars document can be used on multiple pages, with each destination page having a reference to the image file.

The following table lists the image formats supported in Mars and their correspondence in PDF. The information in the column PDF compression filter is intended for use by developers who are familiar with PDF internals.

Mars image type	PDF compression filter
JBIG2	JBIG2Decode
JPEG	DCTDecode
JPEG2000	JPXDecode
PNG	other + indexed color
PNG	other + grayscale color
PNG	other + RGB color

Note: Data used in Mars JPEG2000 images should be limited to the JPX baseline set of features.

About color profiles

Color profiles can be used to get the correct color reproduction when you input images from a scanner or camera and display them on a monitor or print them. They define the relationship between the digital counts your device receives or transmits and a standard color space defined by ICC and based on a measurement system defined internationally by CIE. [http://color.org]

A number of non-ICC based color spaces are used in PDF. Although you can use these in creating Mars documents, they are typically only used when converting PDF documents that already use them to Mars format.

The non-ICC-based color spaces include DeviceGray, DeviceRGB, DeviceCMYK, and Indexed, Pattern, Separation, and DeviceN. For each sample, depending on the color space, one component would be needed for DeviceGray, three for DeviceRGB, and four for DeviceCMYK. For more information on color spaces, see "" on page 51.

Using images

This section explains how to use images in Mars documents.

Specifying color profile definitions for images

Color-managed images are associated with ICC color profiles. These profiles can be defined in the `image` element that references the images, and for certain types of image formats, these profiles can be included in the image itself. Mars processing applications use the following sequence to determine which ICC profile to use on an image:

1. ICC color profiles specified in the `image` element.
2. ICC color profiles specified in the image.
3. Otherwise, Mars processing applications use the internal color in the image file.

Of the supported image file formats, only JPEG, JPEG2000 and PNG image formats can contain embedded ICC color profiles. JBIG2 cannot contain an embedded ICC profile.

"" on page 51 explains how to create a color profile definition.

Adding an image to a page

To add an image to a page in the Mars document, perform the following steps:

1. Add the image file to the Mars document package. The image files must be located in separate files contained in the Mars package. By convention, these files are placed in the image or page directory, as follows:
   - Place image files used only on a particular page in the /page/*page-number* directory, where *page-number* represents the page number on which the image is used.
   - Place image files used on multiple pages in the /image directory.
2. Open the pg.svg file for the page on which the image will be displayed.
3. Create an `image` element at an appropriate place in the SVG graphic content. Specify the following attributes for the `image` element:
   - Set the `xlink:href` attribute to the location of the image within the document package. It is best to use a relative URI to reference the image.
   - Set the `height` and `width` attributes to the number of samples that are obtained by scanning the image array in row and column order. The image will be clipped if it is larger than the `height` and `width` set.
   - Specify where to place the image on the page. For example, set the `x` and `y` attributes to coordinates that specify the position on the page where the upper-left corner of the image is placed.
   - If the image has a color profile, set the `color-profile` attribute to the name of the ICC profile to apply to the image. The name is specified in the ICC color profile definition. (See "Specifying color profile definitions for images" on page 47.)

Note: The resolution of images should be set appropriately for the intended use of the Mars document. Typically, the resolution of images to be printed is higher than the resolution of images to be viewed online.

The following example defines two images. The first image is associated with a the color profile named `cs-1`, and the second with the color profile named `cs-2`. Both of these color profiles are indexed profiles that use a look-up table to reference the ICC color profile named `cs-0`. The `cs-0` color profile is contained in the cs-0.icc file located in the /color directory.

Example 5.1 ***Associating a color profile with an image (/page/page-number/pg.svg)***

```
<svg>
   <defs>
      <color-profile name="cs-0"
         xlink:href="/color/cs-0.icc" pdf:Count="3"/>
      <pdf:Indexed Name="cs-1"
         pdf:Base="cs-0" pdf:HiVal="54" LookupTableSrc="/color/lut-0.lut"/>
      <pdf:Indexed Name="cs-2" pdf:Base="cs-0"
         pdf:HiVal="55" LookupTableSrc="/color/lut-1.lut"/>
      <!-- Other definitions go here.-->
   </defs>
   <g>
      <image xlink:href="/page/0/red-ball.png"
         width="297.4031" height="39.54"
         x="72" y="708.06"
         color-profile="cs-1"/>
      <image xlink:href="/page/0/sunset.png"
         width="297.4031" height="39.54"
         x="72" y="668.52"
         color-profile="cs-2"/>
   </g>
</svg>
```

6 Specifying Color Spaces

This chapter describes how to specify ICC and PDF color spaces.

This chapter contains the following information.

Topic	Description	See
Understanding color spaces	Describes the role of color spaces in Mars documents and describes the structures that specify color spaces.	page 50
	Explains how to associate color spaces with graphics or text	page 51

Understanding color spaces

A color space is a system that can be used to specify abstract colors in a device-independent way. Some color spaces are related to device color representation (grayscale, RGB, CMYK), while others are more related to human visual perception (XYZ). ICC profiles are often separate package files referred to by the SVG page content. ICC color profiles can also appear inside image files themselves.

The use of ICC profiles to represent color spaces is recommended for both SVG and Mars. Since ICC profiles cannot cover all of the facilities defined by PDF, custom namespace extensions are defined for use by Mars documents to support other types of PDF-supported color spaces.

Within SVG page content, color space definitions are placed in the document's `svg:defs` section. These definitions may refer to external ICC profile files or other relevant files. The resources external to the SVG file must be elsewhere in the document package. Color space definitions can also appear in resource files that reside elsewhere in the document package. You should place the color-managed content corresponding to the color space definitions in the /color directory.

The color space definitions in an SVG file must be defined before they are referenced. Color spaces can be referenced from various elements and attributes defined by SVG, including the attributes `fill`, `stroke`, and `stop-color`. Color spaces can also reference other color spaces that their definition depends on.

Color spaces can be shared among multiple pages. For ICC profiles, the elements need to reference the same `.icc` file. For more complex definitions, move the color space definition to a separate file and then reference it from the desired element.

Formats used with color space type

The following table shows how color spaces are defined and referenced.

Device Gray

Topic	
Appearance in `defs` section of SVG	None.
Format for a color definition in this color space	`"rgb(rgb-equivalent)` `device-color(DeviceGray, intensity-value)"`

Example of a color declaration in this color space	`<circle cx="600" cy="200" r="100"`
	`stroke-width="10"`
	`fill="rgb(86,86,86)`
	`device-color(DeviceGray,.6625)"`
	`stroke="rgb(255,255,255)`
	`device-color(DeviceGray,.9855)" />`

DeviceRGB

Topic	
Appearance in `defs` section of SVG	None
Format for a color definition in this color space	`rgb(r-value, g-value, b-value)`

Example of a color declaration in this color space	`<circle cx="600" cy="200" r="100" stroke-width="10"`
	`fill="rgb(86,23,86)"`
	`stroke="rgb(#F0A02B)" />`

DeviceCMYK

Topic	
Appearance in `defs` section of SVG	None
Format for a color definition in this color space	`"rgb(rgb-equivalent)` `device-color(DeviceCMYK, c, m, y, k)"`

Example of a color declaration in this color space	`<circle cx="600" cy="200" r="100" stroke-width="10"`
	`fill="rgb(12,91,255)`
	`device-color(DeviceCMYK,.89,.62,0,0)"`
	`stroke="rgb(120,232,255)`
	`device-color(DeviceCMYK,.89,.62,4,34.5)" />`

CalGray

Topic	

Appearance in `defs` section of SVG	`<svg:defs>`
	`  <pdf:CalGray Name="cs-1"`
	`  pdf:WhitePoint="1.0 1.0 1.0"`
	`  pdf:BlackPoint="0 0 0" pdf:Gamma="23.4" />`
	`...`
	`</svg:defs>`
	Or in a separate resource file, /page/23/res.xml:
	`<Resources>`
	`  <CalGray id="cs-23" WhitePoint="1.0 1.0 1.0"`
	`  BlackPoint="0 0 0" Gamma="23.4" />`
	`...`
	`</Resources>`

Format for a color definition in this color space	`"rgb(rgb-equivalent)` `  device-color(cs-id, intensity-value)"`

Example of a color declaration in this color space	`<circle cx="600" cy="200" r="100" stroke-width="10"`
	`  fill="rgb(86,86,86) device-color(cs-1,.6625)"`
	`  stroke="rgb(86,86,86)`
	`  device-color(url(/page/23/res.xml#cs-23),.6625)" />`

CalRGB

Topic	

Appearance in `defs` section of SVG	`<svg:defs>`
	`  <svg:color-profile name="cs-8"`
	`  xlink:href="alphagraphics.icc"`
	`  pdf:WhitePoint="0 0 0"`
	`  pdf:BlackPoint="2 3 4" pdf:Gamma="1 2 3" />`
	`  ...`
	`</svg:defs>`

Format for a color definition in this color space	`"rgb(rgb-equivalent) icc-color(cs-id, r-value,` `g-value, b-value)"`

Example of a color declaration in this color space	`<circle cx="600" cy="200" r="100" stroke-width="10"`
	`  fill="rgb(86,86,86) device-color(cs-8,.6625)"`
	`  stroke="rgb(86,86,86)`
	`  device-color(url(/page/23/res.xml#cs-23),.6625)" />`

Lab

Topic	

Appearance in `defs` section of SVG	`<svg:defs>`
	`  <svg:color-profile name="cs-2"`
	`  xlink:href="betaprints.icc"`
	`  pdf:WhitePoint="0 0 0" pdf:BlackPoint="1 1 1"`
	`  pdf:Range="0 1 0 1" />`
	`</svg:defs>`

Format for a color definition in this color space	`"rgb(rgb-equivalent) icc-color(cs-id, l-value, a-value, b-value)"`

Example of a color declaration in this color space	`<circle cx="600" cy="200" r="100" stroke-width="10"`
	`  fill="rgb(246,177,153)`
	`  icc-color(cs-2,80,20,20)"`
	`  stroke="rgb(246,177,153)`
	`  icc-color(cs-2,80,20,20)" />`

ICCBased

Topic	

Appearance in `defs` section of SVG	`<svg:defs>`
	`  <svg:color-profile name="cs-9"`
	`  xlink:href="alphagraphics.icc"`
	`  pdf:Count=" 3" pdf:Range="0 1 0 1 0 1"/>`
	`  ...`
	`</svg:defs>`

Format for a color definition in this color space	`"rgb(rgb-equivalent)` `  icc-color(cs-id, value1, value2, valuen)"`

Example of a color declaration in this color space	`<circle cx="600" cy="200" r="100" stroke-width="10"`
	`  fill="rgb(246,177,153)`
	`  icc-color(cs-9,.4,.6,.1)"`
	`  stroke="rgb(246,177,153)`
	`  icc-color(cs-9,.4,.6,.1)" />`

Indexed

Topic	

Appearance in `defs` section of SVG	`<svg:defs>`
	`  ...`
	`  <pdf:Indexed Name="cs-1"`
	`  pdf:Base="cs-2" // cs-2 must be defined earlier`
	`  pdf:HiVal="3" pdf:Gamma="23.4" >`
	`  <pdf:LookupTable src="/color/cs-1.lut"/>`
	`  </pdf:Indexed>`
	`  ...`
	`</svg:defs>`
	
	Or in a separate resource file, /page/p23/res.xml:
	
	`<Resources>`
	`  <Indexed id="cs-54"`
	`  pdf:Base="cs-2" // cs-2 must be defined earlier`
	`  pdf:HiVal="3" pdf:Gamma="23.4" >`
	`  <LookupTable src="/color/cs-1.lut"/>`
	`  </Indexed>`
	`  ...`
	`</Resources>`

Format for a color definition in this color space	`"rgb(rgb-equivalent)` `  device-color(cs-id, index)" ???`

Example of a color declaration in this color space	`<circle cx="600" cy="200" r="100" stroke-width="10"`
	`  fill="rgb(246,177,153) device-color(cs-1,34)"`
	`  stroke="rgb(246,177,153)`
	`  device-color(url(/page/23/res.xml#cs-54),2)" />`

Pattern

Topic	

Appearance in `defs` section of SVG	`<svg:defs>`
	`  ...`
	`  <pdf:TilePattern Name=”pcs-1” pdf:PaintType=”1”`
	`  pdf:TilingType=”1”`
	`  src="/res/t1_3044082663-403.svg"`
	`  pdf:BBox=”0 0 234 180”`
	`  pdf:XStep=”2” pdf:YStep=”4”`
	`  Matrix="-0.3 0 0 -0.3 365.625 414.375">`
	`  </pdf:TilePattern>`
	`  ...`
	`</svg:defs>`
	Or in a separate resource file, /page/p23/res.xml:
	`<Resources>`
	`  ...`
	`  <ShadePattern id=”pcs-23” Shader=”sha-32”`
	`  Matrix="0.2 0.2 -0.2 0.2 323.3 725.9">`
	`  <ExtGState OverprintMode="1" Overprint="false"`
	`  OverprintPaint="false" StrokeAdjust="false"`
	`  SmoothnessTolerance="0.002"/>`
	`  </ShadePattern>`
	`  ...`
	`</Resources>`

Format for a color definition in this color space	`Uncolored Type 1 Patterns: device-color(cs-id, val1, val2, ... valn)`
	`[where n is the number of components in the base colorspace]`
	`Colored Type 1 Patterns: device-color(cs-id)`
	`Type 2 Patterns: device-color(cs-id)`

Example of a color declaration in this color space	`"device-color(cs-1,.33,.25,0,1)"`
	`  <circle cx="600" cy="200" r="100" stroke-width="10"`
	`  fill="rgb(246,177,153) device-color(pcs-1)"`
	`  stroke="rgb(246,177,153)`
	`  device-color(url(/page/23/res.xml#pcs-23))" />`

Separation

Topic	

Appearance in `defs` section of SVG	`<svg:defs>`
	`  ...`
	`  <pdf:Separation Name=”scs-1”`
	`  pdf:Colorant=”Spot1”`
	`  pdf:AlternateSpace=”cs2”`
	`  pdf:TintTransform=”func2” >`
	`  </pdf:Separation>`
	`  ...`
	`</svg:defs>`
	Or in a separate resource file, /page/23/res.xml:
	`<Resources>`
	`  ...`
	`  <Separation id=”scs-54”`
	`  Colorant=”Spot1”`
	`  AlternateSpace=”cs2” // these must appear in the same file`
	`  TintTransform=”func2” >`
	`  </Separation>`
	`  ...`
	`</Resources>`

Format for a color definition in this color space	`"rgb(rgb-equivalent)` `  device-color(cs-id, tint-intensity-value)"`

Example of a color declaration in this color space	`<circle cx="600" cy="200" r="100" stroke-width="10"`
	`  fill="rgb(246,177,153) device-color(scs-1, 0.7)"`
	`  stroke="rgb(246,177,153)`
	`  device-color(url(/page/23/res.xml#scs-54),0.5)" />`

DeviceN

Topic	

Appearance in `defs` section of SVG	`<svg:defs>`
	`...`
	`<pdf:DeviceN Name="ncs-1"`
	`Colorants="Spot1 Spot2 Spot3"`
	`AlternateSpace="cs-23" // must be defined in same file`
	`TintTransform="func-98"> // must be defined in same file`
	`<pdf:Attributes ... />`
	`</pdf:DeviceN>`
	`...`
	`</svg:defs>`
	Or in a separate resource file, /page/p23/res.xml:
	`<Resources>`
	`...`
	`<DeviceN id="ncs-16"`
	`Colorants="Spot1 Spot2 Spot3"`
	`AlternateSpace="cs-23" // must be defined in same file`
	`TintTransform="func-98"> // must be defined in same file`
	`<Attributes ... />`
	`</DeviceN>`
	`...`
	`</Resources>`

Format for a color definition in this color space	`"rgb(rgb-equivalent)` `device-color(cs-id, val1, val2, ... valn)"`

Example of a color declaration in this color space	`"rgb(0,79,163) device-color(cs-1,.4,0,0,.75)`
	`<circle cx="600" cy="200" r="100" stroke-width="10"`
	`fill="rgb(0,79,163) device-color(ncs-1,.4,0,.75)"`
	`stroke="rgb(246,177,153)`
	`device-color(url(/page/23/res.xml#ncs-16),.4,0,.7)"`
	`/>`

Defining and using a color space for text and graphic content

Color profiles must be assigned to color-corrected images that do not contain internal color profiles.

Defining named references to color profiles involves adding elements to the definitions that appear in the SVG file that uses the graphic. For each color profile, the additional elements identify the color space and provide a pointer to the files that contain the color space profiles. They can also specify a convenient name that serves as a global nickname for the color space.

Defining color profiles in SVG

To define a name that references a particular ICC color profile, modify the SVG file that contains the text, graphics or images for which ICC profiles are defined*:*

1. Create the `SVG:color-profile` element as a child of the `defs` element.
2. Set the `color-profile` element's `name` attribute to identify the color profile to use for the color specification.
3. Set the `color-profile` element's `xlink:href` attribute to the location of the ICC profile resource for the image. References to color profiles are specified as a URI or simple color space name. When referencing a color profile using a URI, the color profile definition can appear in a file other than the current SVG file. When referencing a color space using a simple name, the color space must be defined in the `defs` element of the referencing SVG file.
4. Set the `color-profile` element's `pdf:Count` attribute to the number of color components in the color space described by the ICC profile data. This number must match the number of components in the ICC profile. The possible values for `pdf:Count` are `1`, `3`, or `4`.

The following example specifies two color spaces and defines names by which these color spaces can be referenced in subsequent text, graphic, or image elements. The first color profile is defined as an `svg:color-profile` element with the name attribute set to `cs-0`. The `xlink:href="/color/cs-0.icc"` declaration represents a unique URL that specifies the location of the ICC color profile.

Example 6.1 ***Specifying color profiles (/page/page-number/pg.svg)***

```
<defs>
     <color-profile name="cs-0"
        xlink:href="/color/cs-0.icc"
        pdf:Count="3"/>
  …
</svg>
```

Using ICC color spaces to specify color for text or graphic objects

Associating a named color profile with an SVG text or graphic elements involves defining the ICC profile in the `fill` or `stroke` attribute for the object. This object must be of type `paint`.

The following expression describes the paint type (SVG Specification). This section addresses only the form identified in bold characters. Mars does not support URI expressions in a paint type, nor does it support float values in the `icccolorvalue` subtype.

```
none |
currentColor |
<color> [icc-color(<name>[,<icccolorvalue>]*)] |
<uri> [none | currentColor |
   <color> [icc-color(<name>[,<icccolorvalue>]*)]] |
inherit
```

To associate a named color profile with an existing SVG text or graphic object, modify the SVG file as follows.

1. Locate an existing `text` or `graphic` element.
2. Set the text or graphic element's `fill` or `stroke` attribute to specify a device color value and a color in the ICC profile color space.

The following example uses the `cs-0` color space defined in the previous example being used in the `icc-color` function. All the color values specified via the `fill` and `stroke` attributes include both the `RGB` equivalent for the color value and an `icc-color` function value that contains the actual color values that are used on import. In this example, the `fill` and `stroke` attributes are set to an RGB value of `(246,177,153)`. This means the red value is `246`, the green value is `177`, and the blue value is `143`.

Example 6.2 ***Associating a color space with a graphic element (/page/page-number/pg.svg)***

```
<circle cx="600" cy="200" r="100" stroke-width="10"
   <!-- Specifies an RGB color used to fill the circle.
     The color is converted using the ICC color profile named "cs-0". -->
   fill="rgb(246,177,153) icc-color(cs-0,.4,.6,.1)"
   <!-- Specifies an RGB color used as the outline for the circle.
     The color is converted using the ICC color profile named "cs-0". -->
   stroke="rgb(246,177,153) icc-color(cs-0,.4,.6,.1)" />
```

Associating non-ICC color spaces with text or graphic objects

Mars defines other color spaces that do not directly include an ICC profile, such as device color spaces and indexed color spaces.

Device color spaces

For color spaces without an ICC profile, the color value is specified via the `device-color` function. The format for this function is similar to the `icc-color` function.

The DeviceCMYK, DeviceRGB and DeviceGray color spaces do not require any supplemental information to be stored in a color space element, therefore the predefined names DeviceCMYK and DeviceGray can be used to define the color space.

CMYK is an unmanaged and unprofiled color space with no ICC equivalent. CMYK controls the concentrations of cyan, magenta, yellow, and black inks, the four subtractive process colors most commonly used in printing.

To associate a named color profile with an existing SVG text or graphic object, modify the SVG file as follows:

1. Locate an existing `text` or `graphic` element.
2. Set the text or graphic element's `fill` or `stroke` attribute to specify a device color value and reference a non-ICC-based color profile.

Example 6.3 ***Using a device color space (/page/page-number/pg.svg)***

```
<g transform="matrix(1 0 0 -1 0 792)">
   <text transform="matrix(1 0 0 -1 0 571.5643)"
      font-size="16.02"
      font-family="F1"
      fill="rgb(0,0,0) device-color(DeviceGray,0)"
      fill-rule="evenodd">
```

Indexed color spaces

Mars supports other non-ICC-based color spaces, such as L*a*b*, Indexed, Separation, and DeviceN. These color spaces will be described in a later release of this document.

Part II: Adding Interactive Features to a Mars Document

8 Creating Bookmarks

This chapter describes a form of document-level navigation called bookmarks. It describes how bookmarks are used and how to create them using explicit and named destinations.

This chapter contains the following information.

Topic	Description	See
Understanding bookmarks	Explains what bookmarks are and why they are used.	page 62
Specifying bookmarks	Explains how to create bookmarks.	page 63
Specifying explicit destinations	Explains how to create explicit destinations.	page 66
Specifying named destinations	Explains how to create named destinations.	page 67
Using named destination caches	Explains how to create named destination caches.	page 71

Understanding bookmarks

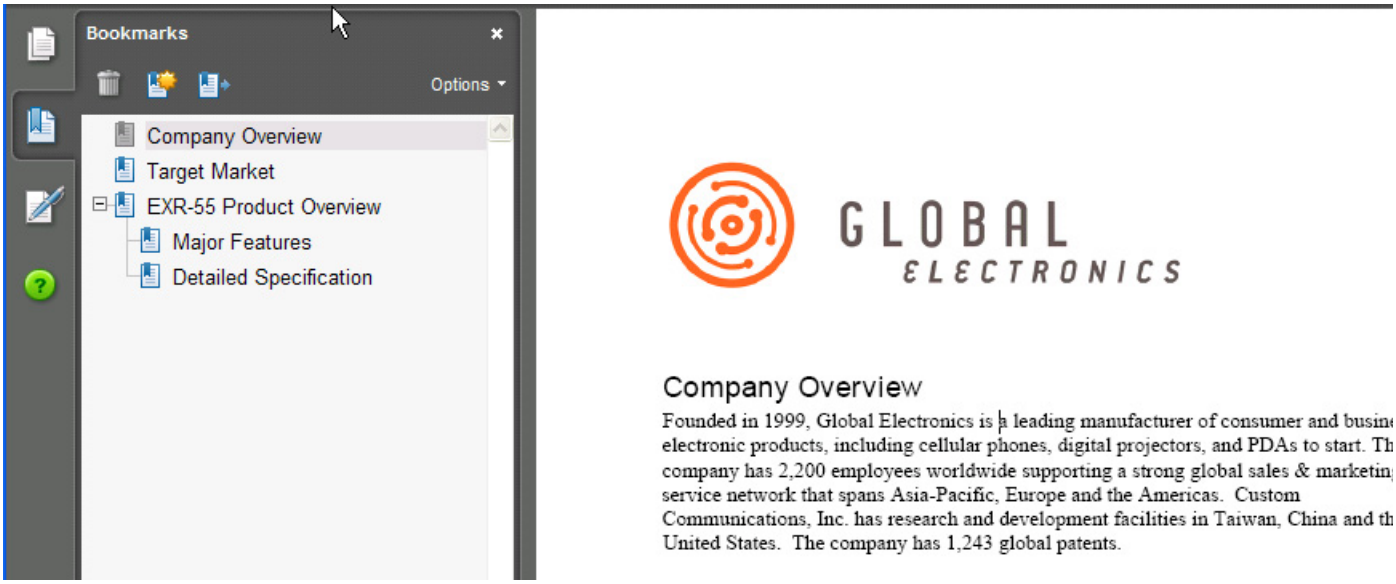
Bookmarks allow the user to navigate from one part of the document to another or to initiate an action such as bringing up a Web site in a browser window. Bookmarks are structured to represent the hierarchy of the document, which serve as a visual table of contents to display the document's structure to the user. Viewing applications typically display bookmarks in a special panel separate from the document. For more information about bookmarks, see the *PDF Reference*.

In Mars, bookmarks are in a separate package file referenced from the backbone. Conventionally, the file containing the bookmarks is named bookmarks.xml and is located in the root directory.

A single bookmark consists of a title (the text that is displayed) and the action to be performed when the bookmark is activated. Each bookmark may contain subordinate bookmarks that typically reflect the hierarchy of the document they represent.

The following illustration shows the appearance of an outline of bookmarks. There are three top-level bookmarks, the last of which has two child bookmarks. Although not apparent in the illustration, each bookmark is associated with a go-to action that changes the view to the part of the document associated with the title.

A simple hierarchy of bookmarks



Specifying bookmarks

Specifying bookmarks involves creating an entry in the backbone file that references the bookmarks file, and then creating the bookmarks file to describe the bookmarks in the Mars document.

Referencing the bookmarks file from the backbone

To create bookmarks for a Mars document, make the following modifications to the backbone file (backbone.xml):

1. Create a `Bookmarks` element, as a child of the `PDF` element.
2. Set the `Bookmarks` element's `src` attribute to reference the source file containing the bookmarks to be used by the document.

The example below shows the backbone file's reference to the bookmarks file.

Example 8.1 ***Specifying the location of the bookmarks file (backbone.xml)***

```
<PDF>
   <Bookmarks src="/bookmarks.xml" />
</PDF>
```

Creating the bookmarks file

To create the bookmarks file that describes the individual bookmarks, perform the following tasks:

1. Create the bookmarks.xml file in the root directory, ensuring the first line is the standard XML declaration `<?xml version="1.0" encoding="UTF-8"?>`.
2. Create the `Bookmarks` element, as the root element for the file, setting the default namespace to `http://ns.adobe.com/pdf/2006`.

3. Set the `Bookmarks` element `Open` attribute to specify whether the nested content is shown. A value of `true` specifies that nested bookmarks are shown, and a value of `false` specifies that nested bookmarks are hidden. The default value is `false`.

4. For the first top-level bookmark, create a `Bookmark` element as a child of the `Bookmarks` element.

5. Optionally, create the `Bookmark` element's `Styles` attribute, specifying a space-separated list of the styles to apply to the title. The defined values are `Italic` and `Bold`.

6. Optionally, create the `Bookmark` element's `Open` attribute, specifying whether the bookmark's nested content is shown.

7. Create an `Action` element as a child of the `Bookmark` element, populating that element with information about the desired action. The `Action` element can specify a location to jump to in the document or an action for the viewer application to perform, such as launching an application, playing a sound, or changing an annotation's appearance state. For more information on how to create actions, see "Understanding actions" on page 74.

8. Create a `Title` element as a child of the Bookmark element, specifying a title to use for the bookmark in the viewer application's bookmark panel.

9. For each bookmark that is a child of the freshly-created bookmark, create a corresponding child `Bookmark` element, recursively following steps 5-9. The order of the bookmarks at each level in the hierarchy establishes the order in which the bookmarks are displayed when the parent bookmark is open.

10. For each subsequent top-level bookmark, create a `Bookmark` element, as a child of the `Bookmarks` element. The order of the top-level bookmarks establishes the order in which those bookmarks are displayed.

The following example creates the bookmarks shown on page 63. This example specifies three top-level bookmarks, the last of which contains two children. Initially, the bookmarks are all open because of the `Open` attribute in the `Bookmarks` element. Each bookmark contains a go-to action whose destination is a part of the page associated with the title.

Example 8.2 ***Specifying bookmarks with actions (path name: /bookmarks.xml)***

```
<Bookmarks xmlns="http://ns.adobe.com/pdf/2006" Open="true">
   <Bookmark>
      <Title>Company Overview</Title>
      <Action>
         <GoTo>
            <Dest>
               <XYZ Left="72" Top="205" Zoom="" Page_ref="/backbone.xml#0"/>
            </Dest>
         </GoTo>
      </Action>
   </Bookmark>
   <Bookmark>
      <Title>Target Market</Title>
      <Action>
         <GoTo>
            <Dest>
               <XYZ Left="72" Top="351" Zoom="" Page_ref="/backbone.xml#0"/>
            </Dest>
         </GoTo>
```

```
      </Action>
   </Bookmark>
   <!-- This next bookmark contains two child bookmarks.-->
   <Bookmark Open="true">
      <Title>EXR-55 Product Overview</Title>
      <Action>
         <GoTo>
            <Dest>
               <XYZ Left="72" Top="639" Zoom="" Page_ref="/backbone.xml#0"/>
            </Dest>
         </GoTo>
      </Action>

      <Bookmark>
         <Title>Major Features</Title>
         <Action>
            <GoTo>
               <Dest>
                  <XYZ Left="72" Top="165" Zoom="" Page_ref="/backbone.xml#1"/>
               </Dest>
            </GoTo>
         </Action>
      </Bookmark>
      <Bookmark>
         <Title>Detailed Specification</Title>
         <Action>
            <GoTo>
               <Dest>
                  <XYZ Left="72" Top="96" Zoom="" Page_ref="/backbone.xml#2"/>
               </Dest>
            </GoTo>
         </Action>
      </Bookmark>
   </Bookmark>
</Bookmarks>
```

Specifying destinations

A destination defines a particular view of a document to be rendered when a bookmark is activated, an annotation is opened, or an action is performed. The view consists of:

- The page to be displayed
- The location of the page within the document window
- The magnification (zoom) factor to use when displaying the page

Destinations are associated with bookmarks, annotations, or actions. A destination may be specified either explicitly by specifying the page and location on the page, or indirectly by specifying the name of a destination that resolves to an explicit destination.

Explicit destinations

The table below shows the allowed syntax for specifying a destination explicitly in a Mars document. In each case, `Page_ref` is an indirect reference to a page element in the document backbone. All coordinate values (`Left`, `Right`, `Top`, and `Bottom`) are expressed in user space. The page's bounding box is the smallest rectangle enclosing all of its contents. If any side of the bounding box lies outside the page's crop box, the corresponding side of the crop box is used instead.

The table below describes the explicit destinations that can appear as children of the `Dest` element.

Explicit destinations

Explicit destination element	Example	Description

XYZ	`<Dest>`
	`  <XYZ`
	`  Left="72" Top="587"`
	`  Zoom=""`
	`  Page_ref="/backbone.xml#2"`
	`  />`
	`</Dest>`
		Displays the page with the upper-left corner of the window located at (72, 587). The contents of the page are magnified with the current zoom factor.

Fit	`<Dest>`
	`  <Fit`
	`  Page_ref="/backbone.xml#2"`
	`  />`
	`</Dest>`
		Displays a page that fits within the horizontal and vertical dimensions of the window.

FitH	`<Dest>`
	`  <FitH Top="500"`
	`  Page_ref="/backbone.xml#2"`
	`  />`
	`</Dest>`
		Displays the page vertically with the upper edge located 500 pixels from the top of the window. The contents of the page are magnified to fit the entire width of the page within the window.

FitV	`<Dest>`
	`  <FitV Left="100"`
	`  Page_ref="/backbone.xml#2"`
	`  />`
	`</Dest>`
		Displays the page horizontally with the left edge located 100 pixels from the left of the window. The contents of the page are magnified to fit the entire height of the page within the window.

FitR	`<Dest>`
	`  <FitR Left="100"`
	`  Bottom="500"`
	`  Right="400"`
	`  Top="100"`
	`  Page_ref="/backbone.xml#2"`
	`  />`
	`</Dest>`
		Displays the page with its contents magnified to fit the rectangle within the horizontal and vertical dimensions of the window.

FitB	`<Dest>`
	`  <FitB`
	`  Page_ref="/backbone.xml#2"`
	`  />`
	`</Dest>`
		Displays the page contents magnified to fit the page's bounding box entirely within the window horizontally and vertically.

Explicit destination element	Example	Description
FitBH	<Dest> <FitBH Top="100" Page_ref="/backbone.xml#2" /> </Dest>	Displays the page with the upper edge located 100 pixels from the top of the window. The contents of the page are magnified to fit the entire width of the page's bounding box within the window.
FitBV	<Dest> <FitBV Left="100" Page_ref="/backbone.xml#2" /> </Dest>	Displays the page with the left edge located 100 pixels from the left of the window. The contents of the page are magnified to fit the entire height of the page's bounding box within the window.

Named destinations

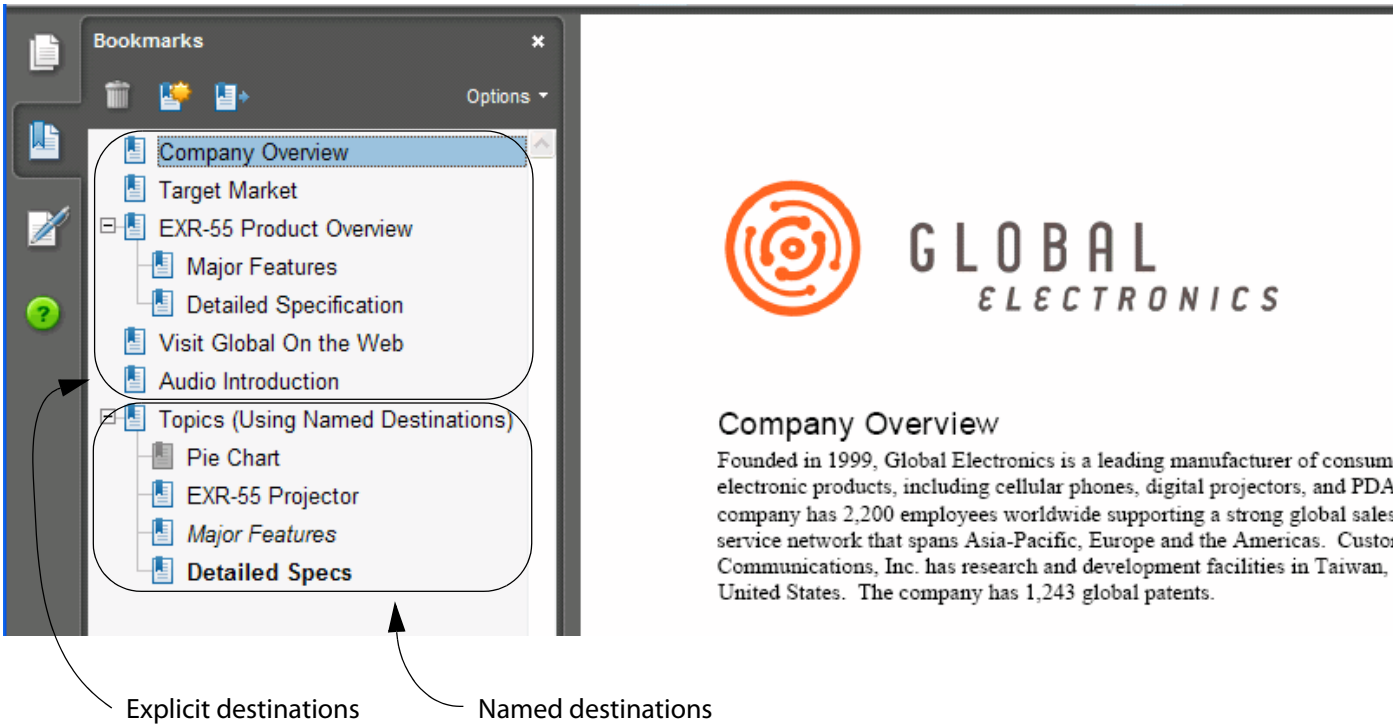
A named destination is a pair consisting of a string name and a PDF destination identifying a page and possible view parameters to be applied to a view of that page. This is known as a named destination and is useful when a Mars document (the source document) specifies a destination in another PDF or Mars document (the destination document). For example, a Mars file (the source file) can have a link that specifies the destination named `Chap6.begin` that is located in another Mars or PDF document. The source file provides the name of the destination and the file name for the destination; and the destination file associates the name with a view of a particular page.

A named destination offers the advantage that the location of the chapter in the other document could change without invalidating the link. For instance, if the destination document's Chapter 6 was moved from page 130 to page 145, the link in the source document would still work because it was not referring to the page that Chapter 6 started on, but the destination named `Chap6.begin`.

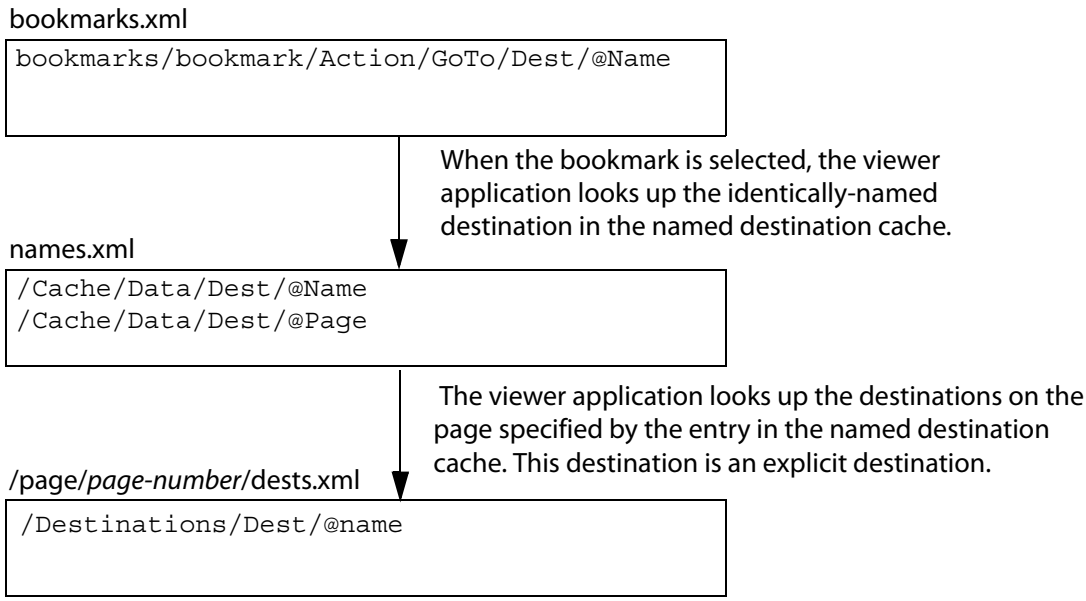
The following figure shows four bookmarks that use named destinations. These bookmarks have the display names "Pie Chart", "EXR-55 Projector", "Major Features", and "Detailed Specs". When one of these bookmarks is selected, the viewer application associates the name provided in the bookmark's destination element with an identically named destination in the dest.xml file in one of the pages that comprise the Mars document. In the case of the "Pie Chart" bookmark, the destination name is associated with a "fit page" view of the first page in the document. This association is expedited by the named destination cache, which enables the viewer application to find the page on which the explicit destination resides, without searching through the destinations listed for each page.

The illustration on page 68 shows the mapping from the bookmark named destination (in /bookmarks.xml), to the cache file named destinations data (in /cache/dests.xml), to the relevant page's explicit destinations (/page/*page-number*/dests.xml).

Named destination bookmarks



Named destination resolves to an explicit destination



Note: This diagram uses XPath expressions to identify attributes.

The corresponding XML and the explanation following the figure explains how these bookmarks are specified and which files are affected.

Designing the bookmark outline for the document

Using named destinations requires some planning, as described in the following steps:

1. Review your destination document (which can also be the source document), and decide where in the document the destinations should point. Typically, these will be chapters, sections, or subsections. You can also identify other points of interest such as figures, indices, examples, and so on.

2. For each location in the document, decide on the placement and alignment you want to use for the page. The choices for placement and alignment are as specified in the explicit destination styles described in the table on page page 66. For example, the `XYZ` destination style is the most common. This style causes the viewer application to position the by placing certain coordinates of the page at the upper left-hand corner of the viewing window.

3. Decide on a name for each named destination. Names must be unique within the document. Names can be logical such as using "Chapter1", "Chapter2", etc.; names can be based on the content such as using section titles, or names can be synthetic, or some combination thereof. Adding a document identifier to the name is useful to avoid collisions if you expect to be moving content between documents.

Creating a bookmark that uses a named destination

One of the most common places in which named destinations are used is within bookmarks. Named destinations are specified in the `Action/GoTo/Dest` properties of `Bookmark` elements.

To specify named destinations in bookmarks for a Mars document, use the following instructions in place of step 7 in "Creating the bookmarks file" on page 63.

For each bookmark that uses a named destination, perform these tasks:

1. Create an `Action` element as a child of the `Bookmark` element.

2. Create a `GoTo` element as a child of the `Action` element.

3. Create a `Dest` element as a child of the `GoTo` element.

4. Set the `Dest` element's `Name` attribute to the unique name you determined in "Designing the bookmark outline for the document" on page 69. This same name will be repeated in the named destination cache (/cache/dests.xml) and the page's named destinations (/page/*page-number*/dests.xml).

The following example shows one top-level bookmark titled "Topics (Using Named Destinations)" that contains four child bookmarks each of which specify named destinations as actions. The bookmarks that use named destinations correspond to the bookmarks titled "Pie Chart", "EXR-55 Projector", "Major Features", and "Detailed Specs."

Example 8.3 ***Specifying named destinations (bookmarks.xml)***

```
<Bookmarks xmlns="http://ns.adobe.com/pdf/2006" Open="true">
   …
   <Bookmark Open="true">
      <Title>Topics (Using Named Destinations)&#13;</Title>
      <Bookmark>
         <Action>
            <GoTo>
               <Dest Name="PieChart"/>
            </GoTo>
         </Action>
         <Title>Pie Chart&#13;</Title>
      </Bookmark>
      <Bookmark>
         <Action>
            <GoTo>
               <Dest Name="EXR55"/>
            </GoTo>
         </Action>
         <Title>EXR-55 Projector</Title>
      </Bookmark>
      <Bookmark Styles="Italic">
         <Action>
            <GoTo>
               <Dest Name="MajorFeatures"/>
            </GoTo>
         </Action>
         <Title>Major Features&#13;</Title>
      </Bookmark>
      <Bookmark Styles="Bold">
         <Action>
            <GoTo>
               <Dest Name="DetailedSpec"/>
            </GoTo>
         </Action>
         <Title>Detailed Specs&#13;</Title>
      </Bookmark>
      <Action>
         <GoTo>
            <Dest>
               <Fit Page_ref="/backbone.xml#2"/>
            </Dest>
         </GoTo>
      </Action>
   </Bookmark>
</Bookmarks>
```

Creating the page-level target destinations

The named destinations specified in the bookmarks file resolve to page-level destinations. These page-level destinations tie a destination name to an explicit destination. These page-level target destinations are specified in the dests.xml file located in the /page/*page-number* directory.

To create the page-level target destinations, perform the following steps for each page that contains defined named destinations:

1. Create the dests.xml file located in the directory /page/*page-number*. Ensure that the first line contains the standard XML declaration: `<?xml version="1.0" encoding="UTF-8"?>`
2. Create the `Destinations` element as the root element, specifying the default namespace as follows:

   ```
   xmlns="http://ns.adobe.com/pdf/2006"
   ```

3. For each named destination provided by the page, do the following:
   - Create a `Dest` element as a child of the `Destinations` element.
   - Set the `Dest` element's `Name` attribute to the name you determined in "Designing the bookmark outline for the document" on page 69.
   - Create an explicit destination element as a child of the `Dest` element. See the table on page 66 for the possible kinds of destinations. Each explicit destination element includes a `Page_ref` attribute that references the page. In most cases, the reference will identify the backbone object that in turn references the page.
4. List the named destinations in the named destination cache ("Creating named destination caches" on page 71).

Example 8.4 ***Specifying named destinations (/page/0/dests.xml)***

```
<Destinations xmlns="http://ns.adobe.com/pdf/2006">
   <Dest Name="EXR55">
      <Fit Page_ref="/backbone.xml#0"/>
   </Dest>
   <Dest Name="PieChart">
      <Fit Page_ref="/backbone.xml#0"/>
   </Dest>
</Destinations>
```

Creating named destination caches

A named destination cache is described by the names.xml file in the /cache directory. The file contains the following parts:

- Header that contains XSLT, XPath, or XQuery expressions. These expressions are functions that can be run against the other content in the document to obtain the cache information in the body. Such expressions allow the cache to be self-describing. The header part is typically the same for every names.xml file but can vary depending on which file organization and naming conventions are used.
- Body that contains an array of `Data` elements, each of which maps a name to the named destinations in on a particular page.

Cache files are either predefined or self-describing.

A predefined cache is one that relies on the Mars processing application to regenerate the data in the cache. Typically, the data is regenerated using a standard XSLT script that operates on the files specified by a pattern.

A self-describing cache is one whose cached data can be recomputed using only information in the cache file itself and the contents of the document package. Self-describing caches provide an extensible cache mechanism. Part of the cache file provides a pattern which selects a set of files in the package. A second part of the cache file specifies an XPath or limited XSLT expression to be applied to the selected files. The final part of the cache file (the data part) contains the XML which is a result of the evaluation of the expression, query, or script.

To create the named destination cache for a Mars document, perform the following tasks:

1. Create an XML file called names.xml located in the /cache directory, ensuring the first line in the file is the standard XML declaration: `<?xml version="1.0" encoding="UTF-8"?>`
2. Create the `Cache` element as the root element, specifying the default namespace as follows:

   ```
   xmlns="http://ns.adobe.com/pdf/2006"
   ```

3. Create a `Query` element as a child of the Cache element.
4. Create a `Files` element as a child of the `Query` element.
5. Create a `Pattern` element as a child of the `Files` element, and set its `Value` attribute to specify a file-selection pattern used to determine which page-level files to consider. A file-selection pattern is a prototype path that can have the wildcard character "*". The patterns are matched against the fully qualified names of package files. A package file is matched by a pattern if its fully qualified name, represented as a string, matches a pattern. For example:

   ```
   <Pattern Value="/page/*/struct.xml"/>
   ```

6. Create additional `Pattern` elements as needed to fully describe the files that contribute to this cache.
7. If the cache is predefined, set the `Cache` element's `Identifier` attribute to the value shown below:

   ```
   <Cache xmlns="http://ns.adobe.com/Mars"
      Identifier="http://ns.adobe.com/pdf/2006/cache/destinations">
   ```

8. If the cache is self-describing, add a `Query` element as a child of the `Cache` element, and set the element's attributes to provide XSLT and/or XPath expressions that establish the method this cache file uses to be self-describing. The expressions are applied to the files identified in the `Pattern` attribute created in step 5.
   - To specify an XPath expression, add the `XPath` attribute to the `Query` element. For example:

     ```
     <Query XPath="/cache" />
     ```

   - To specify a custom XSLT script, add the `Translate` attribute, setting its value to the location of the script. For example:

     ```
     <Query Translate="MyNamedDestinationTranslator.xslt" />
     ```

9. Create the `Data` element as a child of the `Cache` element.

10. For each named destination, create a `Dest` element as a child of the `Data` element. Set each `Dest` element's attributes as follows:
    - Set the `Name` attribute to specify the name you determined in "Designing the bookmark outline for the document" on page 69. This name must match its counterparts in the bookmarks file (/bookmarks.xml) and the page-level target destinations file (/page/*page-number*/dests.xml).
    - Set the `Page` attribute to reference the page-level target destinations file.

If the document is modified later and named destinations added or removed, the cache must be updated to be consistent with the rest of the document.

The following example shows how to specify a named destination cache for a document with four named destinations.

Example 8.5 ***Specifying a named destination cache (/cache/names.xml)***

```
<Cache Identifier="http://ns.adobe.com/pdf/2006/cache/destinations"
   xmlns="http://ns.adobe.com/pdf/2006">
  <Query>
    <Files>
      <Pattern Value="/page/*/dests.xml"/>
    </Files>
  </Query>
   <Data>
      <Dest Name="EXR55" Page_ref ="/page/0/dests.xml"/>
      <Dest Name="PieChart" Page_ref ="/page/0/dests.xml"/>
      <Dest Name="MajorFeatures" Page_ref ="/page/1/dests.xml"/>
      <Dest Name="DetailedSpec" Page_ref ="/page/2/dests.xml"/>
   </Data>
</Cache>
```

9 Representing Actions

This chapter describes how to represent predefined actions and JavaScript actions in a Mars document.

This chapter contains the following information.

Topic	Description	See
Understanding actions	Describes the sequencing mechanisms for actions, explains the difference between predefined and JavaScript actions and provides a list of predefined actions.	page 74
Specifying actions	Explains how to specify actions.	page 76

Understanding actions

Mars supports a set of predefined actions and form-submission actions.

Predefined actions

Predefined actions specify a variety of events that should occur in response to a trigger. The trigger depends on the ancestor elements that contain the action. Examples of predefined actions are changing the current view of the document, changing to a view in another document, launching an application, or printing a document.

Predefined actions can appear in the following settings:

- Document catalogs
- Page information files
- Outlines objects (bookmarks)
- Annotations

Actions do not play a role in XML forms. The *Adobe XML Forms Architecture (XFA) Specification, v2.4* defines a range of triggers and related actions that are independent of the actions described here. For more information, see *Adobe XML Forms Architecture (XFA) Specification, v2.4*.

In the case of outline objects (bookmarks), each object can contain a single action, whose trigger is implied to be the user selecting the bookmark. In all other cases, the trigger is defined as an element that in turn contains a child that expresses the action to be performed when the trigger is activated. Examples of such explicit triggers are opening a document, mousing over a widget (on cursor enter), or closing a page (on page close). The elements that describe the action triggers can specify multiple triggers, each with its own action.

Page content annotations include elements that specify the triggers for their activations. As a simple example, link annotations can have a single trigger, which is specified using the `OnActivation` element. As a more complex example, screen annotations can have an `OnActivation` element and multiple other trigger elements, including the `OnCalculate` element (which specifies a JavaScript action), the

`OnCursorEnter` element (which specifies a predefined action), and the `OnPageClose` element (which also specifies a predefined action).

For more information, see "Specifying content and markup annotations" on page 87.

The following table describes the predefined actions and corresponding examples.

Action	Description
`GoTo`	Go to a destination in the current file. For examples of this action, see "Go-to action" on page 76. For other examples of this action, see "Creating a bookmark that uses a named destination" on page 69 in the chapter Creating Bookmarks.
`GoToR`	Go to a destination in another document.
`GoToE`	Go to a destination in an embedded file.
`Launch`	Launch an application. Typically, this is used to open a file.
`Thread`	Begin reading an article thread. For examples of this action, see "Creating the articles file" on page 101 in the chapter Creating Articles.
`URI`	Resolve a uniform resource identifier. For examples of this action, see "URI action" on page 77.
`Sound`	Play a sound. For examples of this action, see "Sound action" on page 77.
`Movie`	Play a movie.
`Hide`	Set an annotation's Hidden flag.
`Named`	Execute an action predefined by the viewer application.
`SubmitForm`	Send data to a uniform resource locator. For examples of this action, see "Specifying a submit-form action" on page 164 in the chapter Creating Forms.
`ResetForm`	Set fields to their default values.
`ImportData`	Import field values from a file.
`JavaScript`	Execute a JavaScript script. For examples of this action, see "JavaScript action" on page 78.
`SetOCGState`	Set the state of optional content groups.
`Rendition`	Control the playing of multimedia content.
`Trans`	Update the display of a document, using transition parameters.
`GoTo3DView`	Set the current view of a 3D annotation.

Specifying actions

This section explains how to add some of the more common actions to annotations or bookmarks in a Mars document.

Go-to action

A go-to action changes the current document view to a specified destination, where the destination specifies a page in the document, the placement of the page in the viewing window, and the magnification of the page. Although this example uses `XYZ` placement, Mars also provides several other types of placement, as specified in the *Mars Reference*.

To specify a go-to action that uses XYZ placement, perform the following tasks:

1. Create a `GoTo` element as a child of the `Bookmark/Action` element or of a trigger element.
2. Create a `Dest` element that sets the destination for the page to be opened.
3. Create an `XYZ` element and set the following attributes. These attributes reflect SVG page coordinates.
   - Set the `Left` attribute to the left side of the element.
   - Set the `Top` attribute to the top of the element.
   - Set the `Zoom` attribute to the magnification level to be used on the element.
   - Set the `Page_ref` attribute to the page to be referenced.

In this example, a `Bookmark` element is created in which a `GoTo` action with a page reference to the `/backbone.xml#0` page (page 0). The destination page is positioned at the upper left coordinates (`72`, `205`). The zoom value is null, which means that the zoom factor will not be changed for this page. The title for the bookmark is `Company Overview`.

Example 9.1 ***Specifying a GoTo action (bookmarks.xml)***

```
<Bookmarks xmlns="http://ns.adobe.com/pdf/2006" Open="true">
   <Bookmark>
      <Action>
         <GoTo>
            <Dest>
               <XYZ Left="72" Top="205" Zoom="" Page_ref="/backbone.xml#0"/>
            </Dest>
         </GoTo>
      </Action>
      <Title>Company Overview</Title>
   </Bookmark>
```

URI action

A URI action invokes the default browser and directs it to display a page specified by the URI.

To specify a URI action, perform the following tasks:

1. Create a `URI` element, as a child of the trigger element. In the case of bookmarks, the URI action would be located in the `Action` element.
2. Set the `URI` element's `URI`attribute to the URI of the destination site.
3. If the action is triggered by the user clicking a content annotation, set the `URI` element's `IsMap` attribute to specify whether to track the mouse position when the URI is resolved. A value of `true` causes the coordinates of the mouse position at the time the action is performed to be transformed from device space to user space and then offset relative to the upper-left corner of the annotation rectangle (that is, the value of the `Rect` entry in the annotation with which the URI action is associated). Default value: false.

In this example, the URI is set to `http://www.adobe.com`.

Example 9.2 ***Specifying a URI action***

```
<Bookmark>
   <Action>
      <URI
         URI="http://www.adobe.com" />
   </Action>
   <Title>Visit Global On the Web</Title>
</Bookmark>
```

Sound action

A sound action plays a sound through the computer's speakers.

To specify a sound action, perform the following tasks:

1. Create a `Sound` element, as a child of the `Bookmark/Action` element or of a trigger element.
2. Set the `Sound` element's attributes as follows:
   - `Volume` attribute to the volume at which to play the sound, in the range -1. to 1.0. The default value is 1.0.
   - `Synchronous` attribute (optional) to `true` to specify that the sound should be played synchronously, or `false` otherwise. If this flag is true, the viewer application retains control, allowing no further user interaction other than canceling the sound, until the sound has been completely played. The default value is `false`.
   - `Repeat` attribute (optional) to true if this sound should be repeated endlessly. If this attribute is present, the `Synchronous` attribute is ignored. The default value is `false`.
   - `Mix` attribute (optional) to `true` to specify that this sound should be mixed with any other sound already playing, or `false` otherwise. The default value is `false`.

3. Create a `SoundFile` element as a child of the `Sound` element, and set the following attributes:
   - `src` attribute that references the file containing the sound.
   - `SamplingRate` attribute (optional) that specifies the sampling rate for the sound file in seconds.
   - `ChannelCount` attribute (optional) that specifies the number of sound channels. The default value is 1.
   - `BitsPerSample` attribute (optional) that specifies the number of bits per sample value per channel. The default value is 8.
   - `Encoding` attribute (optional) that specifies the encoding format for the sample data. Supported values are `Raw`, `Signed`, `muLaw`, and `ALaw`. The default value is `Raw`.
   - `Compression` attribute (optional) that specifies the sound compression format used.

This example creates a bookmark with a sound action. The sound file is located at in the sound_4209265555-6 file located in the /media directory. The sampling rate is set to `22050` samples per second.

Example 9.3 ***Specifying a sound action (bookmarks.xml)***

```
<Bookmark>
   <Action >
      <Sound>
         <SoundFile src="/media/sound_4209265555-6"
            SamplingRate="22050">
         </SoundFile>
   </Action>
   <Title>Audio Introduction</Title>
</Bookmark>
```

JavaScript action

Every JavaScript action provides a JavaScript expression, as described in Specifying the JavaScript action. If certain JavaScript expressions are repeatedly used by JavaScript actions, they can be defined as named JavaScript expressions, which makes them globally available. That is, those expressions can be invoked by the scripts in the JavaScript actions. Named JavaScript segments are described in Specifying named JavaScript expressions.

Specifying the JavaScript action

To specify a JavaScript action, perform the following tasks:

1. Create a `Script` element, as a child of the element that describes the action trigger. For example, JavaScript actions can appear in widget annotations or in the document catalog.

2. If the JavaScript segment is to be provided in the element, specify it in the `Script` element. If this segment uses JavaScript methods not supported by Acrobat, define these as described in "Specifying named JavaScript segments" on page 80. For information on JavaScript supported by Acrobat, see *Developing Acrobat Applications Using JavaScript*.

3. If the JavaScript is to be provided in a separate file, create a `File` element as a child of the `Script` element. Specify the file name as the contents of the `File` element.

The *Client-Side JavaScript Reference*, available from the Sun Microsystems website, and the *Adobe JavaScript for Acrobat API Reference* give details on the contents and effects of JavaScript scripts.

The following example shows an annotation-level action that specifies a JavaScript segment. The action is triggered on recalculation of the value of the field to which this widget is associated. Such recalculation is triggered when there is a change in the value of another field on which this field depends. This routine increments by 4 the value specified in the `date_beg` field. It then isolates the first and second parts of the date, and sets the value of the event to the first part, separated by a slash ("/"), and followed by the second part.

Example 9.4 ***A widget annotation that specifies JavaScript code (/page/page_number/pg.can)***

```
<Annotations>
   <Widget>
      <OnCalculate>
         <Script>var d=this.getField("date_beg");
            var a=add_days(d,4);
            if (a=="") {
            }
            else {
               var theNumbers = a.split("/");
               event.value = theNumbers[0] + "/" + theNumbers[1];
            }
         </Script>
      </OnCalculate>
   </Widget>
</Annotations>
```

The following example specifies a document-level action. The action is triggered just before the document closes, at which point the JavaScript interpreter executes the function specified in the `Script` element, passing in the parameters. Because MyScript is non-standard, it must be included in the Mars document, as described in "Specifying named JavaScript expressions" on page 79. Because the methods used in this script are standard, they don't need to be defined at the document level.

Example 9.5 ***A document action that references a named JavaScript segment (/backbone.xml)***

```
<PDF>
   <BeforeClose>
      <Script>MyScript(param_1, param_2);</Script>
   </BeforeClose>
</PDF>
```

Specifying named JavaScript expressions

To specify a named JavaScript expression that can be referenced from a `Script` element, perform these tasks:

- Specifying the location of the JavaScript file
- Specifying named JavaScript segments

Specifying the location of the JavaScript file

To specify the location of the JavaScript file, modify the backbone file (backbone.xml), as follows:

1. Create the `JavaScripts` element as a child of the `PDF` element.
2. Set the `JavaScripts` element's `src` attribute to specify the path of the file containing the named JavaScript segments. By convention, the path is /script/javascripts.xml.

The following example specifies the location of the file containing named JavaScript segments.

Example 9.6 ***Specifying the location of the JavaScript files (backbone.xml)***

```
<PDF>
   <JavaScripts src="/script/javascripts.js">
   …
</PDF>
```

Specifying named JavaScript segments

Named JavaScript segments enable you to more efficiently represent custom JavaScript methods used in multiple JavaScript actions. For information on JavaScript supported by Acrobat, see *Developing Acrobat Applications Using JavaScript*.

To specify named JavaScript segments, perform these tasks:

1. In the /script directory, create an XML file that uses the suffix .js. Ensure the first line in the file is the standard XML declaration <?xml version="1.0" encoding="UTF-8"?>.
2. Create a `JavaScripts` element, as the root directory of the file.
3. Create a `JavaScript` element as a child of the `JavaScripts` element, setting its `Key` attribute to a name that uniquely identifies the JavaScript segment.
4. Create a `Script` element as a child of the `JavaScript` element.
5. If the JavaScript segment is to be provided in the element, specify it in the `Script` element.
6. If the JavaScript is to be provided in a separate file, create a `File` element as a child of the `Script` element. Specify the file name as the contents of the `File` element.
7. Repeat steps 3. - 6. for each additional named JavaScript segment.

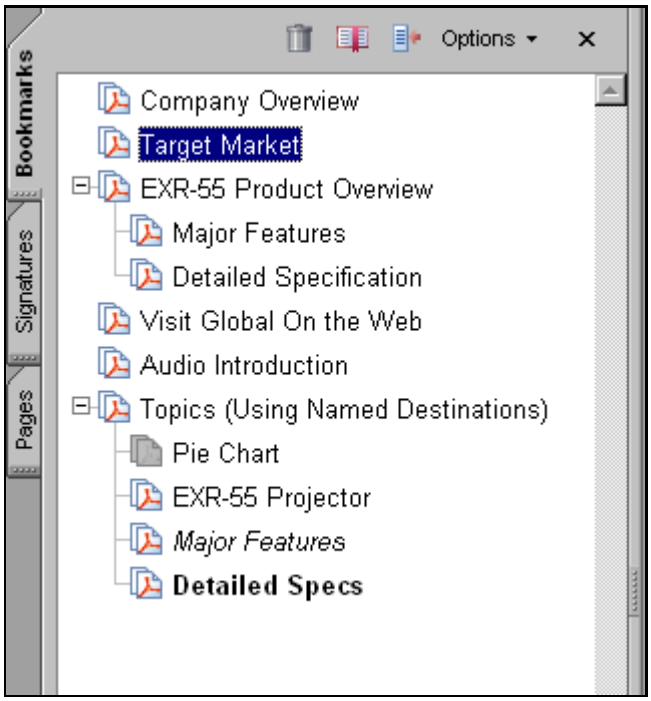
In this example, the JavaScript key is set to is set to `MyScript`. This script is referenced in the example "A document action that references a named JavaScript segment (/backbone.xml)" on page 79 .

Example 9.7 ***Defining the named JavaScript segments (/script/javascripts.js)***

```
<JavaScripts>
   <JavaScript Key="MyScript">
      <Script>
         The JavaScript code
      </Script>
   </JavaScript>
   ...
</JavaScripts>
```

The following figure displays the bookmarks specified in the preceding examples.

Action bookmarks



10 Adding Annotations

This chapter describes how to add content and markup annotations to an existing Mars document. These annotations provide additional graphic and interactive content in the document.

This chapter contains the following information.

Topic	Description	See
Understanding annotations	Explains what annotations are.	page 82
Specifying annotations	Explains how to specify content and markup annotations.	page 87

Understanding annotations

An annotation associates an object such as a note, sound, or movie with a location on a page of a Mars document, and provides a way to interact with the user by means of the mouse and keyboard. Mars includes a wide variety of standard annotation types.

For Mars documents, annotations are represented in XML. The annotations are split into two groups called content and markup annotations.

Annotations are placed in separate files that are grouped by page. For each page, there can be two annotation files, one containing markup annotations such as sticky-notes and text comments, and another containing content-oriented annotations such as form field widgets and hyperlinks.

Content annotations

Content annotations are explicitly referenced by the page information file, and represent material that is considered part of page content. This group includes form fields and link annotations. Content annotations are typically stored in a file called pg.can within each page directory. Page directories are conventionally named /page/*page-number*, where *page-number* is a zero-based page number. The content annotation files are referenced by the page information file.

Mars defines the following content annotations.

Link	Movie	Screen	Widget
PrinterMark	TrapNet	Watermark	A3D

Unlike the other content annotations, widget annotations are used in conjunction with Acrobat form fields. The form field describes the behavior and in some cases the data of a button, choice or text field presented to the user and the widget describes the appearance of that button, choice, or text field. Widget annotations are described in "Creating Acrobat forms" on page 160.

The figure below is a snapshot of a Mars document with a link annotation (titled "Page 3 has great specs") and a widget content annotation (titled "Go to specs on page 3"). "Specifying content and markup annotations" on page 87 provides instructions on creating these annotations.

Mars document page with link and widget content annotations

Markup annotations

Markup annotations represent human reader-created marks on the page that supplement page content, perhaps added as part of a review and approval cycle. These are implicitly associated with the page and document and can be added without changing any other files in the document.

Mars defines the following markup annotations.

Text	FreeText	Line	Square
Circle	Polygon	Highlight	Underline
Squiggly	StrikeOut	Caret	Stamp
Ink	FileAttachment	Sound	

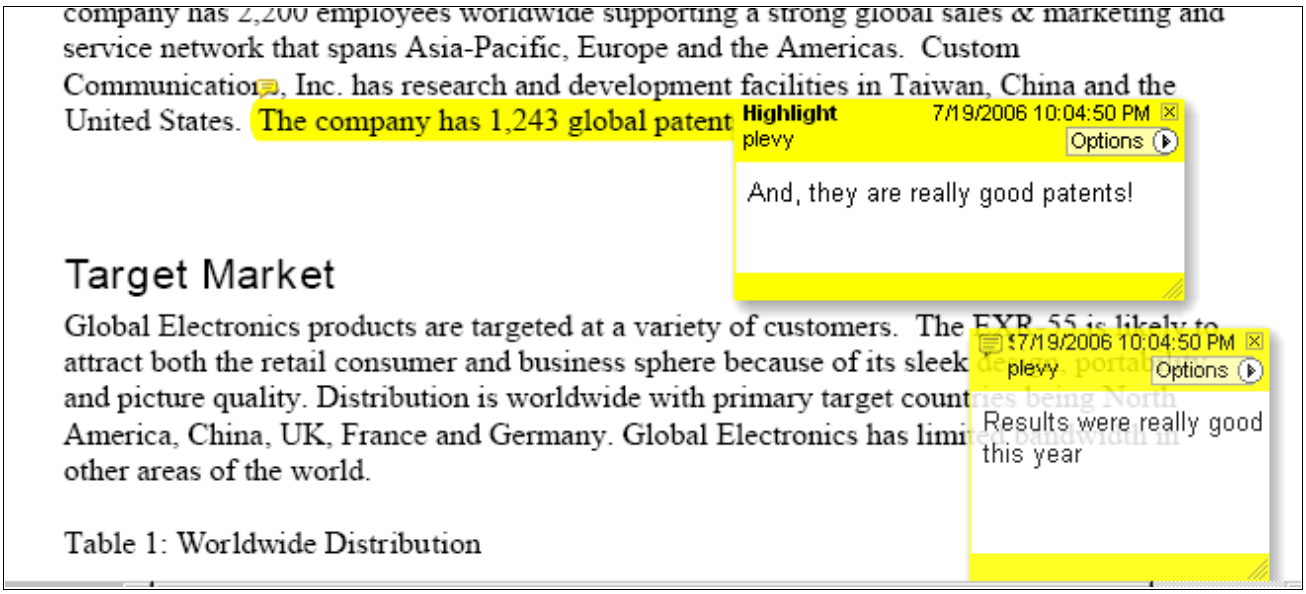
Note: Although the *PDF Reference* describes the Popup annotation as a first rate content annotation, it plays a subordinate role to markup annotations. To reflect this role, Mars represents the Popup annotation as a property of the markup annotations listed above.

Markup annotations are stored in a file called pg.svg.ann within each page directory, assuming the page content is stored in a file named pg.svg within the same directory. These files are referenced by the page information file. Unlike content annotations, the markup annotations file is not directly named by the backbone file or the page information file (info.xml). Instead, the implicit association rule specified in the package file (package.xml) specifies that a file named *Name*.ann provides markup annotations for the file named *Name*, where both files are in the same directory. For example, markup annotations for the pg.svg file would appear in the pg.svg.annfile, where both files are in the directory /page/0.

For information on implicit associations, see the *Mars Reference*.

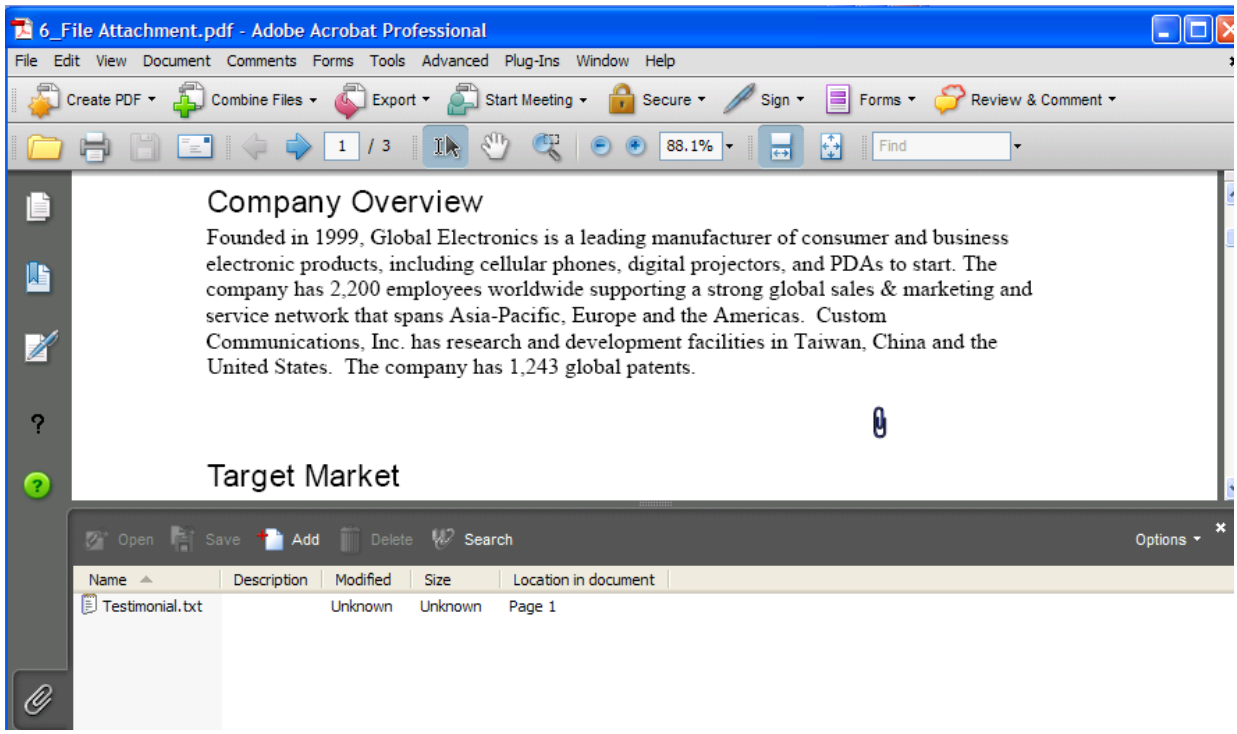
The following figure shows the Mars document with the text and highlight markup annotations. "Creating markup annotations" on page 89 provides instructions on creating these annotations.

Mars document page with text and highlight markup annotations



The following figure shows a Mars document with a file attachment markup annotation. "Creating markup annotations" on page 89 provides instructions on creating file attachment markup annotations.

Mars document page with a file attachment markup annotation



For more information on markup annotations, see Table 8.16 in the *PDF Reference, version 1.7*.

Display properties of annotations: basic properties, appearances and default appearances

The display properties of annotations can be specified at various levels, each one offering increased flexibility. At its simplest level, the display properties can be specified as annotation attributes such as color and border. At the most detailed level, they can be specified using *Appearances*. Appearances are sets of SVG graphic descriptions that are associated with annotation states. And in the case of annotations that present user-provided text, the display properties of text are specified using *default appearances*.

Basic display properties for annotations

Annotations can specify simple display properties such as border width, border style, and border fill color.

Appearances assigned to annotation states

In addition to setting simple display properties, annotations can specify one or more Appearances as an alternative to the simple border and color characteristics available by default. Multiple appearances enable the annotation to be presented visually in different ways to reflect its interactions with the user. Each Appearance specifies the color, fonts, and graphics to be rendered inside the annotation rectangle.

The graphic representation of annotations is usually explicit, and presented in the document as part of the annotation information. This representation, called an Appearance, specifies the display properties and SVG content for specific annotation states. See "Specifying different appearances for different annotation states" on page 85. The SVG content is specified as a reference to an SVG file located in the Mars document. The Appearance also specifies how the SVG content should be transformed and cropped when it is placed on the page.

These SVG fragments can either appear inline in the annotation element or in separate package-level files referenced from the `Appearance` element.

Specifying different appearances for different annotation states

Annotations can specify multiple appearances, where each appearance is used when the annotation is in a different state. An annotation can define appearances for the following states:

- Normal, which is used when the annotation is not interacting with the user. This appearance is also used for printing the annotation.
- Rollover, which is used when the user moves the cursor into the annotation's active area without pressing the mouse button.
- Down, which is used when the mouse button is pressed or held down within the annotation's active area.

Widget annotations that omit appearances

Widget annotations use appearances to provide the display properties of form fields. If any of the widget annotations in a Mars document omit appearances, the Mars viewer application provides an application-specific appearance. To warn the application that it must provide appearances, the document backbone file annotation `PDF/Acroform/@NeedAppearances` attribute is set to true.

Default appearances

Annotations that accept user-provided text can specify a `DefaultAppearance` attribute that defines the default characteristics used to render text such as the font, weight, and color. Widget annotations accept user-provided text on behalf of the form fields they represent.

Specifically default appearances can appear in the following locations:

- Document backbone file as the attribute specified by the XPath expression:

```
PDF/AcroForm/@DefaultAppearance
PDF/AcroForm/Fields/Button/@DefaultAppearance
PDF/AcroForm/Fields/TextField/@DefaultAppearance
PDF/AcroForm/Fields/Choice/@DefaultAppearance
PDF/AcroForm/Fields/Container/@DefaultAppearance
```

Note: The `PDF/AcroForm/Fields/Container` shows only the highest level appearance of the `Container` element. That element can itself include a hierarchy of form fields.

- Content annotations file (/page/*page_number*/pg.can) as the attribute specified by the XPath expression:

```
Annotations/Screen/AppearanceCharacteristics/@DefaultAppearance
Annotations/Widget/AppearanceCharacteristics/@DefaultAppearance
```

- Markup annotations file (/page/*page_number*/pg.svg.ann) as the attribute specified by the XPath expression:

```
Annotations/FreeText/@DefaultAppearance
Annotations/Redact/@DefaultAppearance
```

By themselves, default appearances are different from Appearances, but when default appearances are combined with text entered by the user, they create a new normal-state Appearance for the annotation. This new appearance is virtual. That is, it is never included in Mars documents.

The following example sets the default appearance for text within a text field to the Helvetica font with a gray fill and leaves the font size unchanged.

Example 10.1 ***Specifying a default appearance***

```
<TextField UIName="Total registration fees:  # of registrations"
   Name="Total registration fees:  # of registrations"
   Widget_ref="/page/0/pg.can#11" Widget_ref_type="dictionary">
   <DefaultAppearance Font="Helv" TextSize="0"
      FillColorSpace="Gray" FillColor="0"/>
   <Resources>
      <Fonts>
         <Font Tag="Helv" src="/font/f_240-1.fd"/>
      </Fonts>
      <Encoding type="dictionary">
         <PDFDocEncoding type="dictionary" ref="#0" ref_type="dictionary"/>
      </Encoding>
   </Resources>
</TextField>
```

For more information on content and markup annotations, see section 8.4 in the *PDF Reference, version 1.7*.

Specifying content and markup annotations

This section explains how to create content and markup annotations. This section is a companion to the *Mars Reference* sections "Page content annotations" and "Markup annotations."

Creating content annotations

This section explains how to create link annotations. Such annotations are used for interactive features and are usually associated with Acrobat form fields.

Creating a simple content annotation involves the following general steps:

- Planning the content annotations and the appearances they use
- Specifying the root elements in the content annotation file
- Referencing the content annotations file from the page information file
- Adding annotations to the content annotations file
- Specifying appearances
- Specifying default appearances

Specifying the root elements in the content annotation file

To specify an annotation, perform the following tasks:

1. Create the content annotations file (pg.can) in the page directory. Ensure the first line in the file is the standard XML declaration `<?xml version="1.0" encoding="UTF-8"?>`.
2. Create an `Annotations` element as the root element in the content annotations file, specifying the default namespace as follows:

   ```
   xmlns="http://ns.adobe.com/pdf/2006"
   ```

3. Specify the annotation element. This section provides instructions on specifying some types of annotations, as described in "Adding annotations to the content annotations file" on page 88,

Referencing the content annotations file from the page information file

To reference the content annotations file, make the following modifications to the page information file (info.xml) in the page directory for which the content annotations are being added:

1. Create an `Annotations` element as a child of the `Page` element.
2. In the `Annotations` element, add a `src` attribute that specifies the path of the file containing the content annotations.

The following example provides the location of the content annotation file.

Example 10.2 ***Page information file (/page/page_number/info.xml) provides location of content annotations***

```
<Page xmlns="http://ns.adobe.com/pdf/2006">
   <Annotations src="/page/0/pg.can"/>
   <Contents src="/page/0/pg.svg"/>
</Page>
```

Adding annotations to the content annotations file

To add a link annotation, make the following additions to the content annotation file created in "Specifying the root elements in the content annotation file" on page 87:

1. Create a `Link` element as a child of the `Annotations` element.
2. Set the `Link` element's `Rect` attribute to specify the location where the annotation should be placed on the page. The value of the `Rect` attribute is four numbers that specify SVG coordinates.
3. If you want the visual characteristics of the annotation to change when the user presses or holds down the mouse button in the field's active area, set the `Highlight` attribute to specify the annotation's highlighting mode. A highlighting mode other than `Push` overrides any down appearance defined for the annotation.

   The following highlighting modes are supported.

`None`	No highlighting.
`Invert`	Invert the contents of the annotation rectangle.
`Outline`	Invert the annotation's border.
`Push`	Display the annotation as if it were being pushed below the surface of the page.

4. If you want a border around the annotation, create a `Border` element as a child of the `Link` element, setting the `Border` element's optional `Style` attribute to specify the border effect to apply. The following border styles are defined:

`Solid`	A solid rectangle surrounding the annotation.
`Dash`	A dashed rectangle surrounding the annotation. The dash pattern is specified by a dash array in the `Border` element.
`Beveled`	A simulated embossed rectangle that appears to be raised above the surface of the page.
`Inset`	A simulated engraved rectangle that appears recessed below the surface of the page.
`Underline`	A single line along the bottom of the annotation rectangle

5. Set other annotation as needed.
6. If an action is required, create an `OnActivation` element or a `Dest` element to describe the action to perform when the user selects the annotation. The `OnActivation` element specifies a range of actions that should be performed and the `Dest` element specifies a particular page view.

The example below causes a viewer application to present the annotations shown on page 83. These declarations create a link content annotation within a rectangle that has a solid border. The `URI` for the link is set to `http://www.adobe.com`, which means that when the link is clicked, the view changes to the URI http://www.adobe.com. The `HCornerRadius` and `VCornerRadius` are both set to 0, which means that the border has square corners with a border width of 1.

Example 10.3 ***Specifying a link content annotation (/page/page-number/pg.can)***

```
<Annotations xmlns="http://ns.adobe.com/pdf/2006">
   <Link Rect="165 176 388 101" Highlight="Invert">
      <Border Style="Solid" Width="1"/>
      <OnActivation>
         <URI URI="http://www.adobe.com"/>
      </OnActivation>
   </Link>
   …
</Annotations>
```

Specifying appearances

To add Appearances to the annotation, add the following XML to the annotation:

1. Create an `Appearance` element as a child of the annotation element, which the previous example specifies as the `Link` element.
2. Create a `Normal` element as a child of the `Appearance` element.
3. Create a `Graphic` element as a child of the `Normal` element, and specify its attributes as follows:
   - `src` attribute that references SVG graphic content used in the appearance
   - `Matrix` attribute that describes transformations from the current user space to a new user space.
   - `BBox` attribute that specifies a location on the page to use as the destination for the appearance.
   - Other attributes and child elements as needed
4. If the annotation includes a down or rollover appearance, create the corresponding elements as children of the `Appearance` element.

Specifying default appearances

If you are creating a screen or widget annotation, you can provide default formatting applied to user-supplied text. This formatting includes characteristics such as font face, font weight, and color. For more information about default appearances, see "Display properties of annotations: basic properties, appearances and default appearances" on page 85.

To specify default formatting for an annotation, add the following XML to the annotation:

1. Create an `AppearanceCharacteristics` element as a child of the annotation element.
2. Add a `DefaultAppearance` element as a child of the `AppearanceCharacteristics` element. Specify properties for displaying the field's variable text, such as such as text size and color. See the DA entry in Table 8.71 in the *PDF Reference, version 1.7*.

Creating markup annotations

This section explains how to create text, highlight, line and file attachment markup annotations. Such annotations are used primarily to markup a document. They have text that appears as part of the annotation and may be displayed in other ways by a viewer application, such as in a Comments pane.

Text annotations

To create a text annotation for a Mars document, perform the following steps:

1. Determine where on the page the text annotation should be placed.
2. Create the markup annotations file (pg.svg.ann) in the page directory. Ensure the first line in the file is the standard XML declaration `<?xml version="1.0" encoding="UTF-8"?>`.
3. Create an `Annotations` element as the root element in the markup annotations file, specifying the default namespace as follows:

   ```
   xmlns="http://ns.adobe.com/pdf/2006"
   ```

4. Create the `Text` element, and set its attributes as follows:
   - `CreationDate` attribute that specifies the date and time the annotation was created. Section 3.8.3 of the *PDF Reference, version 1.7* describes the format of this attribute. For example: `CreationDate="D:20060706154901-07'00'"`
   - `Rect` attribute that specifies the annotation rectangle, defining the location of the annotation on the page in SVG page coordinates.
   - `Name` attribute that specifies a name that uniquely identifies the annotation among all the annotations on its page.
   - `Color` attribute that specifies numbers that define a color in a particular color space. The number of color elements determines the color space. For example, three numbers specify a color in the DeviceRGB color space. This color is used for the following purposes:
     - The background of the annotation's icon when closed
     - The title bar of the annotation's pop-up window
5. Set the `Text` element's `Flags` attribute to specify the behavior of the annotation. This value is a space-separated list of options:
   - The `Print` option specifies that the annotation should be included when the page is printed.
   - The `NoZoom` option specifies that the annotation's appearance should be independent of the magnification of the page.
   - The `NoRotate` option specifies that the annotation's appearance should be independent of the page's rotation. That is, the upper-left corner of the annotation rectangle remains in a fixed location on the page, regardless of the page rotation.
   - The other options are `Invisible`, `Hidden`, `Print`, `NoView`, `ReadOnly`, `Locked`, `ToggleNoView`, and `LockedContents`.
6. Set the `Text` element's `ModDate` attribute to specify the date and time when the annotation was most recently modified. Section 3.8.3 of the *PDF Reference, version 1.7* describes the format of this attribute.
7. Set the `Text` element's `IconName` element to the name of an icon to be used in displaying the annotation. Viewer applications should provide predefined icon appearances for the following icon names:

   ```
   Comment          Key              Note

   Help             NewParagraph     Paragraph

   Insert
   ```

8. Set the `Text` element's `Title` attribute to specify the text label to be displayed in the title bar of the annotation's pop-up window when open and active. By convention, this entry identifies the user that added the annotation.

9. Set the `Text` element's `Subj` attribute to specify the text representing a short description of the subject being addressed by the annotation.

10. Create a `Contents` element as a child of the `Text` element, setting its value to the text to be displayed for the annotation.

11. Create a `RTContents` element as a child of the `Text` element, setting its value to a rich text string to be displayed in the pop-up window when the annotation is opened. When specifying the rich text string in the `RTContents` element, use the following conventions:

    - To avoid confusing the rich text XML with the Mars XML, use the escaped expressions for delimiters, as follows:

`<`	In place of the less-than character, "<"
`>`	In place of the greater-than character ">"
`&quot;`	In place of the quote

    - Create the `body` rich text element as a child of the `RTContents` element.
    - Set the `body` element's default namespace to `http://www.w3.org/1999/xhtml`.
    - Set the `body` element's namespace for the `xfa` prefix as follows, replacing each quotation mark with `quot;`. This namespace establishes the level of rich text expressions contained in the `body` element.

      ```
      xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
      ```

    - Qualify the `xfa` prefix with the expression, replacing each quotation mark with `quot;`.

      ```
      xfa:APIVersion="Acrobat:8.0.0"
      ```

    - For more information, see the section "Rich Text Strings" in the *PDF Reference, version 1.7*.

12. Create the `Popup` element as a child of the `Text` element. This element specifies a pop-up annotation for entering or editing the text associated with this annotation. Set the `Rect` and `Flags` attributes to indicate the size, location and viewing options set for the popup. Set the `Open` attribute to specify whether the pop-up annotation should initially be displayed open.

13. Optionally, add Appearances as described in "Specifying appearances" on page 89

14. Optionally, add a reference to the data file in which the Mars viewer saves data for the markup annotation.

The example below causes a viewer application to present one of the annotations shown on page 84. These declarations create a text markup annotation with a creation date of July 6, 2007 at 3:49:01 PM PDT. The color for the annotation is yellow (#FFFF00 in the DeviceRGB color space). The flags specify that the annotation can be printed, but will not be zoomed or rotated. One of the predefined icon names `Comment` is to be used as the icon to be displayed. The title for the text is set to `John Smith`. The subject displayed in the annotation title bar is `Sticky Note` and the annotation's contents are `Results were really good this year`. The font is set to a 10.0 pt size. The popup will not be open when the document is opened. This example was taken from pg.svg.ann from the page directory.

Example 10.4 ***Specifying a text markup annotation (/page/page-number/pg.svg.ann)***

```
<Annotations xmlns="http://ns.adobe.com/pdf/2006">
   <Text Rect="288.422 510.014 308.422 492.01"
      CreationDate="D:20060706154901-07'00'"
      Name="b1554a32-4e82-48c4-ae31-8228cb5b5ce5"
      Color="#FFFF00"
      Flags="Print NoZoom NoRotate"
      ModDate="D:20060706154946-07'00'"
      IconName="Comment"
      Title="John Smith"
      Subj="Sticky Note">
      <Contents>Results were really good this year</Contents>
      <RTContents>&lt;?xml version=&quot;1.0&quot;?&gt;&lt;
         body xmlns=&quot;http://www.w3.org/1999/xhtml&quot;
         xmlns:xfa=&quot;http://www.xfa.org/schema/xfa-data/1.0/&quot;
         xfa:APIVersion=&quot;Acrobat:8.0.0&quot;
         xfa:spec=&quot;2.0.2&quot; &gt;&lt;p dir=&quot;ltr&quot;&gt;
         &lt;span dir=&quot;ltr&quot; style=&quot;font-size:10.0pt&quot;
         &gt;Results were really good this year&lt;/span
         &gt;&lt;/p&gt;&lt;/body&gt;</RTContents>
      <Popup Rect="415.667 601.34 529.666 505.34"
         Flags="Print NoZoom NoRotate" Open="false"/>
      <Appearance>
         <Down>
            <Graphic src="/page/0/form_3208737192-2.svg"
               Matrix="1 0 0 1 0 0" BBox="0 0 20 18"/>
         </Down>
         <Normal>
            <Graphic src="/page/0/form_3208737192-3.svg"
               Matrix="1 0 0 1 0 0" BBox="0 0 20 18"/>
         </Normal>
      </Appearance>
   </Text>
</Annotations>
```

Highlight annotations

To create highlight annotations for a Mars document, perform the following steps:

1. Follow all steps for creating a text annotation, using the `Highlight` element in place of the `Text` element and omitting step 7. That step describes setting the `IconName` attribute for the annotation, which doesn't apply to highlight annotations.

2. Set the `Highlight` element's `Coords` attribute to an array of 8 x n numbers specifying the coordinates of n quadrilaterals in SVG page coordinates that comprise the region in which the link should be activated.

The example below causes a viewer application to present one of the annotations shown on page 84. These declarations create a highlight that is applied to the section of text indicated by the coordinates. The annotation was created on July 6, 2006 at 3:50:30 PM PDT and was modified on July 6, 2006 at 3:51:00 PM PDT. For more information, see section 3.8.3 in the *PDF Reference, version 1.7*. The color for the annotation is yellow (#FFFF00 in DeviceRGB). The flags set for the annotation indicate the annotation can be printed. The title for the text is set to `John Smith` because that is the user's name. The subject of the annotation is `Sticky Note` and the text is set to `And, they are really good patents!`. The font is set to a 10.0

pt size. The popup will be closed when the document is opened. This example was taken from pg.svg.ann from the page directory.

Example 10.5 ***Specifying a highlight markup annotation (/page/page-number/pg.svg.ann)***

```
<Annotations xmlns="http://ns.adobe.com/pdf/2006">
   <Highlight Rect="141 311.077 334.288 294.32"
      CreationDate="D:20060706155030-07'00'"
      Name="f6b23c30-7dc5-449d-a8da-56262b507eff" Color="#FFFF00"
      Flags="Print" ModDate="D:20060706155100-07'00'" Title="John Smith"
      Subj="Highlight"
      Coords="145.363,294.816 330.079,294.816 145.363,310.584
330.079,310.584">
      <Contents>And, they are really good patents!</Contents>
      <RTContents>&lt;?xml version=&quot;1.0&quot;?&gt;
         &lt;body xmlns=&quot;http://www.w3.org/1999/xhtml&quot;
         xmlns:xfa=&quot;http://www.xfa.org/schema/xfa-data/1.0/&quot;
         xfa:APIVersion=&quot;Acrobat:8.0.0&quot;
         xfa:spec=&quot;2.0.2&quot; &gt;&lt;p dir=&quot;ltr&quot;&gt;
         &lt;span dir=&quot;ltr&quot; style=&quot;font-size:10.0pt&quot;&gt;
         And, they are really good patents!&lt;/span&gt;&lt;/p&gt;
         &lt;/body&gt;</RTContents>
      <Popup Rect="326 370.98 497 294.48"
         Flags="Print NoZoom NoRotate" Open="false"/>
      <Appearance>
         <Normal>
            <Graphic src="/page/0/form_3208737192-13.svg"
               Matrix="1 0 0 1 0 0" BBox="0 0 193.134 16.753"/>
         </Normal>
      </Appearance>
   </Highlight>
</Annotations>
```

Line annotations

To create line annotations for a Mars document, perform the following steps:

1. Follow all steps for creating a text annotation (page 90), using `Line` element in place of the `Text` element and omitting step 7. That step describes setting the `IconName` attribute for the annotation, which doesn't apply to highlight annotations.

2. Set the `Line` element's attributes as described below:

   - `InteriorColor` attribute (optional) to numbers that specify the color with which to fill the annotation's line endings. The number of color elements determines the color. For example, three numbers specifies a color in the DeviceRGB color space.
   - `Start` and `End` attributes (required) that specify the starting and ending coordinates of the line. These coordinates are in SVG coordinates.
   - `HeadStyle` and `TailStyle` attributes (optional) that specify the line ending styles to be used in drawing the line. Defined values are `Square`, `Circle`, `Diamond`, `OpenArrow`, `ClosedArrow`, and `None`. The default value is `None`.
   - `Intent` attribute (optional) that specifies the intent of the line annotation. Defined values are `LineArrow`, which means that the annotation is intended to function as an arrow, and `LineDimension`, which means that the annotation is intended to function as a dimension line.

3. Optionally, create a `CS` element as a child of the `Line` element. This element identifies the process color space, which may be any device or CIE-based color space. If an ICC-based color space is specified, it must provide calibration information appropriate for the process color components specified in the names array of the DeviceN color space.

4. Optionally, create a `Border` element as a child of the `Line` element, and set its `Width` attribute to indicate the border width in points.

The following example creates a line annotation with a creation date of `7/06/06` at `3:51:55 PM`. This time is stored in the standard date format as `D:20060706155155-07'00'`. For more information, see section 3.8.3 in the *PDF Reference, version 1.7*. The border width for the line is set to `4` which indicates the thickness of the line and the color is set to `#FF0000` which is red. The `Flags` attribute is set to `Print` which means to print the annotation when the page is printed. If this flag was not set, the annotation would never be printed, regardless of whether it was displayed on the screen. The `Intent` element is set to `LineArrow` which means that the annotation is intended to function as an arrow. The subject that appears on the menu bar is `Line`. The line heading has the `OpenArrow` style, which indicates two short lines meeting in an acute angle to form an open arrowhead. The line tail has no line ending style. The text is set to `These specs are great, aren't they?` via the `Contents` attribute. The `Open` flag was set to `false`, which indicates that the popup will not be open when the document is opened.

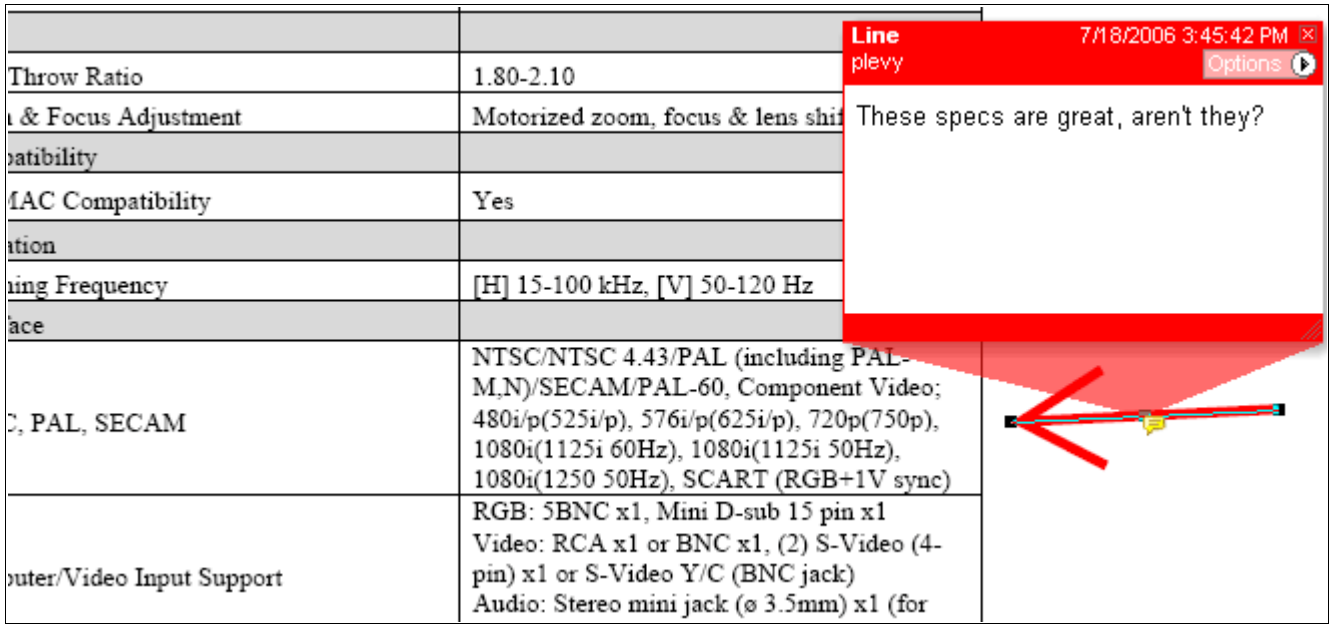
Example 10.6 ***Specifying a line markup annotation (path name: /page/page-number/pg.svg.ann)***

```
<Annotations xmlns="http://ns.adobe.com/pdf/2006">
   <Line Rect="464.772 501.777 575.361 451.818"
      CreationDate="D:20060706155155-07'00'"
      Name="7340a218-6aa5-4c81-9ab0-327692c9dc7f"
      InteriorColor="#0000FF"
      Color="#FF0000"
      Flags="Print"
      Start="467.7,478.6" End="568.3,473.8"
      ModDate="D:20060706155410-07'00'"
      Intent="LineArrow"
      Title="John Smith"
      Subj="Line" HeadStyle="OpenArrow" TailStyle="None">
      <Border Width="4"/>
      <Contents>These specs are great, aren't they?</Contents>
      <CS Value="None" type="name"/>
      <RTContents>&lt;?xml version=&quot;1.0&quot;?&gt;&lt;body
         xmlns=&quot;http://www.w3.org/1999/xhtml&quot;
         xmlns:xfa=&quot;http://www.xfa.org/schema/xfa-data/1.0/&quot;
         xfa:APIVersion=&quot;Acrobat:8.0.0&quot;
         xfa:spec=&quot;2.0.2&quot; &gt;
         &lt;p dir=&quot;ltr&quot;&gt;
         &lt;span dir=&quot;ltr&quot;
         style=&quot;font-size:10.0pt&quot;&gt;
         These specs are great, aren't they?&lt;/span&gt;
         &lt;/p&gt;&lt;/body&gt;</RTContents>
      <Popup Rect="612 595.048 792 475.048"
         Flags="Print NoZoom NoRotate" Open="false"/>
      <Appearance>
         <Normal>
            <Graphic src="/page/2/form_3208737192-16.svg"
               Matrix="1 0 0 1 -464.772 290.223"
               BBox="464.772 290.223 575.361 340.182"/>
         </Normal>
```

```
      </Appearance>
   </Line>
</Annotations>
```

The following figure shows a Mars document with text and line markup annotations.

Mars document page with text and line markup annotations



File attachment annotations

File attachment annotations are markup annotations stored with the other annotations associated with a page. The attached file itself is stored as a separate file within the document package. The embedded files represent supplementary files that are part of the document package, but are not displayed.

Listed below are some factors to be aware of when using embedded files.

- The embedded file is represented as a separate file in the document package. The file must appear in the `/file` directory of the package or a subdirectory. The embedded file can have any name that is a legal name as defined by Zip. The file name within the package is used only to reference the file from the backbone or annotation. A separate `Name` or `UName` attribute within the XML holds the file name that will be displayed for the user.
- In the Mars document backbone file, the `EmbeddedFiles` element contains an `EmbeddedFile` element for every for every embedded file in the document. Each `EmbeddedFile` element contains the file name and optional information, such as a user interface description of the file, MIME type, creation date, checksum, and last modification date. See “Adding embedded files” on page 96.
- Optionally, a file attachment markup annotation contains the same information as the `EmbeddedFile` element, as well as the information needed to display the annotation on its page. The annotation information is stored in the pg.svg.ann file associated with the page on which it appears. See “Adding an embedded file annotation” on page 97..

For more information, see section 3.10.3, and the `EmbeddedFiles` entry in Table 3.28 in the *PDF Reference, version 1.7*.

Adding embedded files

To add embedded files to a Mars document, perform the following steps:

1. In the backbone file, create an `EmbeddedFiles` element as a child of the `PDF` element. Create the `EmbeddedFile` element's `src` attribute to reference the "embedded_files.xml" file located in the /file directory.

   ```
   <EmbeddedFiles src="/file/embedded_files.xml"/>
   ```

2. If it doesn't already exist, create a file named embedded_files.xml located in the /file directory. Ensure the first line contains the standard XML declaration

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

3. If it doesn't already exist, create an `EmbeddedFiles` element, as the root of the file.

4. Create the `EmbeddedFile` element as a child of the `EmbeddedFiles` element, and set its attributes as follows:

   - `Key` attribute (required) that specifies a unique name for the embedded file. No two embedded files can have the same `Key` value.
   - `Name` and `UName` attributes (both optional) that specify the embedded file name. `UName` is the Unicode version of the file name and `Name` is the file name that does not have any encoding specified.

5. Create the `FileData` element as a child of the `EmbeddedFile` element, and set its attributes as follows:

   - `src` attribute (required) that specifies the location of the embedded file within the Mars package
   - `FileType` attribute (optional) that specifies the MIME type for the file. For example: `text/plain` or `image/bmp`

6. Optionally, create the `Params` element as a child of the `FileData` element, and set its attributes as follows:

   - `CreationDate` attribute (optional) that specifies the date and time the document was created. Section 3.8.3 of the *PDF Reference, version 1.7* describes the format of this attribute. For example: `CreationDate="D:20060706154901-07'00'"`
   - `ModDate` attribute (optional) that speciies the date and time the embedded file was last modified
   - `Size` attribute (optional) that specifies the size of the embedded file in bytes
   - `CheckSum` attribute (optional) that specifies a 16-byte string that is the checksum of the bytes of the uncompressed embedded file. The checksum is calculated by applying the standard MD5 message-digest algorithm (described in Internet RFC 1321, The MD5 Message-Digest Algorithm) to the bytes of the embedded file stream.

7. Optionally, create a `Desc` element as a child of the `EmbeddedFile` element, setting its value to the descriptive text associated with the file specification.

In this example, the file was last modified on `7/06/06` at `4:10:59 PM`. The embedded file is an image and is named `arch_pic.bmp`.

Example 10.7 ***Specifying an embedded file (/file/embedded_files.xml)***

```
<EmbeddedFiles xmlns="http://ns.adobe.com/pdf/2006">
   <EmbeddedFile Key="AUniqueIdentifier" UName="arch_pic.bmp"
      Name="arch_pic.bmp">
      <FileData src="/file/ef_3208737192-1" FileType="image/bmp">
         <Params CreationDate="D:20060706161059-07'00'"
            ModDate="D:20060706161059-07'00'"
            Size="506934"
            Checksum="48uBet+wuxfygp5JKshUcQ=="/>
      </FileData>
      <Desc>This is a sample image that can be projected with the EXR-55</Desc>
   </EmbeddedFile>
</EmbeddedFiles>
```

Adding an embedded file annotation

A file attachment annotation contains a reference to a file that is embedded in the Mars document. For example, a table of data might use a file attachment annotation to link to a spreadsheet file based on that data; activating the annotation extracts the embedded file and gives the user an opportunity to view it or store it in the file system.

To add an embedded file annotation to a Mars document, perform the following steps:

1. Perform steps 1-6 for creating a text annotation (page 90), using the `FileAttachment` element in place of the `Text` element.

2. Set the `FileAttachment` element's attributes as follows:

   - `IconName` attribute (optional) to the name of an icon to be used in displaying the annotation. Viewer applications should provide predefined icon appearances for at least the following standard names:

     `Graph` `PushPin`

     `Paperclip` `Tag`

   - `Title` attribute (optional) to specify the text label to be displayed in the title bar of the annotation's pop-up window when open and active. Often, this entry is the file name.
   - `Subj` attribute (optional) to specify the text representing a short description of the subject being addressed by the annotation.
   - `Opacity` attribute (optional) to the opacity value to be used in painting the annotation. This value applies to all visible elements of the annotation in its closed state (including its background and border) but not to the popup window that appears when the annotation is opened.

3. Create a `File` element as a child of the `FileAttachment` element, and set its `Name` attribute to the name of the attached file.

4. Create a `FileData` element as a child of the `File` element, and set its attributes as follows:
   - `src` attribute that references the embedded file.
   - `FileType` attribute that specifies the MIME type for the file. For example: `text/plain` or `image/bmp`.
   - Optionally, add Appearances as described in "Specifying appearances" on page 89

The example below causes a viewer application to present the annotations shown on page 84. These declarations create a file attachment annotation named Testimonial.txt located in the /file folder. This is represented by the blue paper clip icon in the Mars document shown at right. One of the predefined icon names, `Paperclip` is specified as the icon to be displayed. The `subject` attribute for the file attachment is set to `File Attachment`.

Example 10.8 ***Specifying a file attachment annotation (/page/page-number/pg.svg.ann)***

```
<Annotations xmlns="http://ns.adobe.com/pdf/2006">
   <FileAttachment Rect="427.758 341.156 434.759 324.156"
      CreationDate="D:20030920191259-07'00'"
      Name="X61Bc0WXz7TUDoeQGOcwlD"
      Opacity="0.94" Color="#3E53FF"
      Flags="Print NoZoom NoRotate"
      ModDate="D:20030920191303-07'00'"
      IconName="Paperclip" Title="Testimonial"
      Subj="File Attachment">
      <File Name="Testimonial.txt">
         <FileData src="/file/Testimonial.txt" FileType="text/plain"/>
      </File>
      <Appearance>
         <Down>
            <Graphic src="/page/0/form_1664690369-12.svg"
               Matrix="1 0 0 1 0 0" BBox="0 0 7 17"/>
         </Down>
         <Normal>
            <Graphic src="/page/0/form_1664690369-14.svg"
               Matrix="1 0 0 1 0 0" BBox="0 0 7 17"/>
         </Normal>
      </Appearance>
   </FileAttachment>
</Annotations>
```

The following examples contain the SVG referenced by the annotation appearances. Normally, the paperclip is rendered with a blue fill and a black stroke, but when the paperclip is selected, these colors are reversed. Example 10.9 shows the Down appearance, which draws a paperclip with a black fill and a blue stroke. Example 10.10 shows the Normal appearance, which draws a paperclip with a blue fill and a black stroke.

Example 10.9 ***SVG that draws the graphic referenced from the Down appearance***

```
<?xml version="1.0" encoding="utf-8"?>
<svg xml:space="preserve" xmlns:pdf="http://ns.adobe.com/pdf/2006"
xmlns:xlink="http://www.w3.org/1999/xlink" fill="none" stroke="none"
width="7" height="17">
   <defs> … </defs>
   <g transform="matrix(1 0 0 -1 0 17)">
      <g clip-path="url(#cl-1)">
```

```
        <g clip-path="url(#cl-2)">
           <g clip-path="url(#cl-3)">
              <path fill="rgb(0,0,0) device-color(DeviceGray,0)"
                 d="M0.51,13.63c0-0.38-0.03-9.25-0.03-9.89c0-0.45,…"/>
              <path stroke="rgb(62.999,83.9988,255)" stroke-width="0.59"
                 stroke-linecap="butt" stroke-linejoin="round"
                 d="M0.51,13.63c0-0.38-0.03-9.25-0.03-9.89c0-0.45,…"/>
           </g>
        </g>
     </g>
  </g>
</svg>
```

Example 10.10*SVG that draws the graphic referenced from the Normal appearance*

```
<svg xml:space="preserve" xmlns:pdf="http://ns.adobe.com/pdf/2006"
xmlns:xlink="http://www.w3.org/1999/xlink" fill="none" stroke="none"
width="7" height="17">
   <defs> … </defs>
   <g transform="matrix(1 0 0 -1 0 17)">
      <g clip-path="url(#cl-1)">
         <g clip-path="url(#cl-2)">
            <g clip-path="url(#cl-3)">
               <path fill="rgb(62.999,83.9988,255)"
                  d="M0.51,13.63c0-0.38-0.03-9.25-0.03-9.89c0-0.45,…"/>
               <path stroke="rgb(0,0,0) device-color(DeviceGray,0)"
                  stroke-width="0.59" stroke-linecap="butt"
                  stroke-linejoin="round" d="M0.51,…"/>
            </g>
         </g>
      </g>
   </g>
</svg>
```

For more information, see *File Attachment Annotations* in 8.4.5 in the *PDF Reference, version 1.7*.

11 Creating Articles

This chapter describes a form of page-level navigation called *articles* that enables the user to navigate through logically-related information located on different pages within a document.

This chapter contains the following information.

Topic	Description	See
Understanding articles	Explains what articles are and why they are used.	page 100
Creating articles	Explains how to specify articles, threads, and beads.	page 100

Understanding articles

In Mars documents, articles can be used to create an association between information that is logically related but is not physically sequential. For example, a news story may begin on the first page of a newsletter and span several nonconsecutive interior pages. To represent such sequences of noncontiguous logically-related items, a Mars document may define one or more articles. The sequential flow of an article is defined by an article thread; the individual content items that make up the article are called beads on the thread. Mars viewer applications can provide navigation facilities to allow the user to follow a thread from one bead to the next. For more information on articles and beads, see section 8.3.2 in the *PDF Reference, version 1.7*.

Creating articles

Creating articles involves the following general tasks:

- Planning
- Referencing the threads file from the backbone file
- Creating the articles file

Planning

Determine all of the pages in the document that contain parts of articles. Each of these pages needs to be referenced by a `Bead` element in `articles.xml`. To enable this, each of the pages is required to have a unique `ID` attribute added to the `Page` element in the document backbone.

Referencing the threads file from the backbone file

To reference the threads file from the backbone file, modify the backbone file as follows:

1. Create a `Threads` element as a child of the `PDF` element.

2. Set the `Threads` element's `src` attribute to specify source file containing the articles. Conventionally, the source file is named articles.xml and is located in the root directory.

   The following example shows a `Threads` element in `backbone.xml`. The `Threads` element references `articles.xml` as the source of the articles in the document.

```
<PDF PDFVersion="1.7" Version="0.8.0" Class="04-18-07"
   xmlns="http://ns.adobe.com/pdf/2006">
   <Threads src="articles.xml" />
</PDF>
```

Creating the articles file

To create the articles file, perform the following tasks:

1. Create the articles.xml file located in the root directory, unless this file already exists. Ensure the first line has the standard XML declaration: `<?xml version="1.0" encoding="UTF-8"?>`

2. Create a `Threads` element as the root element. The `Threads` element contains all the article threads for a document.

3. Create an `ArticleThread` element as a child of the `Threads` element. This element will specify the first (or next) article thread. The order in which the `ArticleThread` elements appear is unimportant.

   Each article thread represents a logical flow of content within the document. If multiple logical content flows are needed, create a `ArticleThread` element to represent each flow. Each thread consists of a series of beads representing the location of the individual content items comprising the article.

   Each bead is defined in a `Bead` element, which is a rectangular region that points to the location of the content on the page. There can be multiple `Bead` elements within a `ArticleThread` element, where each bead represents the content location in the logical flow for the document.

4. For each bead in the article thread, create a `Bead` element as a child of the `ArticleThread` element. The order in which the `Bead` elements appears dictates the navigational order of the beads.

5. Set each `Bead` element's attributes as follows:

   - `x1`, `y1`, `x2`, `y2` attributes that are the coordinates for the rectangular region where the content is located. The (`x1`, `y1`) coordinates represent the upper-left corner of the rectangular region for the `Bead` element and the (`x2`, `y2`) coordinates represent the lower-right corner.
   - `Page_ref` attribute that references the page on which the bead is located. It should reference a `Page` element within the backbone using the URI for the `Page` element.

6. Optionally, create the `ThreadInfo` element to set some of the common optional attributes about the document such as `Subject`, `Author`, `Keywords`, and `Title`. For more information, see the `I` entry in Table 8.7 in the *PDF Reference, version 1.7*.

   - The `Subject` attribute specifies the subject of the document.
   - The `Author` attribute specifies the name of the person who created this document.
   - The `Keywords` attribute specifies keywords associated with this document.
   - The `Title` attribute specifies the document's title.

The following example was taken from `articles.xml` and shows a single article whose information is located in six locations (beads) throughout the document. The order in which the beads appear determines the navigation order. When the user clicks in the area defied by the first bead, the document view is changed to the destination specified by the next bead. When the user clicks in the area defined by the last bead, the document view changes to the destination specified in the first bead.

The subject of the document is set to `Unusual path through this document` and the author for this document and is set to `William Shakespeare`. The `Keywords` are set to `Fast` and the title for the document is set to `Sample Use of Articles`.

Example 11.1 ***Specifying threads and beads***

```
<Threads xmlns="http://ns.adobe.com/pdf/2006">
   <ArticleThread>
      <Bead x1="62" y1="468" x2="531" y2="595"
         Page_ref="/backbone.xml#0"/>
      <Bead x1="57.6" y1="340" x2="528" y2="450"
         Page_ref="/backbone.xml#0"/>
      <Bead x1="47.5" y1="540" x2="527" y2="639"
         Page_ref="/backbone.xml#1"/>
      <Bead x1="41.7" y1="446.4" x2="528" y2="540"
         Page_ref="/backbone.xml#1"/>
      <Bead x1="256" y1="224" x2="459" y2="289"
         Page_ref="/backbone.xml#2"/>
      <Bead x1="57.6" y1="305" x2="476" y2="364"
         Page_ref="/backbone.xml#1"/>
      <ThreadInfo Subject="Unusual path through this document"
         Author="William Shakespear" Keywords="Fast"
         Title="Sample Use of Articles"/>
   </ArticleThread>
</Threads>
```

12 Specifying Optional Content

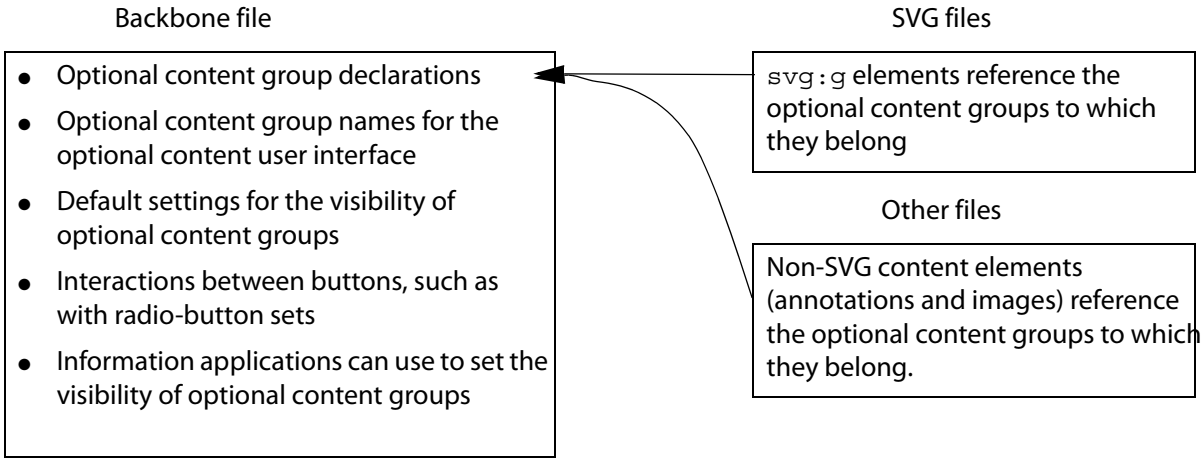
This chapter explains how to specify optional content. It contains the following information.

Topic	Description	See
"Understanding optional content" on page 103	Explains what optional content is and how it can be used.	page 103
"Specifying optional content" on page 103	Explains how to specify optional content in a Mars document.	page 103

Understanding optional content

Optional content refers to text and graphics in a Mars document that can be selectively viewed or hidden by document authors or consumers. This capability is useful in items such as engineering drawings, layered artwork, maps, and multi-language documents.

As shown in the following illustration, the backbone file establish the framework for the optional content groups, including identifiers, optional content group names and other settings. Graphic content in SVG files or elsewhere reference the optional content group to which they belong.



Specifying optional content

This section explains how to apply the optional content feature to an existing Mars document.

Planning

When planning optional content, consider the issues discussed here.

- Identify the optional content groups for your document.

  An optional content group represents a collection of graphics that can be made visible or invisible dynamically by users of viewer applications.

- Decide on the default state of the optional content. That is, decide which optional content groups should initially be visible and which should be invisible.

- Determine whether the optional content has characteristics the viewer application is aware of that would allow it to determine which optional content groups should be visible, such as the default language for the application or whether the document is being zoomed or printed. In such cases, the viewer application can determine whether the optional content groups should be visible or invisible.

  When optional content is grouped according to language or some other factor that the viewer application is aware of, it can be best to allow the viewer application to select the default appearance of the optional content. For example, if the optional content is grouped by language, it is best to allow the viewer application to select the optional content to display, based on its country-locale setting.

- Identify names of the optional content groups that are suitable for presentation in a viewer application user interface.

- Determine which graphic content and text to associate with the optional content groups. The graphic content and text belonging to an optional content group can reside anywhere in the document: they need not be consecutive in drawing order.

- Decide whether the optional content groups should be mutually exclusive.

  Mars enables you to specify that only one set of optional content be visible at a time, which is the behavior of radio buttons. This behavior might be used for a brochure that uses optional content to present multiple languages. For example, you might create multiple overlapping text frames on a page, with each text frame using a different language to convey the same information. The text can be organized into optional content groups that correspond to the language. The default configuration for the groups can be established such that the optional content groups are mutually exclusive. That is, if the user selects Spanish, the text associated with other languages disappears.

Establishing the optional content groups

This section explains how to enable optional content in a Mars document and how to establish the optional content groups, their default states, and other characteristics. All of these settings are made in the backbone file (backbone.xml).

➤ **To display the optional content panel when the document is first opened (backbone.xml):**

- To the `PDF` element, set the `PageMode` attribute to `UseOC`.

➤ **To define the optional content groups (backbone.xml):**

1. Add an `OCProperties` element as a child of the `PDF` element.
2. Add a `Groups` element as a child of the `OCProperties` element.
3. For each content group, add a `Group` element as a child of the `Groups` element.
4. Set the `Group` element's `ID` attribute to a string that uniquely identifies the optional content group. Graphic content and text use this ID value to associate themselves with optional content.
5. Set the `Group` element's `Name` attribute to the name of the optional content group. The name should be suitable for presentation in a viewer application's user interface.

6. Optionally, add a `Usage` element as a child of each `Group` element. This element specifies the nature of the content controlled by the group. It may be used by features that automatically control the state of the group based on outside factors.

7. If the viewer application should select optional content based on language, add a `Language` element to the `Usage` element. Set the Language element's `Lang` attribute to a value that specifies the 2-character language code and two-character country code that specifies the language and dialect. For example, a value of `en-US` specifies English as the language spoken in the United States and a value of `es-MX` specifies Spanish as the language spoken in Mexico.

➤ **To add a default optional content settings to the backbone file:**

1. Add a `Default` element as a child of the `OCProperties` element. This element will specify the default visibility of the different optional content groups. When an optional content document is viewed, this state is overridden by user interactions.

2. If any of the optional content groups should behave like radio buttons (where only one optional content group in the radio-button set is visible at a time), add an `ExclusiveGroups` element as a child of the `Default` element. Then, for each radio-button set, add an `ExclusiveGroup` element as a child of the `ExclusiveGroups` element. To the `ExclusiveGroup` element, add one `Group` element for each conditional content group in the radio-button set, setting the `Group` element's `ref` attribute to reference the name of the conditional content group.

3. Add a `BaseState` attribute to the `Default` element that specifies the default state of the conditional content groups. If the default state is that the groups are visible, set `BaseState` to `ON`. Otherwise, set it to `OFF`.

   This attribute has a default value of `ON`.

4. Add an `ON` element as a child of the `Default` element. To this element, add one `Group` element for each conditional content group that is initially visible.

   This element can be omitted if the `BaseState` attribute has a value of `ON` or is omitted.

5. Add an `OFF` element as a child of the `Default` element. To this element, add one `Group` element for each conditional content group that is initially invisible. Set the `Group` element's `ref` attribute to the name of the invisible optional content group.

   This element can be omitted if the `BaseState` attribute has a value of `OFF`.

6. Add a `PresentationOrder` element as a child of the `Default` element. To this element, add one `Group` element for each conditional content group, listing the conditional content groups in the order in which they should appear in the user interface.

   Note: Not addressed here is how to use the `PresentationOrder` element to represent in the user interface a hierarchical presentation of optional content groups.

7. Add a `ListMode` attribute to the `Default` element, settings its value to indicate whether optional content in the Presentation`Order` array should be displayed to the user. A value of `AllPages` directs the viewer application to display all optional content groups in the `PresentationOrder` array. A value of `VisiblePages` directs the viewer application to display only those groups in the Order array that are referenced by one or more visible pages.

   This attribute has a default value of `AllPages`.

8. Add a `UsageApplications` element as a child of the `Default` element. This element indicates whether the usage settings should be applied during viewing, printing, or exporting the document.

9. Add a `UsageApplication` element as a child of the `UsageApplications` element, setting the `Event` attribute to indicate the situation in which this usage application dictionary should be used, typically `View`.

10. Add a `Groups` element as a child of the `UsageApplication` element. To the `UsageApplication` element, add one `Group` element for each conditional content group in the usage application set, setting the `Group` element's `ref` attribute to reference the name of the conditional content group.

➤ **To add alternate optional content configuration settings:**

1. Add a `Configs` element as a child of the `OCProperties` element. This element will specify the other configurations that may be used under particular circumstances.

2. For each unique alternate configuration setting, add a `Config` element as a child of the `Configs` element. Add properties to the `Config` element, as described in steps 2-10 , beginning on page 105.

The following example is taken from the sample file named itto_autostate.mars. The example shows these optional content settings:

- Optional-content user interface panel. The first line of the file directs the viewer application to display the optional content user interface in the instruction `panel PageMode="UseOC"`.
- Optional content groups are established for the Italian, Spanish, German, French, and English languages. (`OCProperties/Groups`)
- Optional content groups have a radio-button behavior (where only one optional content group in the radio-button set is visible at a time). That is, if the Spanish optional content is selected, the other language-based optional content is invisible. (`OCProperties/Default/ExclusiveGroups`)
- Viewer applications can select optional content groups based on language. The `PDF/OCProperties/Groups/Group/Usage/Language` elements direct viewer applications to select content based on language and locale. The application determines which optional content groups to display, by comparing its own setting for language and locale against the Mars optional content group `Lang` value. The application then sets the visibility recommendation for each the optional group as follows:
  - Exact match. Optional content group is set to an ON recommendation.
  - Partial match (that is, the language matches but not the locale) for groups that have `Preferred` entries with a value of `ON`. Optional content group is set to an ON recommendation.
  - No match. Optional content group is set to an OFF recommendation.

Example 12.1 ***Backbone defines document-level optional content settings***

```
<!-- Optional-content user interface panel enabled. -->
<PDF PDFVersion="1.5" PageMode="UseOC" … >
   <OCProperties>
      <Default>
         <ExclusiveGroups>
         <!-- Optional groups are assigned a radio-button behavior. -->
            <ExclusiveGroup>
               <Group ref="#3"/>
               <Group ref="#2"/>
               <Group ref="#4"/>
               <Group ref="#0"/>
               <Group ref="#1"/>
            </ExclusiveGroup>
         </ExclusiveGroups>
```

```
      <ON/>
      <OFF>
         <Group ref="#3"/>
         <Group ref="#4"/>
         <Group ref="#0"/>
         <Group ref="#1"/>
      </OFF>
      <PresentationOrder>
         <Group ref="#4"/>
         <Group ref="#1"/>
         <Group ref="#3"/>
         <Group ref="#2"/>
         <Group ref="#0"/>
      </PresentationOrder>
      <UsageApplications>
   <!-- Viewer application can select optional groups based on language -->
         <UsageApplication Event="View">
            <Groups>
               <Group ref="#0"/>
               <Group ref="#1"/>
               <Group ref="#2"/>
               <Group ref="#3"/>
               <Group ref="#4"/>
            </Groups>
            <Usages>
               <Usage>Language</Usage>
            </Usages>
         </UsageApplication>
      </UsageApplications>
   </Default>
   <Configs/>
   <Groups>
   <!-- Optional groups are established for
      Italian, Spanish, German, French, and English. -->
      <Group Name="Diagram"/>
      <Group Name="Italian" ID="0">
         <Usage>
            <Language Lang="it-IT" Preferred="ON"/>
         </Usage>
      </Group>
      <Group Name="Spanish" ID="1">
         <Usage>
            <Language Lang="es-SP" Preferred="ON"/>
         </Usage>
      </Group>
      <Group Name="German" ID="2">
         <Usage>
            <Language Lang="de-DE" Preferred="ON"/>
         </Usage>
      </Group>
      <Group Name="French" ID="3">
         <Usage>
            <Language Lang="fr-FR" Preferred="ON"/>
         </Usage>
      </Group>
```

```
        <Group Name="English" ID="4">
          <Usage>
            <Language Lang="en-US" Preferred="ON"/>
          </Usage>
        </Group>
      </Groups>
    </OCProperties>
</PDF>
```

Marking SVG graphic content and text for inclusion in optional content groups

This section explains how to associate graphic content and text with optional content groups.

➤ **To create SVG that includes marked-content tags:**

1. In the `svg:g` element that represents the content associated with an optional content group, specify a `pdf:Mark` attribute with the value `OC`.
2. Add a `pdf:Props` element as the first child element in the `svg:g` element.
3. Add a `Membership` element as a child of the `pdf:Props` element.
4. Add a `Groups` element as a child of the `Membership` element.
5. For each optional content group to which the SVG content belongs, add a `Group` element to the `Groups` element. Set the `Group` element's `ref` attribute to reference the optional content group identification in the backbone file (for example, `ref="/backbone.xml#3"`)

Here is an example of SVG content that is marked as belonging to French. (The backbone file assigns the French optional content group an ID of 3.)

Example 12.2 ***SVG content associated with an optional content group***

```
<g pdf:Mark="OC">
   <Props xml:id="prop_1" xmlns="http://ns.adobe.com/pdf/2006">
      <Membership>
         <Groups>
            <Group ref="/backbone.xml#3"/>
         </Groups>
      </Membership>
   </Props>
   <text transform="matrix(1 0 0 -1 0 202.9538)"
      font-size="23.9449"
      font-family="F1" fill="rgb(255,255,255) icc-color(cs-0,1,1,1)"
      fill-rule="evenodd">
      <tspan x="94.954">Milles de mesure</tspan>
   </text>
   …
</g>
```

Marking non-SVG graphic content and text for inclusion in optional content groups

Any content that can be defined outside SVG can also be associated with optional content groups. Content and markup annotation, bookmarks, image, alternate images and form dictionaries (form XObjects) can all be associated with optional content groups. This is done by adding a `ContentGroup` element to the element that describes the content.

➤ **To associate a markup annotation with an optional content group:**

1. Open the file containing the markup annotation (/page/*page_number*/pg.svg.ann).
2. Add a `ContentGroup` element as a child of the element specifying the markup annotation. Examples of such elements are `Text`, `FreeText`, and `Line`.
3. For each optional content group the annotation belongs to, add a `Group` element as a child of the `ContentGroup` element. Define the `Group` element's ref attribute to reference the object in the backbone file that declares the optional content group.

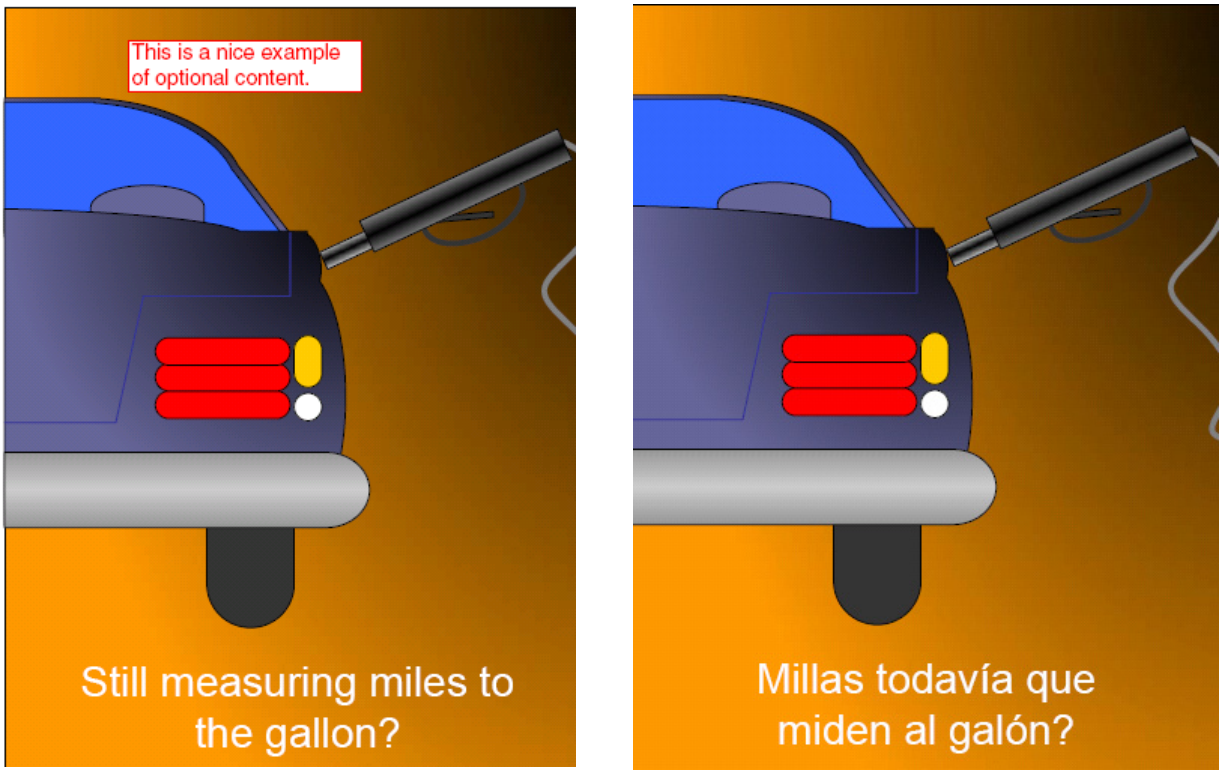
Note: Using markup annotations with an optional content group is very rare and not recommended.

In the following example, the free text markup annotation is attached to the optional content group defined for English (/backbone.xml#4).

Example 12.3

```
<Annotations xmlns="http://ns.adobe.com/pdf/2006">
   <FreeText … >
      <ContentGroup><Group ref="/backbone.xml#4"/></ContentGroup>
      <Contents>This is a nice example of optional content.</Contents>
      …
   </FreeText>
</Annotations>
```

As shown in the following figure, the free text annotation appears in the English version of the document, but not in the Spanish version.



13 Creating Portable Collections

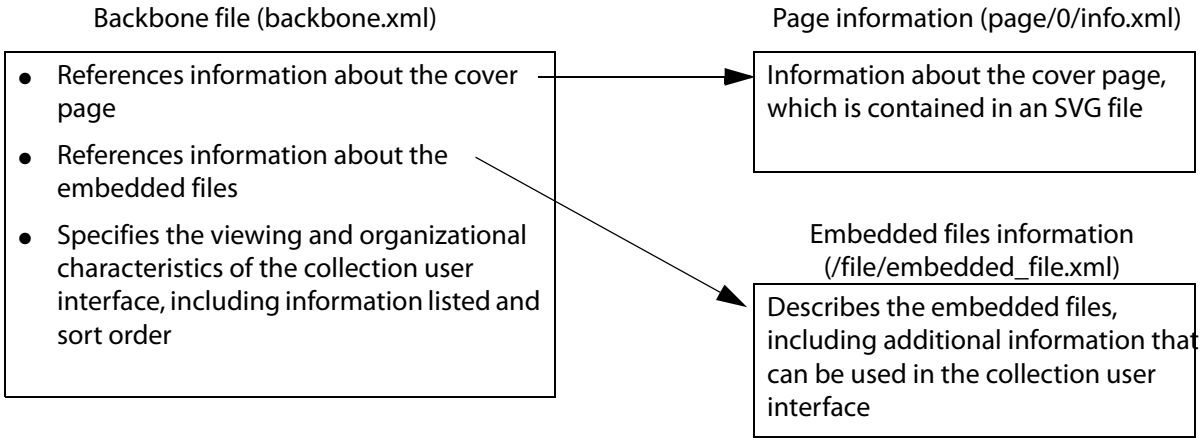
This chapter explains how to create a portable collection (PDF package) that includes multiple documents, including other Mars documents, PDF documents, and documents that use other standards. It contains the following information.

Topic	Description	See
"Understanding portable collections" on page 111	Explains what portable collections are and how they can be used.	page 111
"Specifying portable collections" on page 111	Explains how to specify portable collections in a Mars document.	page 111

Understanding portable collections

Mars documents can specify how a viewer application's user interface presents collections of file attachments, where the attachments are related in structure or content. Such a presentation is called a *portable collection* or a *PDF package*. The intended behavior of portable collections is view and search collections of related documents, such as email archives, photo collections, and engineering bid sets. Files in a collection are not required to have an implicit relationship or even a similarity; however, showing differentiating characteristics of related documents can be helpful for document navigation.

The following figure describes the files that contain information that represent a portable collection.



For more information about portable collections, see the *PDF Reference, version 1.7*.

Specifying portable collections

This section explains how to turn an existing Mars document into a portable collection, where the preexisting Mars document serves as the cover page for the collection. For guidance on creating a basic Mars document, see "Creating a Basic Mars Document" on page 14.

Planning

When planning portable collections, consider these issues:

1. Identify the files to use in the collection.
2. Identify the non-standard characteristics for the files in the collection. Standard characteristics (creation date, file name, and file size) apply to all files. Non-standard characteristics are unique to the files in a particular collection. For example, a collection of email messages could use sender and subject as non-standard characteristics, and a collection of photographs could use subject matter or photographer name as a non-standard characteristic.
3. Decide what information should be presented for each file in the collection user interface.
4. Identify the file information to use in determining the order in which the embedded files are listed in the collection user interface.

Embedding the files in the Mars document

For guidance on adding embedded files to a Mars document, see "Adding embedded files" on page 96.

Providing information about the embedded files

Information about each embedded file appears in the `EmbeddedFile` element and its child elements. Information about the file's name and size and appears as attributes in the `EmbeddedFile` and `EmbeddedFile/Param` elements; text descriptions appear in the `EmbeddFile/Desc` element; and additional information appears in the `EmbeddedFile/Collection` element.

➤ **To specify information about the embedded files in the embedded-files file (/file/embedded_files.xml):**

1. Ensure that the properties in the `EmbeddedFile`, `EmbeddedFile/Param` and `EmbeddedFile/Desc` elements are set to provide the information used in the collection.
2. For additional properties that describe the embedded files, add a `Collection` element as a child of the `EmbeddedFile` element.
3. For each additional property, add one `Field` element as a child of the `Collection` element. Specify the `Field` element's `Name` attribute as the name of the property. Specify the `Field` element's `Value` attribute as the value of the property. Specify the `Field` element's `type` attribute to `string` or `number`, depending on the data type. For example:
```
<Field Name="Control Number" Value="2222" type="string"/>
```

Describing the appearance of the information in the collection user interface

This section explains how to modify the backbone file to describe the viewing and organizational characteristics of the information presented in the collection user interface. These instructions augment the Mars file for the cover page to represent the collection.

➤ **To specify viewing information about the collection (backbone.xml):**

1. Open the backbone file in the Mars document that is used as the cover page.

2. Add a `Collection` element, as a child of the `PDF` element, setting the `Collection` element's `View` attribute to `Title`.

   This setting causes the viewing application to present the collection view in title mode, with each file in the collection denoted by a small icon and a subset of information from the `Schema` element. This subset is ordered to reflect settings in the `Collection/Schema` element (described later).

3. Add a `Schema` element as a child of the `Collection` element.

   This element provides information that the viewer uses to determine the information to present for each file in the collection. The information for each file in the collection resides in its file specification dictionary.

4. For each field that contributes information about the files in the collection, add a `Field` element as a child of the `Schema` element.

5. To each `Field` element, add the following properties:

   - `Name` attribute that specifies a field in the embedded file `Collection` element, which is located in the embedded-files file (/file/embedded_files.xml). The key name of each collection field dictionary is used to identify corresponding information in the embedded file specification dictionary. This attribute is not meaningful if the `Subtype` attribute specifies existing properties in the `EmbeddedFiles` element, such as the file name (`Subtype="F"`) or the modification date (`Subtype="ModDate"`).
   - `Subtype` attribute that specifies existing properties in the `EmbeddedFiles` element or that specifies the type of additional properties that the `FieldName` attribute specifies in the `Collections/Field` element. Values of `F`, `Desc`, `ModDate`, `CreationDate`, and `Size` specify existing properties in the `EmbeddedFiles` element. Values of `S`, `D`, and `N` respectifely specify string, date, and number fields from the embedded file `Collection` element.
   - `Editable` attribute that specifies whether the viewer application should provide support for editing the field value. A value of `true` indicates that the information displayed in the collection user interface is editable.
   - `FieldName` that specifies a textual field name that the viewer application can use to label field names.
   - `Order` attribute that specifies the relative order in which the field name appears in the user interface.
   - `Visible` attribute that specifies whether the the property is visible.

6. Add a `Sort` element as a child of the `Collection` dictionary.

   This element specifies the order in which the file in the collection should be listed in the collection user interface.

7. Add a `SortField` element as a child of the `Sort` element, specifying the name of the field that the viewer application uses to sort the items in the collection.

Example 13.1 shows a backbone file for a collection that has the following characteristics:

- Embedded files are described by the embedded_files.xml file located in the directory /file.
- Cover page is described by the info.xml file in the directory /page/0.
- Attached files are sorted according to their index value. The embedded-files file (/file/embedded_files.xml) assigns index values to individual embedded files, as shown in Example 13.2 on page 114.

- Attached files are described at the title level, which means that each file in the collection is denoted by a small icon and a subset of information specified by the `Schema` element.

Example 13.1 *Backbone file from a collection*

```
<PDF PageMode="UseAttachments" … >
   <EmbeddedFiles src="/file/embedded_files.xml"/>
   <ViewerPreferences Direction="L2R"/>
   <Metadata src="/META-INF/metadata.xml"/>
   <Pages>
      <Page src="/page/0/info.xml" x1="0" y1="0" x2="576" y2="360"/>
   </Pages>
   <Collection View="Title">
      <Sort>
         <SortField>Index</SortField>
      </Sort>
      <Schema>
         <Field Name="Name" Subtype="F" Editable="true"
            FieldName="File name" Order="1"/>
         <Field Name="Control Number" Subtype="S" Editable="true"
            FieldName="Control Number" Order="5"/>
         <Field Name="Description" Subtype="Desc" Editable="true"
            FieldName="Description" Order="2"/>
         <Field Name="Index" Subtype="N" Editable="true"
            FieldName="Index" Order="0" Visible="false"/>
         <Field Name="Size" Subtype="Size" FieldName="Size" Order="4"/>
         <Field Name="Modified" Subtype="ModDate"
            FieldName="Modified" Order="3"/>
      </Schema>
   </Collection>
</PDF>
```

Example 13.2 shows an embedded-files file that provides the following information about the files in the collection:

- Location, name, and other standard properties for each embedded file
- Non-standard properties called Index and Control Number

Example 13.2 *Embedded files for a collection*

```
<?xml version="1.0" encoding="UTF-8"?>
<EmbeddedFiles xmlns="http://ns.adobe.com/pdf/2006">
   <!-- Description of the first embedded file. -->
   <EmbeddedFile Key="1_ProductOverview.pdf" UName="1_ProductOverview.pdf"
      Name="1_ProductOverview.pdf">
      <Collection>
         <Field Name="Control Number" Value="2222" type="string"/>
         <Field Name="Index" Value="1" type="number"/>
      </Collection>
      <FileData src="/file/ef_330-1" FileType="application/pdf"/>
      <Desc>This show the hot new specs for our new product</Desc>
   </EmbeddedFile>
   <!-- Description of the second embedded file. -->
   <EmbeddedFile Key="Grant Application.pdf" UName="Grant Application.pdf"
      Name="Grant Application.pdf">
      <Collection>
```

```
        <Field Name="Control Number" Value="1111" type="string"/>
        <Field Name="Index" Value="3" type="number"/>
      </Collection>
      <FileData src="/file/ef_330-2" FileType="application/pdf"/>
      <Desc>If you can't afford the new product, fill this out.</Desc>
    </EmbeddedFile>
    <!-- Description of the third embedded file. -->
    <EmbeddedFile Key="Purchase Order.pdf" UName="Purchase Order.pdf"
      Name="Purchase Order.pdf">
      <Collection>
        <Field Name="Control Number" Value="2313" type="string"/>
        <Field Name="Index" Value="4" type="number"/>
      </Collection>
      <FileData src="/file/ef_330-3" FileType="application/pdf"/>
      <Desc>Use this to place an order</Desc>
    </EmbeddedFile>
  </EmbeddedFiles>
```

Part III: Adding Application Interchange to a Mars Document

15 Specifying Metadata

This chapter describes how to specify metadata using the Extensible Metadata Platform (XMP) standard. XMP provides a standard format for the creation, processing, and interchange of metadata, for a wide variety of applications.

This chapter contains the following information.

Topic	Description	See
Understanding XMP metadata	Background information on XMP metadata.	page 117
Specifying metadata	How to specify document-level metadata.	page 118

Understanding XMP metadata

Metadata is data that describes the characteristics or properties of a document. It never contains information that affects the appearance of the document.

Metadata can be associated with the entire document. It can also be associated with any of the files that comprise the document, such as the pages, fonts, images, page fragments (form XObjects), and ICC color profiles.

Mars uses the Extensible Metadata Platform (XMP) for its metadata. XMP is a widely adopted standard. For information on XMP data formatting, see the *Adobe XMP Specification*, located at: http://www.adobe.com/go/devnet. (Use the Search feature to find the title.)

XMP standardizes the definition, creation, and processing of metadata by providing the following structure:

- Data model - A useful and flexible way of describing metadata in documents.
- Storage model - The implementation of the data model. This includes the serialization of the metadata as a stream of XML and it includes XMP Packets, a means of packaging the data in files.
- Schemas - Predefined sets of metadata property definitions that are relevant for a wide range of applications, including all of the Adobe editing and publishing products, as well as for applications from a wide variety of vendors.

XMP metadata is encoded as XML-formatted text, using the W3C standard Resource Description Framework (RDF).

In Mars, metadata can be applied to an entire package and to individual resources within the package. In general, metadata about a specific resource (image, font or colorspace) is stored as a separate package object using the implicit metadata resource association rule. Package metadata is stored in the metadata.xml file in the /META-INF directory for the package. All other metadata must be in a file with an .xmp suffix, as defined in the resource association rule for metadata.

If metadata is stored separately from content, there is a risk that the metadata can be lost. The metadata should be associated with the file containing the content in the following way:

- Write the external file as a complete well-formed XML document, including the leading XML declaration.
- Place the metadata in the same directory as the content file. Use the same file name but change the file name extension to .xmp. This allows metadata to be added without modifying the document in other ways.
- If a MIME type is needed, use `application/rdf+xml`.
- Write external metadata as if it were embedded and then had the XMP Packets extracted and concatenated by a postprocessor.
- If possible, place the values of the `xmpMM:DocumentID`, `xmpMM:InstanceID`, or other appropriate properties within the file the XMP describes, so that format-aware applications can make sure they have the right metadata.

Metadata can be stored directly in the resource if the format of the resource supports it. Many formats, including PDF, XHTML, and SVG, have direct support for metadata. Others, such as JPEG, can use XMP for metadata support. In all cases, XMP is the recommended way to express metadata in an interoperable manner.

When metadata about a resource can be found in multiple locations, metadata found external to the resource has precedence over metadata in the resource itself. Care should be taken when a resource is extracted from a package. Depending on the application, it may be appropriate to extract the resource's metadata as well, even if the metadata is stored separately from the resource.

Specifying document-level metadata

To create the document-level metadata file for a Mars document, perform the following tasks:

1. Create the metadata.xml file in the /META-INF directory.

Note: Do not include the `?xpacket` declaration in this file.

2. Create the `x:xmpmeta` element as the root element, and specify a namespace definition. By convention, the namespace label is "`x`". For example:

```
xmlns:x="adobe:ns:meta/"
```

3. In `x:xmpmeta`, add XMP and private attributes as needed. All unrecognized attributes are ignored, and there are no required attributes. The only defined attribute is `x:xmptk`, whose value is the version of the Adobe XMP Toolkit.

4. Create an `rdf:RDF` element as a child of the `x:xmpmeta` element.

5. Create one or more `rdf:Description` elements, as children of the `rdf:RDF` element, and add elements and attributes as needed.

```
<x:xmpmeta xmlns:x="adobe:ns:meta/">
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
```

   - The `rdf:Description` element conventionally describes all properties from a given schema, and only that schema.

```
<rdf:Description rdf:about=""
   xmlns:pdf="http://ns.adobe.com/pdf/1.3/">
   <pdf:Producer>Acrobat Distiller 8.0.0 (Windows)</pdf:Producer>
</rdf:Description>
```

6. Write the XMP metadata to the file `/META-INF/metadata.xml` in the document package.

In the following example, the XMP metadata shown was taken from `metadata.xml` in the `META-INF` directory. The `xmlns:pdf` attribute defines the namespace prefix `pdf` to be used. This implies that all the properties within this `rdf:Description` element are `pdf` properties. There are a number of other `rdf:Description` elements shown in the example. Each one defines the properties for a given schema.

Example 15.1 ***Specifying XMP metadata (/META-INF/metadata.xml)***

```
<?xml version="1.0" encoding="UTF-8"?>
<x:xmpmeta xmlns:x="adobe:ns:meta/">
   <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
      <rdf:Description rdf:about="" xmlns:pdf="http://ns.adobe.com/pdf/1.3/">
         <pdf:Producer>Acrobat Distiller 8.0.0 (Windows)</pdf:Producer>
      </rdf:Description>
      <rdf:Description rdf:about=""
         xmlns:pdfx="http://ns.adobe.com/pdfx/1.3/">
         <pdfx:_EmailSubject>Files</pdfx:_EmailSubject>
         <pdfx:_AuthorEmailDisplayName>
            Lorie Furio
         </pdfx:_AuthorEmailDisplayName>
         <pdfx:_AdHocReviewCycleID>-1516059457</pdfx:_AdHocReviewCycleID>
         <pdfx:Company>Adobe Systems Incorporated</pdfx:Company>
         <pdfx:SourceModified>D:20060614050002</pdfx:SourceModified>
      </rdf:Description>
      <rdf:Description rdf:about="" xmlns:xap="http://ns.adobe.com/xap/1.0/">
         <xap:CreateDate>2006-06-13T22:00:27-07:00</xap:CreateDate>
         <xap:CreatorTool>Acrobat PDFMaker 8.0 for Word</xap:CreatorTool>
         <xap:ModifyDate>2006-07-06T16:23:56-07:00</xap:ModifyDate>
         <xap:MetadataDate>2006-07-06T16:23:56-07:00</xap:MetadataDate>
      </rdf:Description>
      <rdf:Description rdf:about=""
         xmlns:xapMM="http://ns.adobe.com/xap/1.0/mm/">
         <xapMM:DocumentID>uuid:92dd046e-25c6-4f7a-96f8-b703f1dd4bcd
            </xapMM:DocumentID>
         <xapMM:InstanceID>uuid:d8d1184b-3f96-478d-96f8-7d2682c35a10
            </xapMM:InstanceID>
         <xapMM:subject>
            <rdf:Seq>
               <rdf:li>11</rdf:li>
            </rdf:Seq>
         </xapMM:subject>
      </rdf:Description>
```

```
      <rdf:Description rdf:about=""
        xmlns:dc="http://purl.org/dc/elements/1.1/">
        <dc:format>application/pdf</dc:format>
        <dc:title>
          <rdf:Alt>
            <rdf:li xml:lang="x-default">Company Overview</rdf:li>
          </rdf:Alt>
        </dc:title>
        <dc:creator>
          <rdf:Seq>
            <rdf:li>Lori DeFurio</rdf:li>
          </rdf:Seq>
        </dc:creator>
      </rdf:Description>
    </rdf:RDF>
  </x:xmpmeta>
```

16 Specifying Marked Content, Logical Structure and Tagging

This chapter describes how to specify marked content, logical structure and tagging to a Mars document. These features build one upon the other. Marked content, the simplest feature, associates non-rendered information with groups of SVG content; logical structure associates a document structure and additional non-rendered information with groups of SVG content; and tagging builds on logical structure by providing a set of keywords that specify the role and class of particular types of marked content.

This chapter contains the following information.

Topic	Description	See
Understanding marked content	Describes marked content and explains how marked content is represented in Mars documents.	page 121
Understanding logical structure	Describes logical structure and explains the structure of marked content in Mars documents.	page 122
Understanding tagging	Describes tagging.	page 129
Creating a Mars document with marked content	Describes the steps required to add marked content to a Mars document.	page 130
Creating a Mars document with logical structure and tagging	Describes the steps required to add logical structure and PDF tagging to a Mars document.	page 131

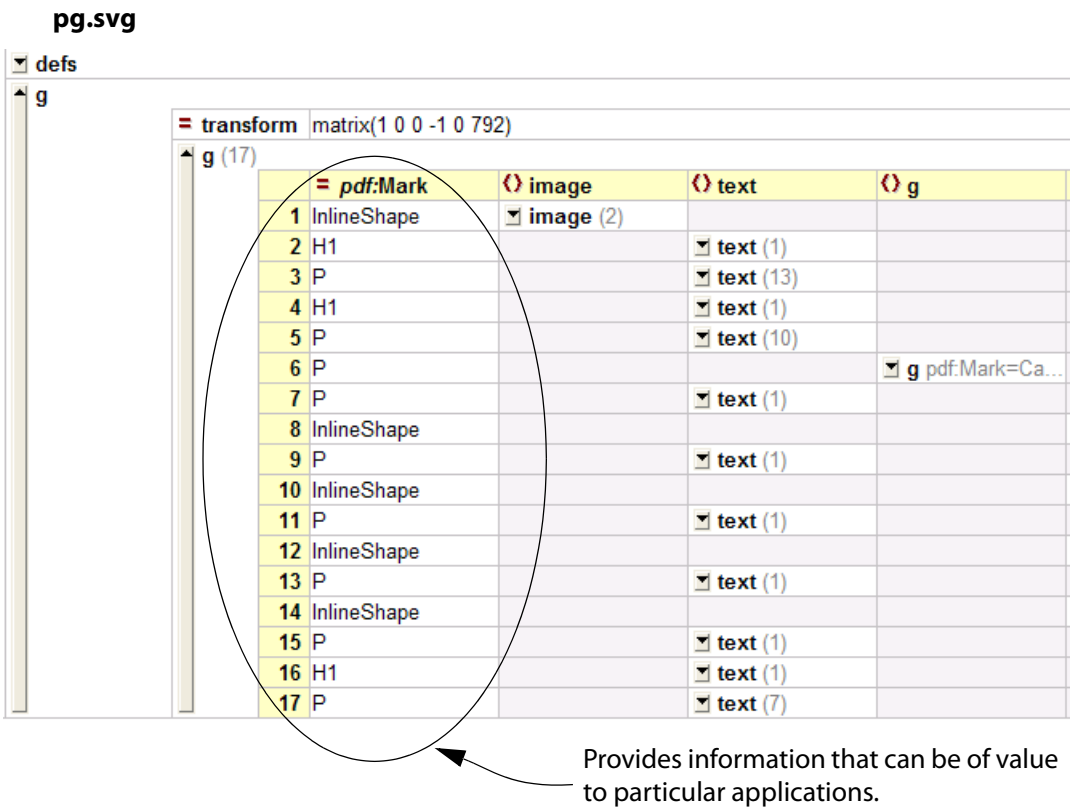
Understanding marked content

The marked content feature associates non-rendered information with groups of SVG content.

There are many uses for marked content. A graphics application, for example, might use marked content to identify a set of related objects as a group to be processed as a single unit. A text-processing application might use it to maintain a connection between a footnote marker in the body of a document and the corresponding footnote text at the bottom of the page.

The marked-content attribute (`pdf:Mark`) identifies the contents of SVG grouping elements (`g`) as being elements of interest to a particular application. The marked-content attribute can be accompanied by a set of user properties that further identify the contents of grouping elements.

The following diagram shows the addition of `pdf:Mark` attributes to the `g` elements in an SVG file for a page.



Understanding logical structure

Logical structure provides a mechanism for representing the structure of a document's content. A Mars application uses the logical structure to derive a *structure tree*. The structure tree is a model that provides a structured view of the otherwise linear document content. The page content is broken into chunks called content elements that appear at the leaves of the structure tree. The tree defines the order in which the content should flow. It can also provide data that the describes the roles of the content associated with the nodes in the tree.

Logical structure might include, for example, the organization of the document into chapters and sections or the identification of special elements such as figures, tables, and footnotes. The logical structure facilities are extensible, allowing applications that produce Mars documents to choose what structural information to include and how to represent it, while enabling PDF consumers to navigate a file without knowing the producer's structural conventions.

The logical structure of a Mars document is packaged with the document but is stored in files that are separate from its visible content. The association between the logical structure and the content it describes is established using XML references to marked content in the SVG files in the document. This separation allows the ordering and nesting of logical elements to be entirely independent of the order and location of graphics objects on the document's pages.

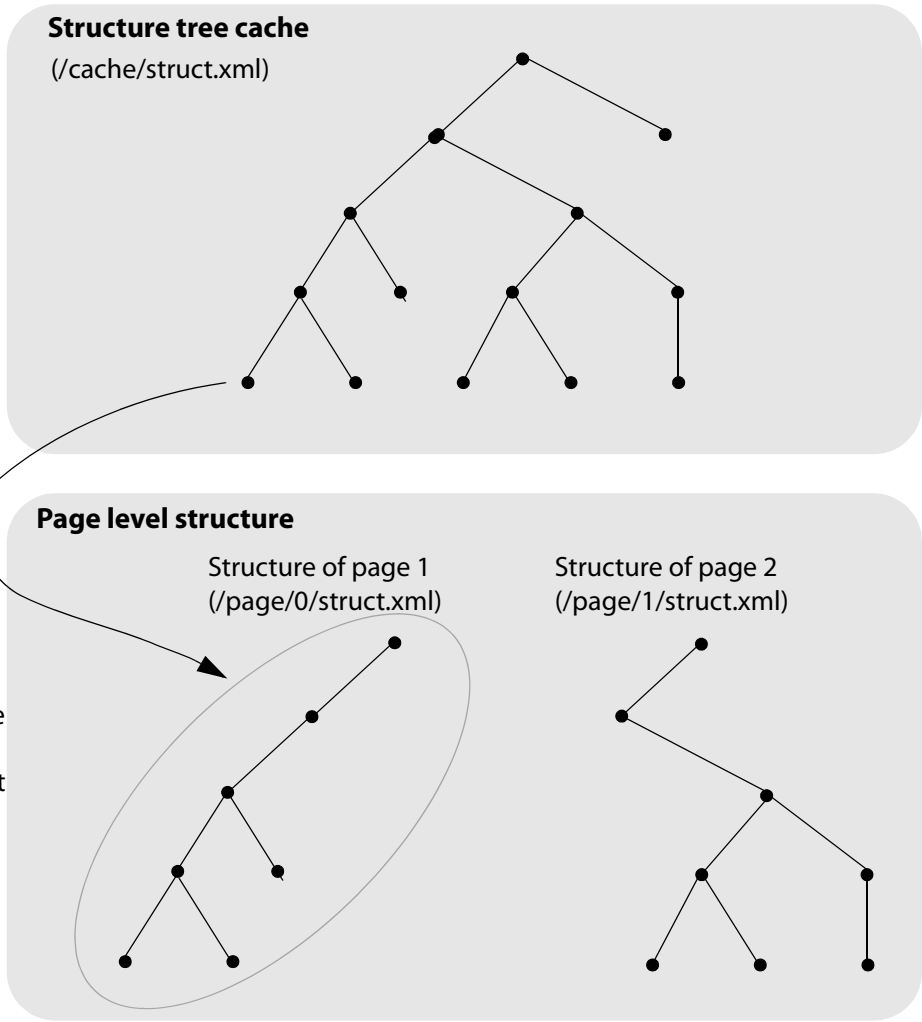
The following sections describe the logical (or hierarchical) representation and the association between that representation and the content.

Logical representation

Logical structure is represented in multiple page-level structure files and a single document-level structure cache. Each page-level logical file contains the structure for a single page. The document-level representation is a cache file that references the page-level structure files. The document-level structure cache can be reconstructed from the page-level logical structure files.

Relationship between files that describe logical structure

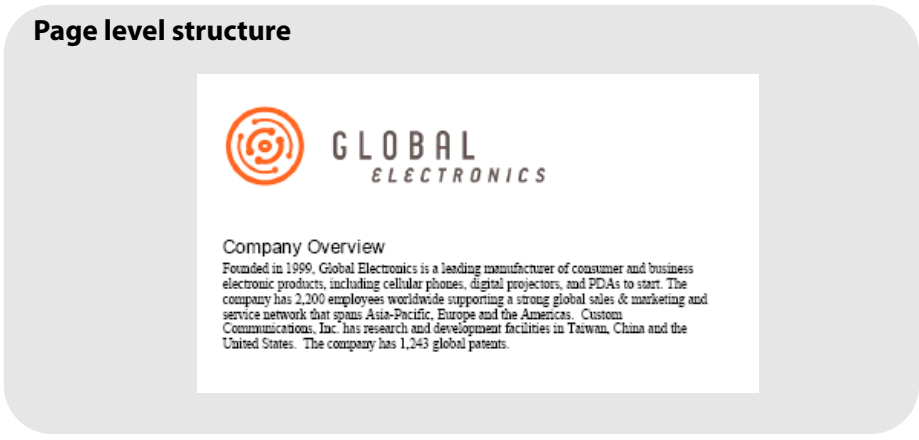
The structure cache represents all nodes in the document. Each node references a page-level structure file that contains the node it mirrors.



In the page-level structure files, intermediate nodes provide information about the node, such as its role and language.

Leaf nodes reference marked content within an SVG file.

The SVG file provides marked content identifiers.



Representing the logical structure at the page level and document level introduces some duplication of information; however, this duplication has benefits as discussed later. There is duplication between some of the nodes in the page-level logical structure. And there is complete duplication between the page-level and document level structure.

Duplication of page-level and document-level logical structure

The logical structure is duplicated between the document-level and page-level representations for the following reasons:

- The page representation is used to enable the breakdown and reassembly of the structure tree (document-level structure) in a predictable and reliable way in the face of page removal and insertion as part of document assembly
- The document-level structure is used for quick access to the entire document, without have to read every page-level file in the document in order to create the required part of the full structure tree.

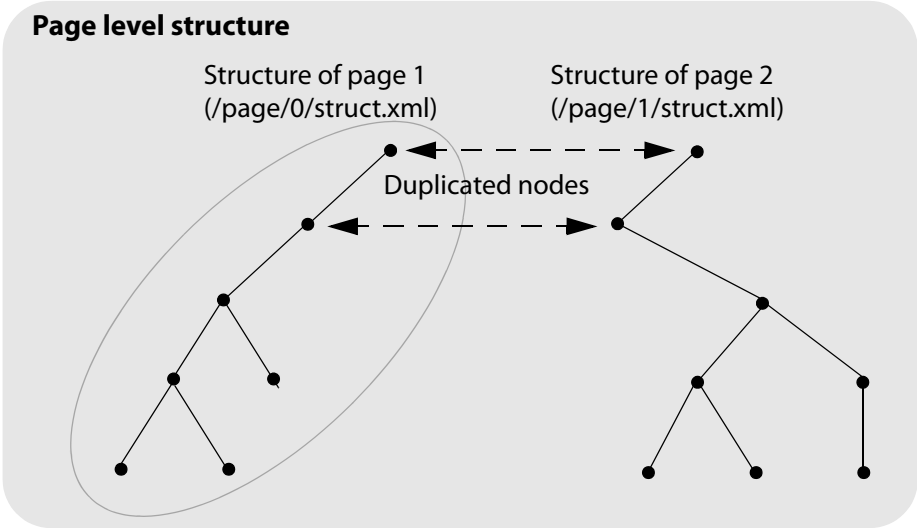
Duplication of nodes in the page-level logical structure

In a document with more than one page, intermediate nodes can appear in the page-level structure files for more than one page. When nodes are duplicated, the following rules apply:

- The page-level structure file referenced from the node description in cache structure file must contain the union of all information listed for a given structure element.
- The page-level structure file descriptions for the same node must not contain conflicting values for that structure element.

Duplication between page-level structure files

In the page-level structure files, some intermediate nodes can be duplicated.



Paths describe structure

A document's logical structure is expressed in the paths represented by the fully-qualified node names of the structure elements. The individual node names are separated by slashes. For example: `/1/1/1`. Individual node names must be unique among their siblings.

The following example shows a structure tree that uses descriptive names rather than the node names supported by Mars. The leaves in the structure tree appear in bold characters and their internal nodes appear in plain text.

Example 16.1 ***Descriptive structure tree (does not conform to Mars node-naming conventions)***

```
/Document
/Document/LetterHead
/Document/LetterHead/CorporateLogo
/Document/LetterHead/CorporateLogo/Part1
/Document/SectionTitle1
/Document/SectionTitle1/Leaf1
/Document/Section1
/Document/Section1/Leaf1
…
/Document/SectionX
/Document/SectionX/Figure1/
/Document/SectionX/Figure1/Graphic1
/Document/SectionX/Figure1/Graphic2
```

The following example translates the descriptive (but illegal) structure tree in Example 16.1 into node names that conform to the Mars node-naming convention ("Node naming convention" on page 127). The simplest node names are numbers, where the relative values of the numbers indicate the left-to-right order of sibling nodes. The name for each node must be unique among its siblings.

Example 16.2 ***Structure tree using numeric node names (conforms to the Mars node-naming conventions)***

```
/1
/1/1
/1/1/1
/1/1/1/1
/1/2
/1/2/1
/1/3
/1/3/1
…
/1/X
/1/X/1
/1/X/1/1
/1/X/1/2
```

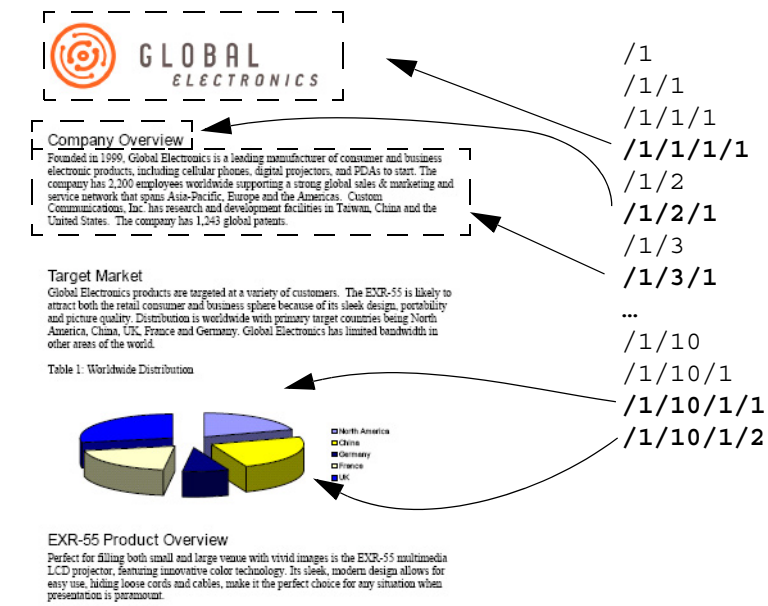
Connection between the structure tree and the content

The structure tree associates each node in the tree with the following levels of information:

- Intermediate nodes provide information about the node, such as its role and language. Intermediate nodes are represented as `Elem` elements in the document-level structure caches.
- Leaf nodes include references that link to marked content identifiers in the SVG. Leaf nodes are represented as `MCR` elements in the document-level structure caches.

The following diagram shows the links between the leaf nodes in the structure tree and the content to which they link.

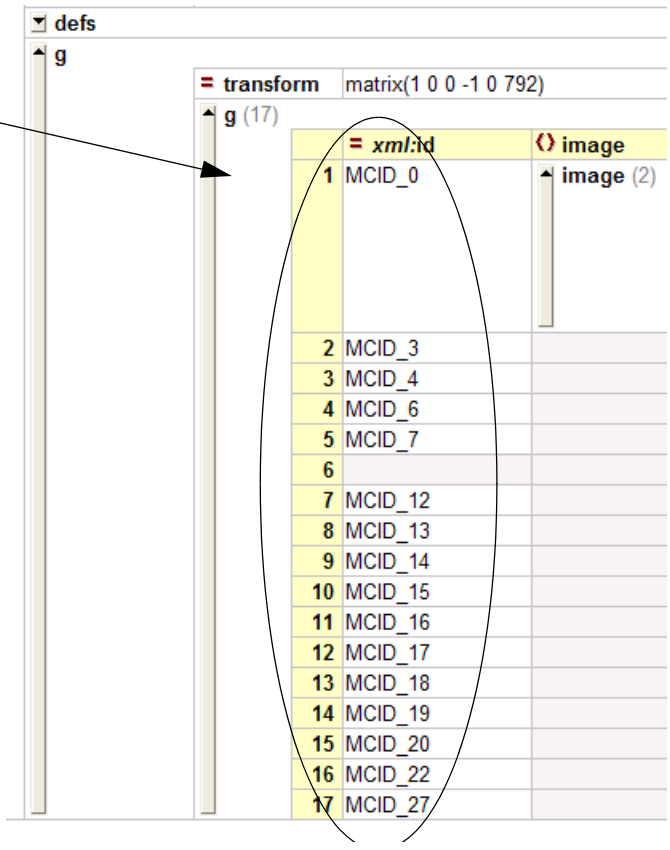
Association between the leaves in the structure and SVG content



The leaves in the logical structure reference marked content identifiers embedded in the SVG grouping (g) elements, as shown in the following example:

pg.svg

The structure tree leaf nodes use the marked content identifiers to associate the structure with the content.



Node naming convention

Mars specifies a more advanced method for naming nodes than the one described in the earlier sections. That convention uses a path comprised of single words, each one contributing the fully qualified name of the node. For example: `/1/1/1` or `/Document/LetterHead/CorporateLogo`.

This convention by itself does not work well if editing programs make incremental modifications that require the structure tree to be changed to retain the proper order of the content. Consider the following structure tree segment. Adding a new node that would logically occur between `/1/1/1/1` and `/1/1/1/2` would require the re-labeling of the existing nodes.

```
/1
/1/1
/1/1/1
/1/1/1/1
/1/1/1/2
```

To meet this need, Mars supports dot segments, which enable node names to be extended. This allows new nodes to be inserted between, before, or after any existing nodes without renaming the existing nodes. Applying a dot segment to the structure tree segment considered earlier results in the following structure.

```
/1
/1/1
/1/1/1
/1/1/1/1
/1/1/1/1.1
/1/1/1/2
```

Syntax

The following expressions provide the syntax of the dot segment naming convention:

```
NodeName -> (Series ":")? Segment (“.” Segment)* (“-” NodeTag)?
Segment -> Digit+
Digit -> “0” | ... | “9”
Series -> identifier
NodeTag -> identifier
identifier->any graphic Unicode character except “-”, “/”, “:”, and “.”
```

The following expressions are valid node names.

```
1.1.1
12
123.456.76543.345654.3321-Info
Ads:1
Ads:2
Ads:1.5
A:3.1-Sect
A:3.2-Sect
```

Determining left-to-right order of sibling nodes

The left-to-right order of sibling nodes is defined by comparing their `NodeName` expressions as follows:

1. Nodes are first sorted alphabetically by the `Series`. If the `Series` is not present, it is considered to be the null string which sorts before any non-null strings. Sorting "alphabetically" here is defined as comparing characters one at a time from first in the string to last in the string. Characters with greater Unicode character values are considered to come after characters with lesser Unicode values. If one string is a prefix of another, the longer one comes after the shorter one. If the two strings are equal, comparison continues with the next step.

2. The `Segments` are then compared by converting each Segment of digits to an integer and comparing those integers. Larger integers are considered greater than smaller ones. Leading zeros are ignored. If corresponding segment numbers are equal, comparison proceeds to the next segment. If only one `NodeName` has a next segment (because one `NodeName` is a prefix of the other), the longer `NodeName` is considered greater.

3. If two `NodeNames` are identical up to the trailing `NodeTag`, then the `NodeTags` are compared alphabetically in the same manner as the `Series`.

4. Although they should not appear, syntactically invalid `NodeNames` should be compared in the same manner as detailed in the previous steps if the invalid part is not required during an otherwise successful comparison. If an invalid segment is encountered it should be considered greater than the corresponding legal segment, regardless of the value of the legal segment.

The segment part of a `NodeName` may not consist of all zeros. For example, "0.0.0" or "0" are not allowed; "0.0.1" is allowed. This requirement guarantees that it is always possible to insert a new node with a `NodeName` before the given `NodeName`. A node named "0.0.0.1" would appear logically before "0.0.1".

The following expressions show the relative order of pairs of node names.

Comparison of node names	Explanation
1 < 002	Per Rule #1, both numbers are assumed to have null-value Series. Per Rule #2, the strings of the first segment are converted into a numeric value and compared.
1.1 < 2	Per Rule #1, both numbers are assumed to have null-value Series. Per Rule #2, the strings of the first segment are converted into a numeric value and compared. No evaluation of the second segment is necessary.
1.2.1.1 > 1.2.1	Per Rule #1, both numbers are assumed to have null-value Series. Per Rule #2, the node names are compared through the third segment, and then the longer of the names is judged to come after the shorter.
a:1.1 < b:1.1	Per Rule #1, the Series named "a" comes before the Series named "b".
1.1 < a:1.1	Per Rule #1, the first node name is assigned a null-value Series, which comes before the Series named "a".
1.23-beta < 1.23-betafish	Per Rule #3, the nodes are equivalent up to the "-" symbol and then their Node Tags are compared alphabetically.
1.2.3.4 < 1.2.xyzzy.4 (illegal value)	Per Rule #4, the nodes are compared using Rules #1-3. The character string "xyzzy" is considered greater than any numeric value.

Rational for Node Naming Syntax

The naming structure is defined to meet several requirements. First, and especially for machine generated node names, a simple integer as the name will often suffice. Thus, the root node would have the name "1", its first child would have the path "/1/1", its second child would have the path "/1/2" and so on.

This scheme by itself does not work well if editing programs make incremental modifications as insertion of a new structure node in the tree could require significant renumbering of nodes whose names may appear in a large number of locations. To meet this need, the "." segments are added. This allows new nodes to be inserted between, before, or after any existing nodes without renaming the existing nodes. One example of this is the insertion of a new section in a document.

Another likely manipulation of document structure is the merging of orthogonal material such as article content and advertising content in a publication. The structure tree for these two parts of the document would usually remain segregated. Although large gaps in the numbering of the root nodes could be used to achieve this, a more general approach would be to place the two trees effectively in different "name spaces" by giving the root node of one a string prefix of some sort (in this example, perhaps the root node would be named "ads:1"). This guarantees that the node names sort into disjoint groups.

Finally, the suffix allows addition of a human readable name string to help identify structural elements or the addition of a non-numeric key to supplement numeric ordering.

One key aspect of the naming structure is that the additional features do not complicate cases where they are not required.

Understanding tagging

Tagging builds on the logical structure framework. It defines a set of standard structure types and attributes that allow page content (text, graphics, and images) to be extracted and reused for other purposes.

Note: The term *tagging* represents a specific convention for tagging that was originally defined in the PDF file format. Although Mars has developed a distinct convention for representing such tagging, the intent is still true to its PDF counterpart. In general, a Mars feature is named using the term PDF only when that modifier connotes a specific meaning.

Tagging is intended for use by tools that perform the following types of operations:

- Simple extraction of text and graphics for pasting into other applications
- Automatic reflow of text and associated graphics
- Processing text for such purposes as searching, indexing, spell-checking, and accessibility
- Conversion to other common file formats (such as HTML, XML, and RTF)

Creating a Mars document with marked content

Creating a Mars document with marked content involves the following tasks:

- Planning for marked content
- Indicating the Mars document contains marked content
- Creating SVG that includes marked-content tags

Planning for marked content

Plan the marked content, by following these steps:

1. Determine the marking needs of the application that will be consuming the content. For example, if the marked content is intended for consumption by a CAD application, used marked content tags that are supported by that application.

2. Decide whether to include user properties with the marked-content tags. Such properties provide information beyond what is possible with the standard structure attributes. For example, some Mars producers, such as CAD applications, may use objects that have a standardized appearance, each of which contains non-graphical information that distinguishes the objects from one another. For example, several transistors might have the same appearance but different attributes such as type and part number.

Indicating the Mars document contains marked content

To indicate the Mars document contains marked content, change the backbone file as follows:

1. Add the `Marked` element as a child of the `PDF` element.

Creating SVG that includes marked-content tags

Marked content describes the SVG content that appears in an `svg:g` element. The marked content can include special properties, which can appear in the SVG file or in a separate file. By convention, the separate file is called properties.xml and is co-located with the SVG file it describes (/page/page_number) or is located with document-level resources (/page/

To create SVG that includes marked-content tags, perform the following tasks:

1. In the `svg:g` element that represents the content for a node, specify a `pdf:Mark` attribute whose value reflects the content and is meaningful to the consuming application.

2. If you are creating special properties that are inline in the SVG content, add a `pdf:Props` element immediately as the first child element in the `svg:g` element it describes. Set the `pdf:Props` element's properties as described in "Adding private data to logical-structure" on page 173.

3. If you are creating special properties that are separate from the SVG content, add a `pdf:Props` attribute to the `svg:g` element. Set the value of the `pdf:Props` attribute to an XPath expression that specifies a user property name in the properties file. For example:

   ```
   pdf:Props="/resource/properties.xml#CADView1"
   ```

   The element referenced by `pdf:Props` must be defined with properties as described in "Adding private data to logical-structure" on page 173.

For information on marking content for optional content, see "Marking SVG graphic content and text for inclusion in optional content groups" on page 108.

The following example applies marked-content tags to the graphics, headings and paragraphs used on an SVG page file.

Example 16.3 ***Marked content (/page/page_number/pg.svg)***

```
<?xml version="1.0" encoding="utf-8"?>
<svg>
   <defs/>
   <g >
      <g pdf:Mark="InlineShape">
         <pdf:Props>
            <Custom>
               <NumberKey Value="123.4" type="number"/>
               <BoolKey Value="true" type="boolean"/>
               <StringKey Value="StringVal" type="string"/>
            </Custom>
         </pdf:Props>
         <image xlink:href="/page/0/im_756-1.png" />
         <image xlink:href="/page/0/im_756-2.png" />
      </g>
      <g pdf:Mark="H1">
         <text>
            <tspan>Company Overview </tspan>
         </text>
      </g>
      <g pdf:Mark="P">
         <text>
            <tspan>Founded in 1999, Global Electronics is a leadi</tspan>
         </text>
         <text>
            <tspan>ng manufacturer of c</tspan>
         </text>
         …
      </g>
      <g pdf:Mark="H1">
         <text>
            <tspan>Target Market </tspan>
         </text>
      </g>
      …
   </g>
</svg>
```

Creating a Mars document with logical structure and tagging

Logical structure and tagging builds on the features of marked content, by specifying a structure that ties into the marked content element and that provides additional information about the content.

Adding structure to an existing Mars document involves the following tasks:

- "Planning the logical structure" on page 132

- "Specifying the structuring state and class mapping" on page 133
- "Identifying SVG content groups" on page 134
- "Creating a page-level structure file" on page 135
- "Referencing the page-level structure file in the page information file" on page 136
- "Creating a document-level structure cache" on page 137

Planning the logical structure

Plan the marked content, by following these steps:

1. Review the planning steps in "Planning for marked content" on page 130. Logical structure builds upon marked content.

2. Plan the structure for the individual pages in the document. The structure will be reflected in the paths defined in the page-level structure file. Familiarize yourself with the node naming convention described "Node naming convention" on page 127.

3. Decide whether to add a class map to the document. If you have many nodes that share the same class properties, such class maps may be preferable over specifying the class information separately for each node.

   If many structure elements share the same set of attribute values, they can be defined as an attribute class sharing the identical attribute object. Structure elements refer to the class by name. The association between class names and attribute objects is defined by an XML expression called the *class map*, kept in the `ClassMap` element of the `StructureTypes` element in the document backbone. Each `Class` element in the class map has a `Name` attribute denoting the name of a class. The following is an example a node definition that references a named class map.

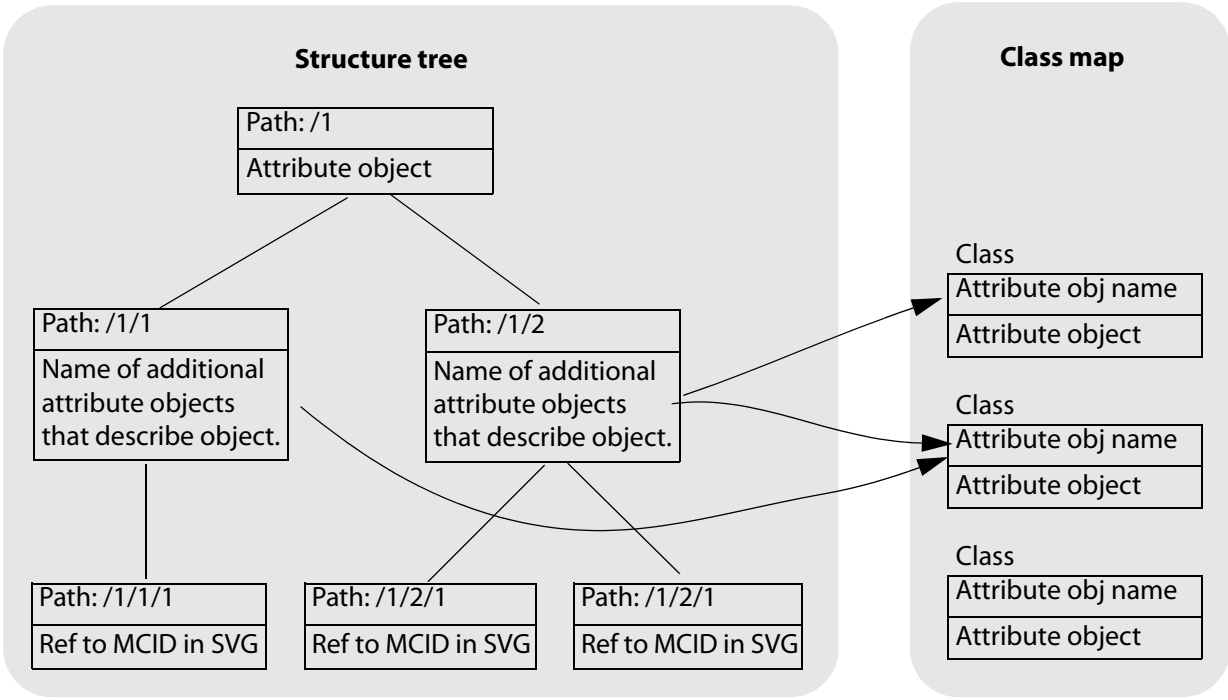
   The following example shows a node that specifies a mapped class attribute object. The appearance of the `Name` element in the `Class` element specifies this class map object. There must be a corresponding entry in the `ClassMap` element in the backbone file.

Example 16.4 ***Node associated with attributes contained in the class map***

```
<Structure>
   <Elem Path="/1" Role="Sect" Tag="Sect" >
      <Class Name="MyProductionInformation"/>
   </Elem>
   …
</Structure>
```

The following illustration shows a simple structure where the root node contains an `Attribute` element that provides an attribute object. This object contains properties such as `Owner`, `UserProperties` and `RubyPosition`. The intermediate nodes /1/1 and 1/2 also contain an `Attribute` element, but rather that providing attribute objects, they map to an attribute class specified in the class map.

Class map makes attribute objects globally-available to nodes



4. Decide whether to add role maps to the document.

   The role map provides the mapping of author-defined structure tags to standard structure tags. Role maps associate the names of structure types used in the document with their approximate equivalents in the set of standard structure types. Standard structure types characterize the role of a content element within the document and, in conjunction with the standard structure attributes, how that content is laid out on the page.

Specifying the structuring state and class mapping

To specify the structuring state and class mapping, include the following properties in the backbone file:

1. Add the `Marked` element as a child of the `PDF` element.
2. If the structure includes special user properties, add the `UserProperties` attribute to the `Marked` element, setting its value to `true`. For example: `UserProperties="true"`.
3. If role or class maps are desired, add a `StructureTypes` element as a child of the `PDF` element.
4. If a role map is desired, add a `RoleMap` element as a child of the `StructureTypes` element.
5. For each role association, add a `Role` element to the `RoleMap` element, specifying the following attributes:
   - `Name` attribute whose value is an author-defined name used in the document's marked content, as the value of the SVG grouping element's `pdf:Marked` attribute.
   - `MapTo` attribute whose value specifies the standard structure tag that corresponds to the named marked content
6. If a class map is desired, add a `ClassMap` element as a child of the `StructureTypes` element.

7. For each top-level class association, add a `Class` element to the `ClassMap` element, specifying the following properties:
   - `Class` attribute whose value is an author-defined name that can be used in the intermediate nodes of the page-level structure file
   - `Attribute` or `Attributes` element whose properties specify the attributes to associate with the attribute class

Example 16.5 ***Role Map (/backbone.xml)***

```
<PDF>
   <Marked/>
   <StructureTypes>
      <RoleMap>
         <Role Name="Body Text 2" MapTo="P"/>
         <Role Name="InlineShape" MapTo="Figure"/>
         <Role Name="DropCap" MapTo="Figure"/>
         <Role Name="Outline" MapTo="Span"/>
         <Role Name="Subscript" MapTo="Span"/>
         <Role Name="Superscript" MapTo="Span"/>
         <Role Name="TOA" MapTo="TOC"/>
         <Role Name="TOF" MapTo="TOC"/>
         <Role Name="Strikeout" MapTo="Span"/>
         <Role Name="Table Text" MapTo="P"/>
         <Role Name="TextBox" MapTo="Div"/>
         <Role Name="Heading 1" MapTo="P"/>
         <Role Name="Heading 2" MapTo="P"/>
         <Role Name="Normal" MapTo="P"/>
         <Role Name="Endnote" MapTo="Note"/>
         <Role Name="Footnote" MapTo="Note"/>
         <Role Name="Underline" MapTo="Span"/>
         <Role Name="TOFI" MapTo="TOCI"/>
         <Role Name="Frame" MapTo="Div"/>
         <Role Name="Shape" MapTo="Figure"/>
         <Role Name="TOAI" MapTo="TOCI"/>
         <Role Name="Normal (Web)" MapTo="P"/>
      </RoleMap>
   </StructureTypes>
</PDF>
```

Identifying SVG content groups

The leaf nodes (`MCR` elements) in the page-level structure files include `ref` attributes to specify the content associated with the node. For example: `ref="/page/1/pg.svg#MCID_0"`. The first part of this reference indicates the SVG page containing the marked content and the second indicates an identifier that appears in the SVG content groups (`svg:g`) associated with the nodes. This identifier uses the `xml:id` attribute. For example: `<g pdf:Mark="P" xml:id="MCID_0"/>`.

To identify the SVG content groups, include identifiers in each SVG file as follows:

1. In each `svg:g` element that houses content associated with a leaf-level node, specify an `xml:id` attribute whose value provides an identifier for the element. The identifier must be unique within the SVG file.
2. Specify other marking or properties as needed, as described in "Creating SVG that includes marked-content tags" on page 130.

The following example shows the inclusion of marked content identifiers in an SVG file. These identifiers appear in bold letters. The first identifier (`MCID_0`) marks the `svg:g` element that contains the graphics used for the corporate logo shown on page 126.

Example 16.6 ***Marked content identifiers added to an existing SVG file (/page/page_number/pg.svg)***

```
<g>
   <g pdf:Mark="InlineShape" xml:id="MCID_0">
      <image … />
      <image … />
   </g>
   <g pdf:Mark="H1" xml:id="MCID_3">
      <text>
         <tspan>Company Overview </tspan>
      </text>
   </g>
   <g pdf:Mark="P" xml:id="MCID_4">
      <text>
         <tspan>Founded in 1999, Global Electronics is a leadi</tspan>
      </text>
      <text>
         <tspan>ng manufacturer of c</tspan>
      </text>
      …
   </g>
   …
</g>
</svg>
```

Creating a page-level structure file

To create a page-level structure file, do the following:

1. Create an XML file called struct.xml located in the /page/*page_number* directory. Ensure the first line in the file is the standard XML declaration: `<?xml version="1.0" encoding="UTF-8"?>`
2. Add a `Structure` element, as the root element in the file.
3. For each internal node on the page, add an `Elem` element as a child of the `Structure` element.
4. In each `Elem` element, add the following attributes:
   - `Path` attribute that identifies where the node fits into the structure tree. See the discussion in "Paths describe structure" on page 124 and "Node naming convention" on page 127.
   - `Page_src` attribute that references the information file for the page associated with the internal node.
   - Other attributes, such as `Class`, `Role` or `Tag`. If you add the name of a `Class` or `Role` attribute, ensure you create the corresponding mapping structure in the backbone file. (See "Specifying the structuring state and class mapping" on page 133.)
5. For each leaf in the structure, create an `MCR` element as a child of the `Structure` element.

6. In each `MCR` element, add the following attributes:
   - `Path` attribute that identifies where the node fits into the structure tree. See discussion in "Paths describe structure" on page 124 and "Node naming convention" on page 127.
   - `ref` attribute that references the marked content identifier in the SVG file described by the `Elem` element for the parent node.
   - `Page_ref` attribute, if the referenced content appears on a different page from the one referenced in the parent `Elem` element. This can happen if the parent `Elem` element describes content split over multiple pages,

The following example shows the page-level structure tree for the page shown on page 126. The `Elem` and `MCR` elements' `Path` attributes establish the logical structure of the document. The `Elem` elements provide information about the interior nodes, and the `MCR` elements reference the marked content identifiers in the SVG content.

Example 16.7 ***Page-level structure file (/page/page_number/struct.xml)***

```
<?xml version="1.0" encoding="UTF-8"?>
<Structure>
   <Elem Path="/1" Role="Sect" Tag="Sect"/>
   <Elem Path="/1/1" Page_src="/page/0/info.xml" Role="P" Tag="Heading 1"/>
   <Elem ActualText="Global Electronics Logo"
      Alt="Global Electronics Logo" Path="/1/1/1" Page_src="/page/0/info.xml"
      Role="Figure" Tag="InlineShape"/>
   <MCR Path="/1/1/1/1" ref="/page/0/pg.svg#MCID_0"/>
   <Elem Path="/1/2" Page_src="/page/0/info.xml" Role="H1" Tag="H1"/>
   <MCR Path="/1/2/1" ref="/page/0/pg.svg#MCID_3"/>
   <Elem Path="/1/3" Page_src="/page/0/info.xml" Role="P" Tag="P"/>
   <MCR Path="/1/3/1" ref="/page/0/pg.svg#MCID_4"/>
   …
   <Elem ActualText="Pie Chart Distribution of Sales"
      Alt="Pie Chart Distribution of Sales" Path="/1/7/2"
Page_src="/page/0/info.xml"
      Role="Figure" Tag="InlineShape"/>
   <MCR Path="/1/7/2/1" ref="/page/0/pg.svg#MCID_11"/>
   <MCR Path="/1/7/2/2" ref="/page/0/pg.svg#MCID_12"/>
   <MCR Path="/1/7/2/3" ref="/page/0/pg.svg#MCID_13"/>
   <MCR Path="/1/7/2/4" ref="/page/0/pg.svg#MCID_14"/>
   <MCR Path="/1/7/2/5" ref="/page/0/pg.svg#MCID_15"/>
   <MCR Path="/1/7/2/6" ref="/page/0/pg.svg#MCID_16"/>
   <MCR Path="/1/7/2/7" ref="/page/0/pg.svg#MCID_17"/>
   <MCR Path="/1/7/2/8" ref="/page/0/pg.svg#MCID_18"/>
   <MCR Path="/1/7/2/9" ref="/page/0/pg.svg#MCID_19"/>
   <MCR Path="/1/7/2/10" ref="/page/0/pg.svg#MCID_20"/>
   …
</Structure>
```

Referencing the page-level structure file in the page information file

To specify the page-level structure file in the page information file, specify the following properties in the page information file (/page/*page_number*/info.xml):

1. Create a `Structure` element as a child of the `Page` element, and set its `src` attribute to the path of the page-level structure file.

The following example specifies the location of the page-level structure file.

Example 16.8 ***Page information file references page-level structure file***

```
<Page xmlns="http://ns.adobe.com/pdf/2006">
   <Contents src="/page/1/pg.svg"/>
   <Structure src="/page/1/struct.xml"/>
</Page>
```

Creating a document-level structure cache

While the breakdown of the structure tree and its distribution among the pages is sufficient for document assembly and content extraction, operations that traverse the document using the structure tree are more efficient if the shape of the structure tree is known without having to read each page's structure file. Caching the overall structure tree information improves performance by eliminating the need to read all pages in advance. The structure tree is stored in the document cache directory /cache/struct.xml.

The cache contains an entry for each interior structure tree node indicating which page contains information about it. Since more than one page may contain the same information about a node, any such page can be used in the cache entry. The cache itself lists all of the structure tree nodes and their paths; this is sufficient information to allow a traversal of the tree. Individual nodes can then be filled in on demand.

Cache files are either predefined or self-describing.

A predefined cache is one that relies on the Mars processing application to regenerate the data in the cache. Typically, the data is regenerated using a standard XSLT script that operates on the files specified by a pattern.

A self-describing cache is one whose cached data can be recomputed using only information in the cache file itself and the contents of the document package. Self-describing caches provide an extensible cache mechanism. Part of the cache file provides a pattern which selects a set of files in the package. A second part of the cache file specifies an xpath or limited xslt expression to be applied to the selected files. The final part of the cache file (the data part) contains the XML which is a result of the evaluation of the expression, query, or script.

Creating a document-level structure cache

To create a predefined document-level structure cache, perform the following tasks:

1. Create an XML file called struct.xml located in the /cache directory. Ensure the first line in the file is the standard XML declaration: `<?xml version="1.0" encoding="UTF-8"?>`
2. Create the `Cache` element as the root element, specifying the default namespace as follows:

   ```
   xmlns="http://ns.adobe.com/pdf/2006"
   ```

3. Create a `Query` element as a child of the Cache element.
4. Create a `Files` element as a child of the `Query` element.
5. Create a `Pattern` element as a child of the `Files` element, and set its `Value` attribute to specify a file-selection pattern used to determine which page-level files to consider. A file-selection pattern is a prototype path that can have the wildcard character "*". The patterns are matched against the fully

qualifies names of package files. A package file is matched by a pattern if its fully qualified name, represented as a string, matches a pattern. For example:

```
<Pattern Value="/page/*/struct.xml"/>
```

6. Create additional `Pattern` elements as needed to fully describe the files that contribute to this cache.

7. If the cache is predefined, set the `Cache` element's `Identifier` attribute to the value shown below:

```
<Cache xmlns="http://ns.adobe.com/Mars"
   Identifier="http://ns.adobe.com/pdf/2006/cache/structure">
```

8. If the cache is self-describing, add a `Query` element as a child of the `Cache` element, and set the element's attributes to provide XSLT and/or XPath expressions that establish the method this cache file uses to be self-describing. The expressions are applied to the files identified in the `Pattern` attribute created in step 5.

   - To specify an XPath expression, add the `XPath` attribute to the `Query` element. For example:

     ```
     <Query XPath="/cache" />
     ```

   - To specify a custom XSLT script, add the `Translate` attribute, setting its value to the location of the script. For example:

     ```
     <Query Translate="MyStructureTranslator.xslt" />
     ```

9. Create a `Data` element as a child of the `Cache` element. The `Data` element must reflect information from each page-level structure file in the Mars document, as follows:

   - For each unique `Elem` element in each page-level structure file, create a corresponding `Elem` element. Ensure nodes are represented only once in the cache. Set the `Path` attribute to the value of the corresponding page-level attribute. Set the `ref` attribute to specify the page information file that provides the most complete description of the node. For more information, see "Duplication of nodes in the page-level logical structure" on page 124.

   - For each `MCR` element in each page-level structure file, create a corresponding `MCR` element. Set the `Path` attribute to the value of the corresponding page-level attribute.

The following example shows a document-level strcture cache. The appearance of the `Identifier` attribute indicates the cache file uses the predefined method for regenerating the contents of the `Data` element. That is, it relies on the Mars processing application to regenerate the contents of that element.

The nodes represented by the first set of `Elem` and `MCR` elements reference the page-level logical structure file for the first page in the document. The last set of `Elem` and `MCR` elements reference the page-level logical structure file for the third page in the document.

Example 16.9 ***Predefined document-level structure cache (/cache/struct.xml)***

```
<?xml version="1.0" encoding="UTF-8"?>
<Cache xmlns="http://ns.adobe.com/pdf/2006"
   Identifier="http://ns.adobe.com/pdf/2006/cache/structure">
   <Query>
      <Files>
         <Pattern Value="/page/*/struct.xml"/>
      </Files>
   </Query>
   <Data>
      <Elem Path="/1" ref="/page/0/struct.xml"/>
      <Elem Path="/1/1" ref="/page/0/struct.xml"/>
      <Elem Path="/1/1/1" ref="/page/0/struct.xml"/>
```

```
      <MCR Path="/1/1/1/1" ref="/page/0/struct.xml"/>
      <Elem Path="/1/2" ref="/page/0/struct.xml"/>
      <MCR Path="/1/2/1" ref="/page/0/struct.xml"/>
      <Elem Path="/1/3" ref="/page/0/struct.xml"/>
      <MCR Path="/1/3/1" ref="/page/0/struct.xml"/>
      <Elem Path="/1/4" ref="/page/0/struct.xml"/>
      <MCR Path="/1/4/1" ref="/page/0/struct.xml"/>
      …
      <Elem Path="/1/14/1/30" ref="/page/2/struct.xml"/>
      <Elem Path="/1/14/1/30/1" ref="/page/2/struct.xml"/>
      <Elem Path="/1/14/1/30/1/1" ref="/page/2/struct.xml"/>
      <MCR Path="/1/14/1/30/1/1/1" ref="/page/2/struct.xml"/>
      <Elem Path="/1/14/1/30/2" ref="/page/2/struct.xml"/>
      <Elem Path="/1/14/1/30/2/1" ref="/page/2/struct.xml"/>
      <MCR Path="/1/14/1/30/2/1/1" ref="/page/2/struct.xml"/>
   </Data>
</Cache>
```

17 Creating Forms

This chapter describes how to create interactive forms.

This chapter contains the following information.

Topic	Description	See
Understanding forms	Introduces XFA-based forms and Acrobat forms.	page 140
Creating XML forms	Explains how to incorporate an existing XML form into a Mars document.	page 148
Creating Acrobat forms	Explains how to create a simple Acrobat form.	page 160

Understanding forms

A Mars document can include either an *XML form* or an *Acrobat form*. An XML form is based on the conventions described in the *Adobe XML Forms Architecture (XFA) Specification, v2.4*. Both types of forms are interactive and both use a collection of fields intended for gathering information interactively with the user. A Mars document can contain a number of fields appearing on any combination of pages, all of which make up a single, global interactive form spanning the entire document.

XML and Acrobat forms consist of two main parts: *form fields* and *form field widgets*. The form fields define each of the fields and provide information about the data they contain. Form field widgets define the visible manifestation of a form field. That is, they define what should be drawn on the page to show the form field. Form fields and form field widgets may be in one-to-one correspondence, or there may be more than one widget per field, for example, if the "Name" field appears on more than one page. A field also may not be displayed at all in which case there is no widget associated with it.

XML forms build on Acrobat forms by providing additional capabilities for mapping XML data to form field values, and providing a much richer scripted behavior model with more capabilities than is available with Acrobat forms alone.

The backbone file specifies general information about the form, and it specifies the form fields. Each form field contains a reference to the widget that describes the field's appearance, and in the case of XML forms each form field provides a reference to the template field it represents. The form fields specify the behavior of the field and optionally provide data (such as choice lists or default values) used to populate the field. (See the figure on page 141.)

Note: An XML form can be either static or dynamic. In the case of a *static XML form*, the layout of the form does not vary regardless of the amount of data entered by a user. In the case of a *dynamic XML form*, the layout changes to accommodate changes in the amount of data that must be represented by the form. This chapter addresses only static XML forms.

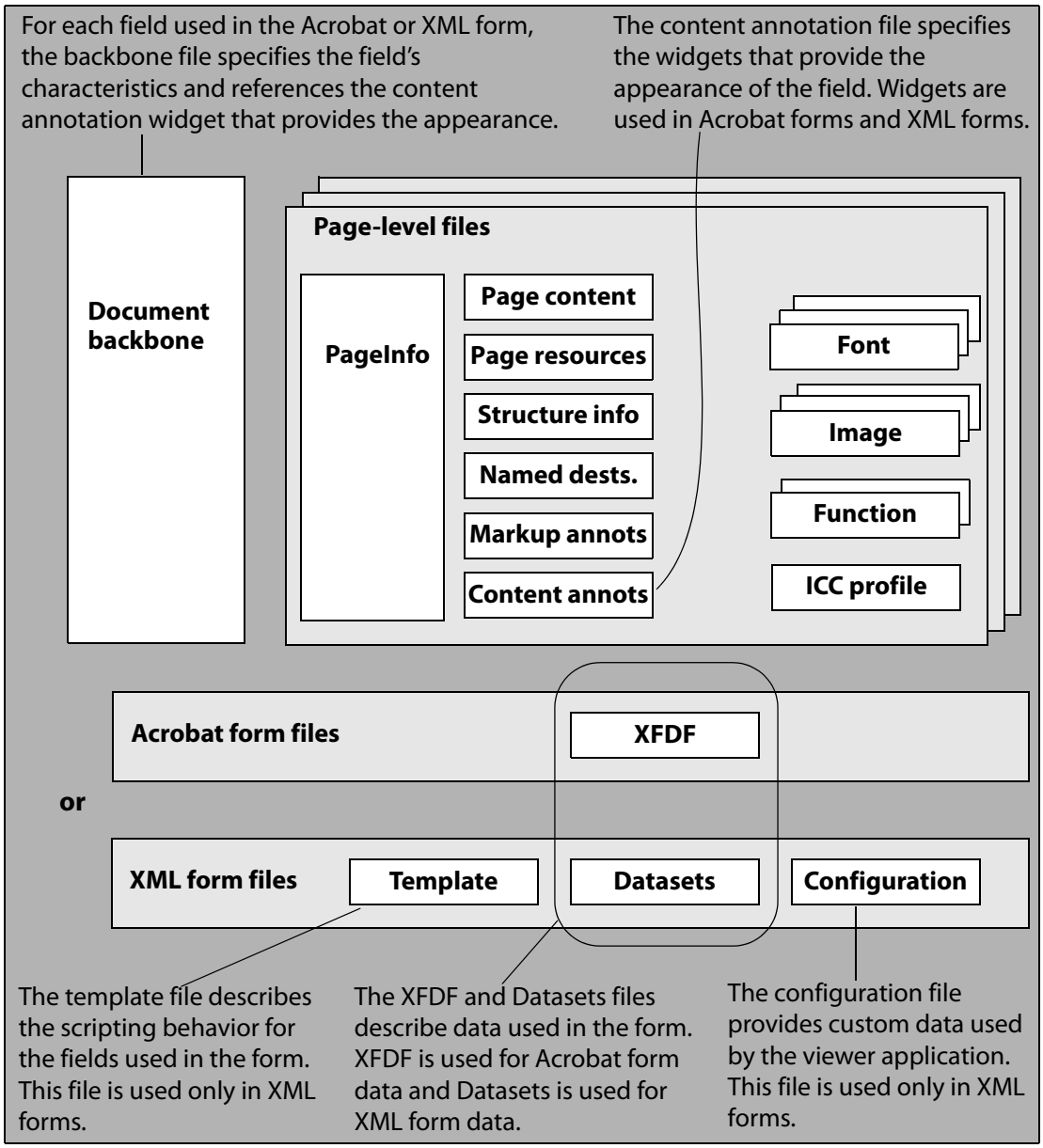
The form fields specified in the backbone file duplicate certain characteristics of the fields defined in the XML template file. Those duplicated characteristics are the features that XML and Acrobat forms have in common. This duplication enables applications to use a common architecture in handling both types of form fields. The XML form template characteristics that are not common with Acrobat forms are represented only in the template file. Dynamic XML forms cannot take advantage of these common features because the field information is unknown in advance of the user interacting with the form.

Each page containing form fields must have a content annotations file (pg.can) which contains an XML description of the fields. The content annotation file is referenced from the page's page information file (info.xml). The widgets specify appearance characteristics and can reference files containing graphics or SVG content.

In the case of XML forms, the individual XML documents that comprise the form are placed in the /form directory. The template file contains the original specification for the fields (now duplicated in the backbone), and it specifies scripting behavior and other characteristics. The datasets file contains data associated with the form. When the XML form is first opened in a viewer application, this data is used to populate the displayed form. If the user provides new data, the datasets file is updated with the newer data.

In the case of Acrobat forms, the /form directory can contain an XML Forms Data File (XFDF) that provides data used in the form (form_data.xfdf).

Structure of a Mars document containing a form (partial)



Form fields

The backbone `AcroForm` element is the starting point for all form-related data. It's child `Fields` element specifies the form fields. A field can be specified using any one of the elements: `Button`, `Choice` or `TextField`. Fields specified using the `Signature` element are not yet supported.

A field specified by the `Button` element represents an interactive control on the screen that the user can manipulate with the mouse or keyboard. There are three types of button fields:

- A push button, which is a purely interactive control that responds immediately to user input without retaining a permanent value
- A check box, which toggles between two states, on and off
- Radio button fields, which contain a set of related buttons that can each be on or off. Typically, at most one radio button in a set may be on at any given time, and selecting any one of the buttons automatically deselects all the others.

A field specified by the `TextField` element is a box or space in which the user can enter text from the keyboard. The text may be restricted to a single line or may be permitted to span multiple lines.

A field specified by the `Choice` element contains several text items, one or more of which may be selected as the field value. The items may be presented to the user in either of two forms:

- A scrollable list box
- A combo box consisting of a drop-down list optionally accompanied by an editable text box in which the user can type a value other than the predefined choices

Representing XML forms in a Mars document

This section explains how XML forms are represented in a Mars document.

Form parts

The `XFA` element in the `AcroForm` element can specify the XML form as a collection of form part files (using a series of `FormPart` elements).

The `XFA` element contains a series of `FormPart` elements describing individual XML files that make up the `XFA` resource. At a minimum, the `FormPart` element must include references to the form template and datasets files:

- The template form file describes the characteristics of the form, including its fields, calculations, validations, and formatting.
- The datasets form file represents the current data values in the XML form.

The names of the files containing the form parts can be arbitrary, except that the files containing the form parts XFA template, XFA datasets, and XFA config must be named `template`, `datasets`, and `config`, respectively.

Each `FormPart` elements references a file located in the Mars package. The first and last `FormPart` elements reference files that contain the begin and end tags for the `xdp:xdp` element. The other `FormPart` elements reference files that contain a part of the XML form, such as the XFA template document or the dataset document.

Each of the XML documents referenced by the `FormPart` elements must be located in the /form folder.

When an `XFA` element is present in the `AcroForm` element, the `XFA` element provides all the information about the form-related events such as calculations and validations. The other entries in the `AcroForm`

element must be consistent with the information in the `XFA` element. When creating or modifying a Mars file with an `XFA` element, your applications should follow these guidelines:

- Mars interactive form field objects must be present for each field specified in the `XFA` resource. The Acrobat form fields must be consistent with the type of the field in XML form template.
- The XFA Scripting Object Model (SOM) specifies a naming convention that must be used to connect interactive form field names with field names in the XFA resource. Information about this model is available in the *Adobe XML Forms Architecture (XFA) Specification, v2.4*.
- No `Action` elements should be present in the annotation elements of fields that also have actions specified by the `XFA` resource. The behavior of a field whose actions are specified in both ways is undefined.

Relationship between the backbone, template and content annotations

For static XML forms, the structure of the backbone `AcroForm` element must reflect the structure in the template. In particular, the backbone `AcroForm` element must have a hierarchy of `Container` elements that mirrors the structure of the template container elements subordinate to the `template` element. Only the template container elements (or their ancestors) that include interactive fields are represented in the backbone `AcroForm` element. An interactive field is any template `field` element that has a `ui` child element.

Each backbone `Container` element must include a `Name` attribute that specifies a SOM expression that references its template counterpart. If a template container or its descendents includes interactive fields, the corresponding backbone `Container` element must have a child `SubFields` element.

For each backbone `Container` element whose template counterpart has interactive fields, that `Container` element's child `SubFields` element must have one `Button`, `TextField`, or `Choice` element for each of the interactive fields.

The choice between which backbone `Button`, `TextField`, or `Choice` element reflects the interactive characteristics specified in the template `field` element's `ui` child element. The `ui` element has a child element that specifies the interactive characteristics of the field. The following table shows the correlation between the template `ui` element's child elements and the backbone `SubFields` element.

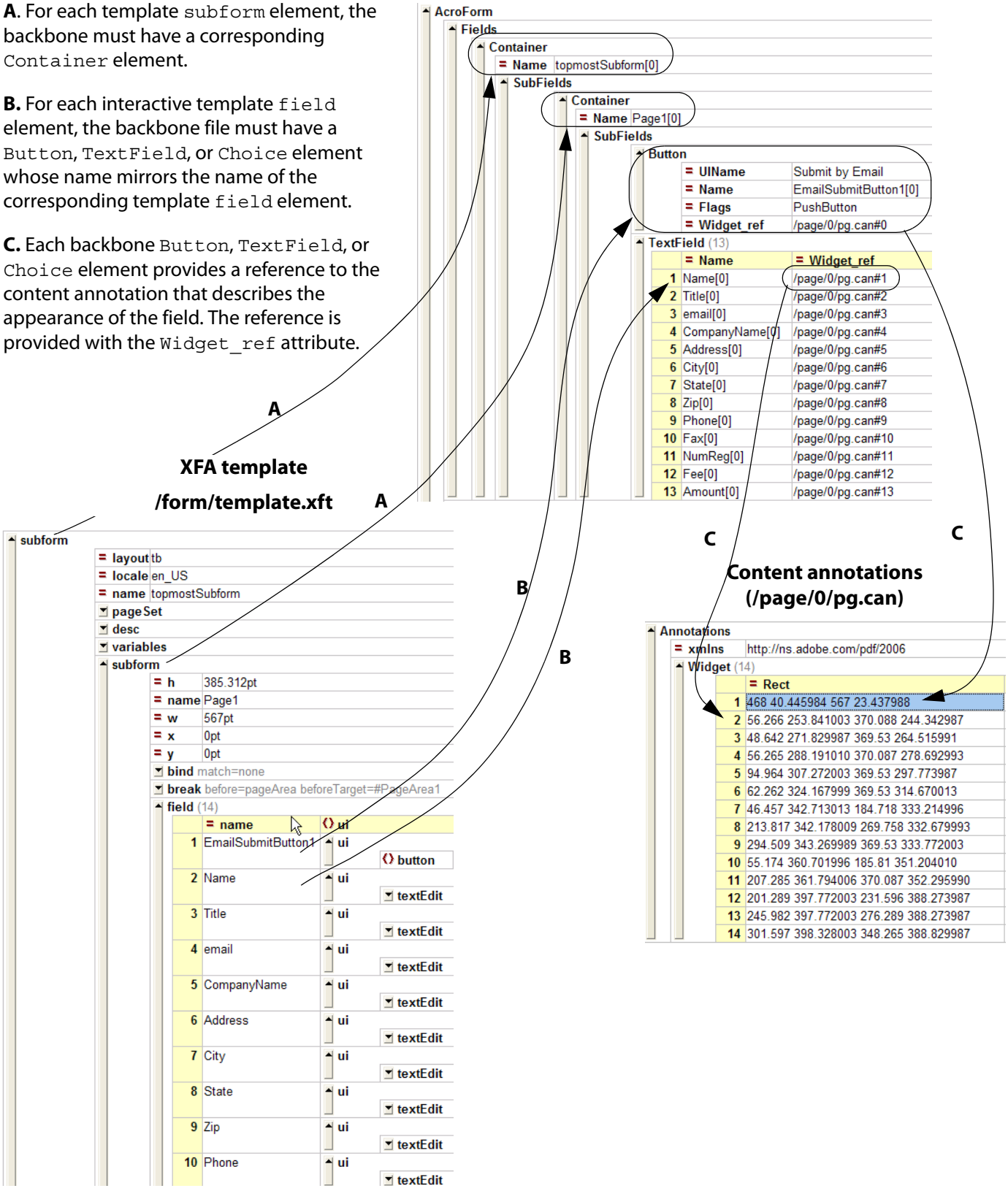
Template field/ui child element	Backbone Fields or SubFields element
`barcode`	`TextField`
`button`	`Button`, as described for Pushbuttons in the *PDF Reference*
`checkButton`	`Button`, as described for Radio Buttons or Check Buttons in the *PDF Reference*
`choiceList`	`Choice`
`dateTimeEdit`	`Textfield`
`defaultUi`	Depends on content type
`exObject`	`TextField`
`imageEdit`	`TextField`
`numbericEdit`	`TextField`
`passwordEdit`	`TextField`
`signature`	`Signature`
`textEdit`	`TextField`

Relationship between backbone, template and content annotations

A. For each template `subform` element, the backbone must have a corresponding `Container` element.

B. For each interactive template `field` element, the backbone file must have a `Button`, `TextField`, or `Choice` element whose name mirrors the name of the corresponding template `field` element.

C. Each backbone `Button`, `TextField`, or `Choice` element provides a reference to the content annotation that describes the appearance of the field. The reference is provided with the `Widget_ref` attribute.

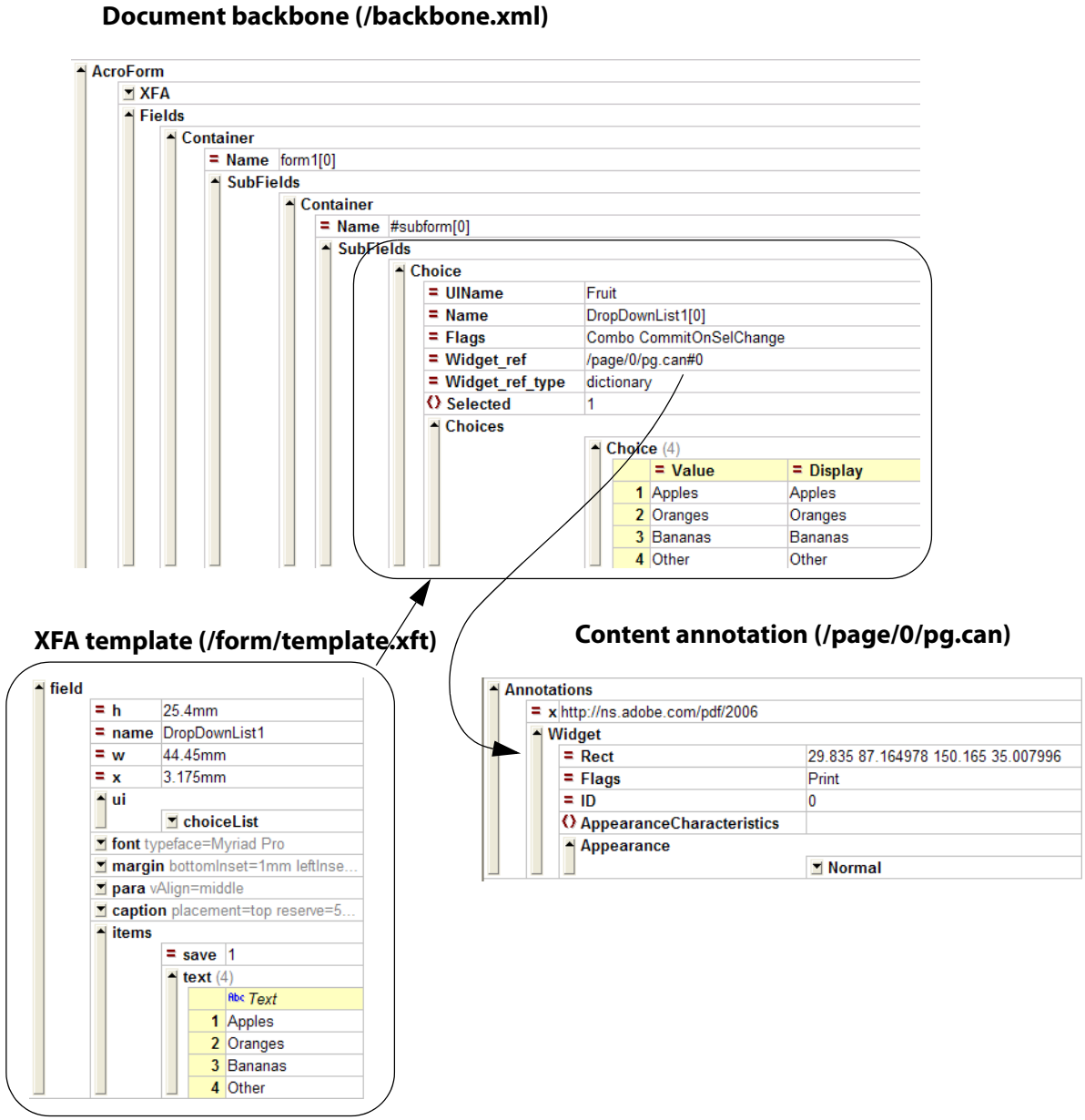


Choice fields in the backbone

In general, when a user enters data (or the data is provided by some other source), the backbone's linking attributes enable the user-supplied data to be represented in the XML form data set; however, when the user selects the data from a choice list, those choices must be duplicated in the backbone `Choice` element.

If the template `field` element contains an `items` element that provides a set of data choices and the values of those choices differ from the displayed items, the backbone `Choice` element must include a `MapTo` attribute. That is, the `MapTo` attribute handles the case where the value needed in the form data differs from the display in the UI. For example, the display might be a name (401K, IRA, None) and the corresponding data might be a value (0, 1, 2).

Backbone duplicates choice fields



Coordinates translation between template fields and content annotations

When a template field is represented as a widget annotation on a page, the coordinates specified for the template field must be translated into placement and size information for the widget annotation. The template field uses the XFA coordinate system, while the widget annotation uses the SVG coordinate system. While both of these systems place the origin at the upper left hand corner of the page, the XFA coordinate system is relative to the ancestor subforms, whereas the SVG coordinate system is relative to the entire page. As a result, if the top-level XFA subform places itself at the bottom of the page, its child subform's coordinate system uses the upper left hand corner of that subform as its origin (*Adobe XML Forms Architecture (XFA) Specification, v2.4*).

In the following example, the template field named `EmailSubmitButton1` specifies a placement and dimension that translates into a rectangle in the SVG coordinate system (Example 17.2). Notice that the template units may be specified while the SVG units must be points.

Example 17.1 ***Template subform and field size and placement (template.xft)***

```
<template xmlns="http://www.xfa.org/schema/xfa-template/2.5/"
baseProfile="interactiveForms">
   <subform layout="tb" locale="en_US" name="topmostSubform">
      <pageSet>
         <pageArea id="PageArea1" name="PageArea1">
            <contentArea h="385.312pt"
               name="ContentArea1" w="567pt" x="0pt" y="0pt"/>
         </pageArea>
      </pageSet>

      <subform h="385.312pt" name="Page1" w="567pt" x="0pt" y="0pt">
         <field name="EmailSubmitButton1"
            h="6mm" w="34.925mm" x="165.1mm" y="0">
         </field>
         …
      </subform>
   </subform>
</template>
```

Example 17.2 ***Annotations created for the XML form template (/page/0/pg.can)***

```
<Annotations xmlns="http://ns.adobe.com/pdf/2006">
   <Widget rect="402.3526, 618.4958, 531.0466, 635.4498" flags="print" ID="0">
      …
   </Widget>
</Annotations>
```

Display properties of annotations: basic properties, appearances and default appearances

In general, the display properties of individual form fields are provided by the widget annotations they reference. In addition, form fields can provide display properties of text entered by the user, called default appearances.

The display properties of annotations can be specified at various levels, each one offering increased flexibility. At its simplest level, the display properties can be specified as annotation attributes such as color and border. At the most detailed level, they can be specified using *Appearances*. Appearances are

sets of SVG graphic descriptions that are associated with annotation states. And in the case of annotations that present user-provided text, the display properties of text are specified using *default appearances*.

Basic display properties for annotations

Annotations can specify simple display properties such as border width, border style, and border fill color.

Appearances assigned to annotation states

In addition to setting simple display properties, annotations can specify one or more Appearances as an alternative to the simple border and color characteristics available by default. Multiple appearances enable the annotation to be presented visually in different ways to reflect its interactions with the user. Each Appearance specifies the color, fonts, and graphics to be rendered inside the annotation rectangle.

The graphic representation of annotations is usually explicit, and presented in the document as part of the annotation information. This representation, called an Appearance, specifies the display properties and SVG content for specific annotation states. See "Specifying different appearances for different annotation states" on page 147. The SVG content is specified as a reference to an SVG file located in the Mars document. The Appearance also specifies how the SVG content should be transformed and cropped when it is placed on the page.

These SVG fragments can either appear inline in the annotation element or in separate package-level files referenced from the `Appearance` element.

Specifying different appearances for different annotation states

Annotations can specify multiple appearances, where each appearance is used when the annotation is in a different state. An annotation can define appearances for the following states:

- Normal, which is used when the annotation is not interacting with the user. This appearance is also used for printing the annotation.
- Rollover, which is used when the user moves the cursor into the annotation's active area without pressing the mouse button.
- Down, which is used when the mouse button is pressed or held down within the annotation's active area.

Widget annotations that omit appearances

Widget annotations use appearances to provide the display properties of form fields. If any of the widget annotations in a Mars document omit appearances, the Mars viewer application provides an application-specific appearance. To warn the application that it must provide appearances, the document backbone file annotation `PDF/Acroform/@NeedAppearances` attribute is set to true.

Default appearances

Annotations that accept user-provided text can specify a `DefaultAppearance` attribute that defines the default characteristics used to render text such as the font, weight, and color. Widget annotations accept user-provided text on behalf of the form fields they represent.

Specifically default appearances can appear in the following locations:

- Document backbone file as the attribute specified by the XPath expression:

```
PDF/AcroForm/@DefaultAppearance
PDF/AcroForm/Fields/Button/@DefaultAppearance
PDF/AcroForm/Fields/TextField/@DefaultAppearance
PDF/AcroForm/Fields/Choice/@DefaultAppearance
PDF/AcroForm/Fields/Container/@DefaultAppearance
```

Note: The `PDF/AcroForm/Fields/Container` shows only the highest level appearance of the `Container` element. That element can itself include a hierarchy of form fields.

- Content annotations file (/page/*page_number*/pg.can) as the attribute specified by the XPath expression:

```
Annotations/Screen/AppearanceCharacteristics/@DefaultAppearance
Annotations/Widget/AppearanceCharacteristics/@DefaultAppearance
```

- Markup annotations file (/page/*page_number*/pg.svg.ann) as the attribute specified by the XPath expression:

```
Annotations/FreeText/@DefaultAppearance
Annotations/Redact/@DefaultAppearance
```

By themselves, default appearances are different from Appearances, but when default appearances are combined with text entered by the user, they create a new normal-state Appearance for the annotation. This new appearance is virtual. That is, it is never included in Mars documents.

The following example sets the default appearance for text within a text field to the Helvetica font with a gray fill and leaves the font size unchanged.

Example 17.3 ***Specifying a default appearance***

```
<TextField UIName="Total registration fees:  # of registrations"
   Name="Total registration fees:  # of registrations"
   Widget_ref="/page/0/pg.can#11" Widget_ref_type="dictionary">
   <DefaultAppearance Font="Helv" TextSize="0"
      FillColorSpace="Gray" FillColor="0"/>
   <Resources>
      <Fonts>
         <Font Tag="Helv" src="/font/f_240-1.fd"/>
      </Fonts>
      <Encoding type="dictionary">
         <PDFDocEncoding type="dictionary" ref="#0" ref_type="dictionary"/>
      </Encoding>
   </Resources>
</TextField>
```

For more information on content and markup annotations, see section 8.4 in the *PDF Reference, version 1.7*.

Creating XML forms

This section describes how to incorporate existing XML forms into a Mars document. See *Adobe XML Forms Architecture (XFA) Specification, v2.4* for information on creating XML forms. It is broken into the following basic steps,:

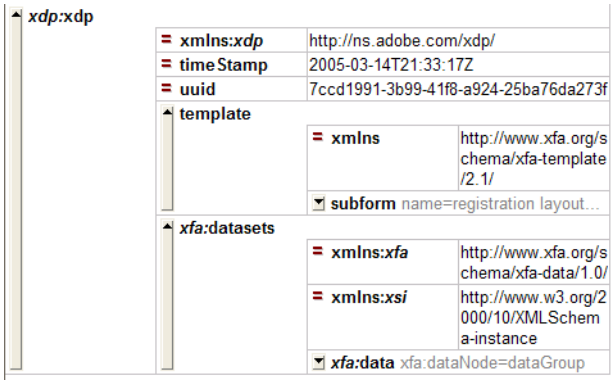
1. "Adding the form files to the /form directory" on page 149
2. "Referencing the form files" on page 150

3. "Specifying containers and fields that mirror the template" on page 151
4. "Specifying XML form field actions" on page 158
5. "Specifying XML form predefined appearances" on page 159
6. "Creating XML form data" on page 159

Adding the form files to the /form directory

This section assumes you have already created an XML form that is packaged as an XML Data Package (XDP). Such a package can be produced by Adobe LiveCycle Designer. The *Adobe XML Forms Architecture (XFA) Specification, v2.4* specifies the XDP grammar.

Each of the xdp child elements represents an XML form component (see illustration below), whose contents you will relocate to a form file in the /form directory.



To add the form files to the /form directory:

1. In the /form directory, create a file named preamble.
2. In the preamble file, add the following content. This content provides the root tag for the XDP file. It contains the following expression:

```
<?xml version="1.0" encoding="UTF-8"?>
  <xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/">
```

3. In the /form directory, create a file named template.xft.
4. Copy the entire `template` element from your XDP file into the template.xft file. The template.xft file specifies the layout and fields of an XML form and can specify interactive and server/client behavior. It also specifies (implicitly or explicitly) the binding between data provided in datasets.xml and its fields. It contains a root element such as the following example:

```
<template xmlns="http://www.xfa.org/schema/xfa-template/2.5/"
  baseProfile="interactiveForms">
  ...
</template>
```

5. In the /form directory, create a file named datasets.xml.
6. Copy the entire datasets element from your XDP file into the datasets.xml file.

7. If your XDP file includes a config element, create a file named config.xml and populate that file with the entire `config` element from the XDP file.
8. If your XDP file contains any other form parts, create a file for it and populate that file with the entire element contents. Although not required, the file names should reflect the top-level element name.
9. In the /form directory, create a file named postamble.
10. In the postamble file, add the following content. This content provides the ending tag for the XDP file. It contains the following expression:

    ```
    </xdp:xdp>
    ```

See also "Form parts" on page 142.

Referencing the form files

To modify the backbone file to reference the form files, make the following changes to the backbone file:

1. Create the `AcroForm` element as a child of the root `PDF` element.
2. Create an `XFA` element as a child of the `AcroForm` element. The presence of this element indicates an XML form is represented by the Mars document.
3. For each form file created in "Adding the form files to the /form directory" on page 149, add a `FormPart` element as a child of the `XFA` element (Example 17.4). Ensure the `FormPart` element that references the beginning tag for the `xdp:xdp` element is the first element in the `XFA` element and the `FormPart` element that references the ending tag is the last.
4. For each of the `FormPart` elements, specify the following attributes:
   - Set the `Name` attribute to a name to be associated with the form part. The string names can be arbitrary with the following exceptions:
     - XFA template must be identified by the string "`template`"
     - XFA datasets must be identified by the string "`datasets`"
     - XFA configuration must be identified by the string "`config`"
   - Set the `src` attribute to reference the form part.

In the following example, there are nine `FormPart` elements. Each element identifies the type of element and specifies its location.

Example 17.4 ***XML form parts (backbone.xml)***

```
<PDF PDFVersion="1.6" Version="0.8.0" Class="04-18-07"
   xmlns="http://ns.adobe.com/pdf/2006">
   <AcroForm >
      <XFA>
         <FormPart Name="xdp:xdp" src="/form/xfa_1914032634-1.xml"/>
         <FormPart Name="config" src="/form/config.xml"/>
         <FormPart Name="template" src="/form/template.xft"/>
         <FormPart Name="datasets" src="/form/datasets.xml"/>
         <FormPart Name="localeSet" src="/form/localeSet"/>
         <FormPart Name="PDFSecurity" src="/form/PDFSecurity"/>
         <FormPart Name="form" src="/form/form"/>
         <FormPart Name="&lt;/xdp:xdp&gt;" src="/form/xfa_1914032634-2.xml"/>
```

```
      </XFA>
      …
   </AcroForm>
</PDF>
```

Specifying containers and fields that mirror the template

Certain characteristics of the fields defined in the XML template file must be duplicated in the form fields specified in the backbone file. Those duplicated characteristics are the features that XML and Acrobat forms have in common. This duplication enables applications to use a common architecture in handling both types of form fields.

To modify the backbone file to represent the template containers and fields, perform the following steps. These steps continue from the previous section.

1. Add a `Fields` element as a child of the `XFA` element.
2. Create a `Container` element as a child of the `Fields` element, setting its `Name` attribute to `topmostSubform[0]`. This value is a SOM expression that identifies the top level `subform` element in the template. The *Adobe XML Forms Architecture (XFA) Specification, v2.4* describes SOM expressions.
3. Create a `SubFields` element as a child of the `Container` element created in Step 2.

Repeat the following steps for each level of template `subform` element that contains interactive fields (or that contains `subform` elements whose descendents contain interactive fields).

4. Create a `Container` element as a child of the `SubFields` element, setting its `Name` attribute to a SOM expression that identifies the corresponding template container element. The position of this element mirrors the hierarchy of the template subform, as described in "Relationship between the backbone, template and content annotations" on page 143.
5. Add a `SubFields` element as a child of the `Container` element. The `SubFields` element groups related fields.
6. If the corresponding template container element contains interactive fields, create a `Button`, `TextField` or `Choice` element in the `Container/SubFields` element at a level that corresponds to the template container, as described in "Creating buttons, text fields and choice fields" on page 152. See "Relationship between the backbone, template and content annotations" on page 143 for a mapping of template UI characteristics to widget types.

The following example from a template file represents an XML form that has two containers, the lower of which contains four interactive fields: one button and one text. The button interactive field has the appearance described by the widget annotation specified in its `Widget_ref` attribute. Any value exported by the field is associated with the template field specified by the SOM expression `EmailSubmitButton1[0]`. The text interactive field has the appearance described by the widget annotation specified in its `Widget_ref` attribute. Any value exported by the field is associated with the template field specified by the SOM expression `Name[0]`.

Example 17.5 ***AcroForm/Fields element reflects the XFA template structure***

```
<AcroForm NeedAppearances="true">
   <Fields>
      <Container Name="topmostSubform[0]">
         <SubFields>
            <Container Name="Page1[0]">
               <SubFields>
                  <Button UIName="Submit by Email" Name="EmailSubmitButton1[0]"
                      Flags="PushButton" Widget_ref="/page/0/pg.can#0"/>
                  <Text UIName="Name" Name="Name[0]"
                      Widget_ref="/page/0/pg.can#1"/>
                     …
               </SubFields>
            </Container>
         </SubFields>
      </Container>
   </Fields>
</AcroForm>
```

Creating buttons, text fields and choice fields

This section explains how to add buttons, text fields or choice fields to a `Fields` or `SubFields` element. These instructions apply to the creation of XML forms and Acrobat forms.

Button field

To add a button field to a `Fields` or `SubFields` element, perform the following steps:

1. Create a `Button` element as a child of the `Fields` or `SubFields` element
2. Add `Button` element attributes as follows:
   - `Name` attribute that identifies the button on the form. For XML forms, this name must be consistent with the name specified in the corresponding field in the XML form template file.
   - `UIName` attribute that specifies a string the document reader can display to the user using certain interfaces, such as the tooltip.
   - `Flags` attribute with the desired field characteristics, such as `RadiosInUnison`, `Radio`, and `Pushbutton`.
   - If the button behavior is defined by a single widget, set the `Button` element's `Widget_ref` attribute to the URI that references a widget that provides the appearance and behavior for the button. For example:

     ```
     <Button Widget_ref="/page/0/pg.can#0"/>
     ```

   - If the button appearance and behavior is defined by multiple widgets, add a `SubFields` element that contains one child `Container` element for each of the widgets. In each `Container` element, add a `Widget_ref` attribute that specifies the widget location. For example:

     ```
     <Button>
        <SubFields>
           <Container Widget_ref="/page/0/pg.can#0"/>
           <Container Widget_ref="/page/0/pg.can#1"/>
           <Container Widget_ref="/page/0/pg.can#2"/>
        </SubFields>
     ```

```
</Button>
```

- For XML forms, set the `MapTo` attribute value to a SOM expression that references the corresponding field in the template. For example:

```
<Choice MapTo="form1[0].grantApplication[0].page1[0].State[0]"/>
```

3. Create a `DefaultAppearance` element as a child of the `Button` element. Set the properties of this element to specify the default text size and color properties for the button.

4. For Acrobat forms, if the button should initiate an action, do the following:
   - Add the `OnEvent` element as a child of the `Button` element.
   - Add one of the additional actions elements as a child of the `OnEvent` element. Examples of additional actions elements are the elements `OnCursorEnter`, `OnMouseDown`, and `OnPageVisible`.
   - Add a a predefined action element as a child of the additional actions element, as described "Specifying actions" on page 76. Examples of predefined action elements are the elements `GoTo`, `Movie`, and `Hide`.

Note: XML form behavior is controlled by the information in the template file (/form/template.xft).

The following example creates a simple button field that has a push-button behavior but lacks any actions. The `Widget_ref` attribute is used only for XML forms.

Example 17.6 ***Specifying a button field for an XML form***

```
<Button Name="Button1"
   Flags="PushButton"
   Widget_ref="/page/0/pg.can#0">
   <DefaultAppearance … />
</Button>
```

Text field

To add a text field to a `Fields` or `SubFields` element, perform the following steps:

1. Create a `TextField` element as a child of the `Fields` or `SubFields` element

2. Add `TextField` element attributes as follows:
   - `Name` attribute that identifies the text field on the form. For XML forms, this name must be consistent with the name specified in the corresponding field in the XML form template file.
   - `UIName` attribute that specifies a string the document reader can display to the user using certain interfaces, such as the tooltip.
   - `Flags` attribute with the desired field characteristics, such as `Multiline`, `Password`, and `DoNotSpellCheck`.
   - If the text field behavior is defined by a single widget, set the `TextField` element's `Widget_ref` attribute to the URI that references a widget that provides the appearance and behavior for the button. For example:

```
<TextField Widget_ref="/page/0/pg.can#0"/>
```

   - If the text field appearance and behavior is defined by multiple widgets, add a `SubFields` element that contains one child `Container` element for each of the widgets. In each `Container` element, add a `Widget_ref` attribute that specifies the widget location. For example:

```
<TextField>
   <SubFields>
      <Container Widget_ref="/page/0/pg.can#0"/>
      <Container Widget_ref="/page/0/pg.can#1"/>
      <Container Widget_ref="/page/0/pg.can#2"/>
   </SubFields>
</TextField>
```

- For XML forms, set the `MapTo` attribute value to a SOM expression that references the corresponding field in the template. For example:

```
<Choice MapTo="form1[0].grantApplication[0].page1[0].State[0]"/>
```

3. Create a `DefaultAppearance` element as a child of the `TextField` element. Set the properties of this element to specify the default text size and color properties for the button.

4. For Acrobat forms, if the text field should initiate an action, do the following:
   - Add the `OnEvent` element as a child of the `TextField` element.
   - Add one of the additional actions elements as a child of the `OnEvent` element. Examples of additional actions elements are the elements `OnCursorEnter`, `OnMouseDown`, and `OnPageVisible`.
   - Add a a predefined action element as a child of the additional actions element, as described "Specifying actions" on page 76. Examples of predefined action elements are the elements `GoTo`, `Movie`, and `Hide`.

Note: XML form behavior is controlled by the information in the template file (/form/template.xft).

The following example creates a text field in the grant application whose text is `A. Title of Project`. The following text field is used for the "C8. Email Address" field in the Grant Application shown on page 158.

Example 17.7 ***A text field connecting the template field with the annotation (backbone.xml)***

```
<TextField
   MapTo="form1[0].grantApplication[0].page1[0].Email[0]"
   UIName="C8. Email Address" MaxLen="50" Name="Email[0]"
   Widget_ref="/page/0/pg.can#13">
   <DefaultAppearance Font="MyriadPro-Regular" TextSize="8"
      FillColorSpace="Gray" FillColor="0"/>
</TextField>
```

Choice field

To add a choice field to a `Fields` or `SubFields` element, perform the following steps:

1. Create a `Choice` element as a child of the `Fields` or `SubFields` element.

2. Add `Choice` element attributes as follows:
   - `Name` attribute that identifies the choice on the form. For XML forms, this name must be consistent with the name specified in the corresponding field in the XML form template file.
   - Set the `UIName` attribute to the string that the document reader may display for the user, for example, as a tooltip.
   - `Flags` attribute with the desired field characteristics, such as `FileSelect` and `MultiSelect`.

- If the choice appearance and behavior is defined by a single widget, set the `Choice` element's `Widget_ref` attribute to the URI that references a widget that provides the appearance and behavior for the choice. For example:

```
<Choice Widget_ref="/page/0/pg.can#0"/>
```

- If the choice appearance and behavior is defined by multiple widgets, add a `SubFields` element that contains one child `Container` element for each of the widgets. In each `Container` element, add a `Widget_ref` attribute that specifies the widget location. For example:

```
<Choice>
   <SubFields>
      <Container Widget_ref="/page/0/pg.can#0"/>
      <Container Widget_ref="/page/0/pg.can#1"/>
      <Container Widget_ref="/page/0/pg.can#2"/>
   </SubFields>
</Choice>
```

- For XML forms, set the `MapTo` attribute value to a SOM expression that references the corresponding field in the template. For example:

```
<Choice MapTo="form1[0].grantApplication[0].page1[0].State[0]"/>
```

3. Create a `DefaultAppearance` element as a child of the `Choice` element. Set the properties of this element to specify the default text size and color properties for the button.

4. If the choice represents a scrollable list box, optionally set the `TopIndex` attribute to specify the index of the first choice option visible in the list. If this property defaults to "0" (zero).

5. If the choice field allows multiple selections (specified in the `Flag` attribute having a `MultiSelect` entry), add a `Selected` attribute to the `Choice` element. Set the value of the element to an array of integers that specify the indices of the pre-selected or currently-selected choice options.

6. Create a `Choices` element as a child of the `Choice` element.

7. For each of the choice options selectable in the list, create a `Choice` element as a child of the `Choices` element. In each of the Choice elements, do the following:
   - Add a `Value` attribute that specifies the value associated with choice is selected
   - Add an optional `Display` attribute that specifies the choice presented to the user. If the `Display` attribute is not present, the `Value` attribute value is used in the display.

For more information, see "Choice fields in the backbone" on page 145.

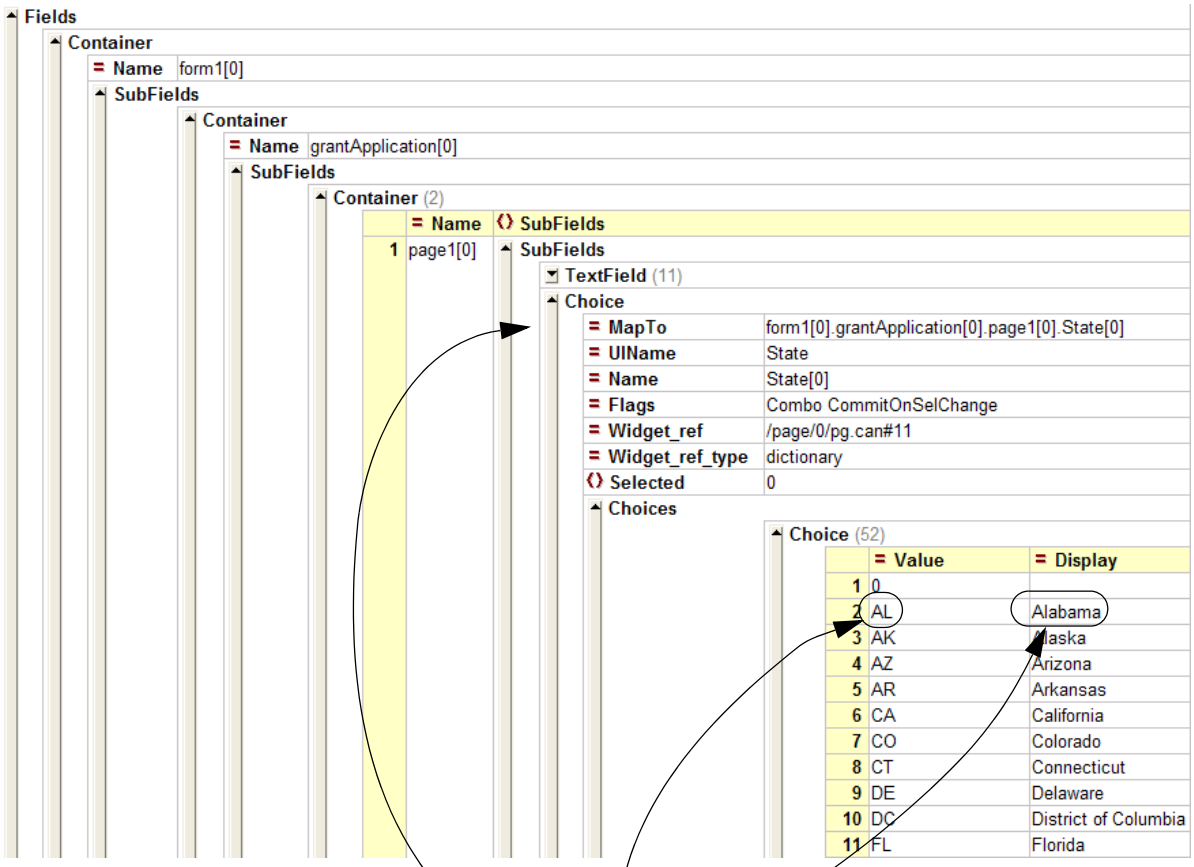
In this example, the export `Value` and `Display` values were specified. The following choice list is used for the "State" field in the Grant Application shown on page 161.

Example 17.8 ***Choice list with display values that differ from export values***

```
<Choice Flags="Combo CommitOnSelChange" … />
   <Choices>
      <Choice Value="0" Display="" />
      <Choice Value="AL" Display="Alabama" />
      <Choice Value="AK" Display="Alaska" />
      <Choice Value="AZ" Display="Arizona" />
      <Choice Value="AR" Display="Arkansas" />
   </Choices>
</Choice>
```

As shown in the following diagram, when the user selects one of the states in the choice list, the Mars application exports the value for the choice to the dataset associated with the template field.

Specifying a Choice field for selecting state abbreviations



The choice list exports values that differ from the display value, so the Choice must include a `MapTo` attribute that associates the field with the template field. This allows the exported value from the choice list to be exported to the form dataset.

The pull-down list allows the user to select from the full state names, and the exported values are the abbreviations for the states.

The following example combines the preceding examples to illustrate the order of form-related elements within the `backbone.xml` file for the Grant Application displayed on page 158.

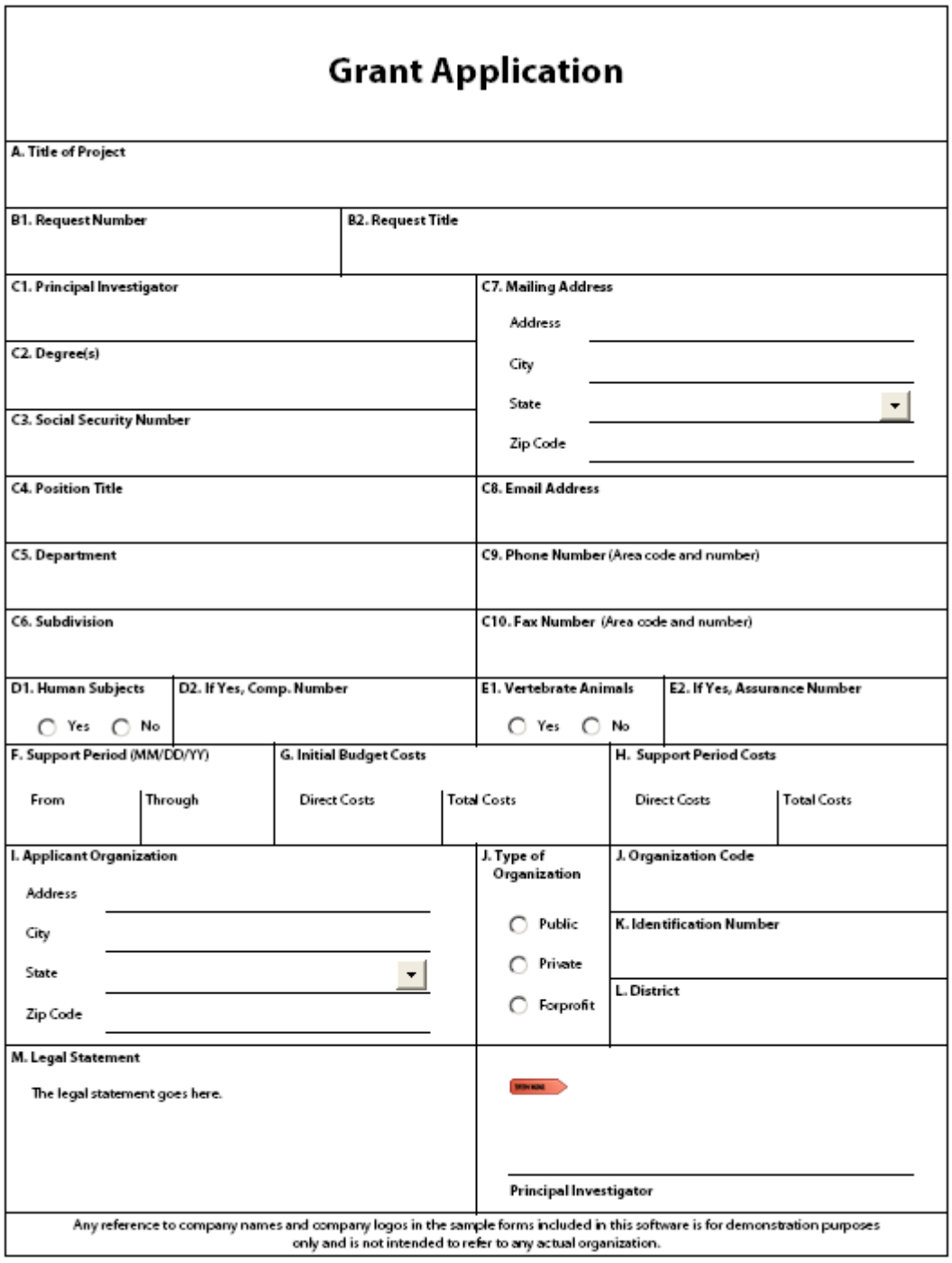
Example 17.9 *Specifying a text, button, and choice fields for an XML form*

```
<PDF><AcroForm>
  <XFA>
    <FormPart Name="xdp:xdp" src="/form/xfa_1797471653-1.xml"/>
    <FormPart Name="config" src="/form/config.xml"/>
    <FormPart Name="template" src="/form/template.xft"/>
    <FormPart Name="datasets" src="/form/datasets.xml"/>
    <FormPart Name="localeSet" src="/form/localeSet"/>
    <FormPart Name="xmpmeta" src="/form/xmpmeta"/>
    <FormPart Name="xfdf" src="/form/xfdf"/>
    <FormPart Name="PDFSecurity" src="/form/PDFSecurity"/>
    <FormPart Name="form" src="/form/form"/>
    <FormPart Name="&lt;/xdp:xdp&gt;" src="/form/xfa_1797471653-2.xml"/>
```

```
  </XFA>
  <Fields>
    <Container Name="form1[0]"> <SubFields>
      <Container Name="grantApplication[0]"> <SubFields>
        <Container Name="page1[0]"><SubFields>
          <TextField
            MapTo="form1[0].grantApplication[0].page1[0].ProjectTitle[0]"
            UIName="A. Title of Project" MaxLen="100"
            Name="ProjectTitle[0]"
            Widget_ref="/page/0/pg.can#0" Widget_ref_type="dictionary">
            <DefaultAppearance Font="Helv" TextSize="0"
              FillColorSpace="Gray" FillColor="0"/>
          </TextField>
            …
          <Button
            MapTo="form1[0].grantApplication[0].page1[0].HumanSubjects[0]"
            UIName="HumanSubjects"
            Name="HumanSubjects[0]"
            Flags="NoToggleOff Radio">
            <DefaultAppearance Font="Helv" TextSize="0"
              FillColorSpace="Gray" FillColor="0"/>
            <SubFields>
              <Button Widget_ref="/page/0/pg.can#16"
                Widget_ref_type="dictionary"/>
              <Button Widget_ref="/page/0/pg.can#17"/>
            </SubFields>
          </Button>
          …
          <Choice Selected="0"
            MapTo="form1[0].grantApplication[0].page1[0].OrgState[0]"
            UIName="State"
            Name="OrgState[0]"
            Flags="Combo CommitOnSelChange"
            Widget_ref="/page/0/pg.can#30">
            <DefaultAppearance Font="Helv" TextSize="0"
              FillColorSpace="Gray" FillColor="0"/>
            <Choices>
              <Choice Value="0" Display=" "/>
                <Choice Value="AL" Display="Alabama"/>
                <Choice Value="AK" Display="Alaska"/>
                <Choice Value="AZ" Display="Arizona"/>
                <Choice Value="AR" Display="Arkansas"/>
                  …
            </Choices>
          </Choice>
          …
        </SubFields> </Container>
      </SubFields> </Container>
    </SubFields> </Container>
  </Fields>
</AcroForm> </PDF>
```

XML forms are typically created in an external application, such as Adobe LiveCycle Designer. The following document was generated using LiveCycle Designer and represents the XML form for the grant application. All the XML form examples in this section correspond to the following figure.

Viewer application's display of an XML form



Grant Application

A. Title of Project

B1. Request Number | B2. Request Title

C1. Principal Investigator

C2. Degree(s)

C3. Social Security Number

C7. Mailing Address
Address
City
State
Zip Code

C4. Position Title | C8. Email Address

C5. Department | C9. Phone Number (Area code and number)

C6. Subdivision | C10. Fax Number (Area code and number)

D1. Human Subjects ◯ Yes ◯ No | D2. If Yes, Comp. Number | E1. Vertebrate Animals ◯ Yes ◯ No | E2. If Yes, Assurance Number

F. Support Period (MM/DD/YY) From | Through | G. Initial Budget Costs Direct Costs | Total Costs | H. Support Period Costs Direct Costs | Total Costs

I. Applicant Organization
Address
City
State
Zip Code

J. Type of Organization
◯ Public
◯ Private
◯ Forprofit

J. Organization Code

K. Identification Number

L. District

M. Legal Statement
The legal statement goes here.

Principal Investigator

Any reference to company names and company logos in the sample forms included in this software is for demonstration purposes only and is not intended to refer to any actual organization.

Page 1 of 2

Specifying XML form field actions

XML form field actions must not be defined. All form behavior is controlled by the XFA information.

Specifying XML form predefined appearances

Appearances are defined similarly for XML forms and Acrobat forms. For more information on how to specify predefined and default appearances for forms, see "Specifying Acrobat form predefined appearances" on page 165.

Creating XML form data

XML form data is saved in a file referenced from the `FormPart` element whose `Name` attribute is set to `datasets`. For example:

```
<FormPart Name="datasets" src="/form/datasets.xml"/>
```

By convention, the file is placed in the /form folder.

In the previous example, the elements and values entered by the user corresponding to the `Text` element with the attribute value `Name="A. Title of Project"` and the `Button` element with the attribute `UIName="HumanSubjects"` are saved in the datasets file (/form/datasets.xml).

XML forms provide a number of ways of mapping data in a particular XML schema to form fields. See the *Adobe XML Forms Architecture (XFA) Specification, v2.4* for more details. For this example, the user entered `Project XYZ` for the form field element `ProjectTitle`. The corresponding element name and value can be found in the XML segment below taken from `datasets.xml`.

```
<ProjectTitle>Project XYZ</ProjectTitle>
```

The user selected `No` for the button for the `HumanSubjects` element. This value of `No` corresponds to the `2` that is saved as the element's value in `datasets.xml`. The following XML segment shows the corresponding element name and value data taken from `datasets.xml`.

```
<HumanSubjects>2</HumanSubjects>
```

The data that is shown in the Grant Application form above is saved in the `datasets.xml` file located in the /form directory in this example. An excerpt of `datasets.xml` is shown in the following example.

Example 17.10*Saving data from an XML form*

```
<xfa:datasets xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
   <xfa:data>
   <form1>
      <grantApplication>
         <page1>
            <ProjectTitle>Project XYZ</ProjectTitle>
            <RequestNum>123456</RequestNum>
            <RequestTitle>title_request</RequestTitle>
            <PIName>John Smith</PIName>
            <Degrees>B.S. Computer Engineering</Degrees>
            <SSN>123456789</SSN>
            <PositionTitle>Senior Software Engineer</PositionTitle>
            <Department>Platform Engineering</Department>
            <Subdivision>Hardware West</Subdivision>
            <Email>johnsmith@example.com</Email>
            <PhoneNum>4088881234</PhoneNum>
            <FaxNum>4088885678</FaxNum>
            <HumanSubjects>2</HumanSubjects>
            <CompNum />
            <VertebrateAnimals>1</VertebrateAnimals>
```

```
            ...
            <CurrentPage>1</CurrentPage>
            <PageCount>2</PageCount>
         </page1>
         <page2>
            <Description>The project objective is to test a new hardware device
               with special software that checks vertebrate animal
               hearing.</Description>
            <CurrentPage>2</CurrentPage>
            <PageCount>2</PageCount>
         </page2>
      </grantApplication>
   </form1>
   </xfa:data>
</xfa:datasets>
```

Creating Acrobat forms

Acrobat forms are interactive forms used to gather information from users. The form field and behavior description in Mars appears directly in the Mars document backbone. The `AcroForm` and `Fields` elements are added to the `backbone.xml` file to specify form information for the Mars document. The interactive features that can be used on a form are buttons, text fields, and choice fields. Each of these interactive features are associated with one or more widget annotations that provide the appearance and behavior of the feature. For more information on Acrobat forms, see *PDF Reference, version 1.7*, Section 8.6.

The following list outlines the general steps required to create Acrobat form elements. These steps are explained in detail in subsequent sections.

- Planning
- Creating Acrobat field elements in the backbone file
- "Creating the widget annotations that provide the field behavior" on page 162
- Specifying Acrobat form field actions
- Specifying Acrobat form predefined appearances
- Creating Acrobat form data

Planning

Design the form by determining where in the document form fields should appear, and what data they connect to. Form fields that connect to the same data item should have the same name so when the user fills one in, they all get the same value.

Creating Acrobat field elements in the backbone file

To create an Acrobat form, perform the following steps:

1. Create the `AcroForm` element as a child of the root `PDF` element in `backbone.xml` to indicate that an Acrobat form is being created.

2. Create the `Fields` element as a child of the `AcroForm` element. This element will contain the Acrobat form's root (those with no ancestors in the field hierarchy).

3. If the `Fields` element contains top-level buttons, text fields or choice fields, follow the instructions in "Creating buttons, text fields and choice fields" on page 152. Such top-level features are independent of any hierarchical structure. Use the `Container` element to describe a hierarchy of form fields.

4. If the `Fields` (or `SubFields`) element should represent a hierarchy of subfields, do the following:
   - Create a `Container` element as a child of the `Fields` element.
   - In the `Container` element, add the `Name` attribute, setting its value to a name that will contribute to the path for the field. When users supply data to the field, the data is identified with a fully-qualified path that incorporates the field names.
   - Create a `SubFields` element as a child of the `Container` element.
   - If the `SubFields` element contains buttons, text fields or choice fields, follow the instructions "Creating buttons, text fields and choice fields" on page 152.
   - If the `SubFields` element should represent an additional layer in the hierarchy of subfields,, repeat step 4 (this step).

The diagram below shows the viewed form of an Acrobat form for the Conference Registration. Example 17.11 shows some of the elements from the backbone file that specify the form. All the Acrobat form examples in this section correspond to the figure below.

Conference Registration Acrobat form

The following example combines the preceding examples to illustrate the order of form-related elements within the `backbone.xml` file for the Conference Registration example. It shows how to create a push button, and various text fields.

Example 17.11 ***Specifying a button and text fields for an Acrobat form***

```
<PDF>
   <AcroForm>
      <Fields>
         <Button Name="Button1"
            Flags="PushButton" Widget_ref="/page/0/pg.can#0">
            <DefaultAppearance Font="Helv" TextSize="0"
               FillColorSpace="Gray" FillColor="0"/>
         </Button>
         <TextField UIName="Name" Name="Name" Widget_ref="/page/0/pg.can#1">
            <DefaultAppearance Font="Helv" TextSize="0"
               FillColorSpace="Gray" FillColor="0"/>
         </TextField>
         <TextField UIName="Title:"
            Name="Title:" Widget_ref="/page/0/pg.can#2">
            <DefaultAppearance Font="Helv" TextSize="0"
               FillColorSpace="Gray" FillColor="0"/>
         </TextField>
         …
      </Fields>
   </AcroForm>
</PDF>
```

Creating the widget annotations that provide the field behavior

To create the widget annotations that provide the field behavior, perform the following tasks:

1. Create a `Widget` element as a child of the `Annotations` element, in the content annotation file for the page (/page/*page_number*/pg.can). Set the element's attributes as follows:
   - The `Rect` attribute specifies the annotation rectangle, defining the location of the annotation on the page in SVG coordinates.
   - The `Flags` attribute specifies a space-separated list of values that specify the characteristics of the annotation.
   - The `ID` attribute provides a unique identifier that can be referenced by other elements. This attribute is equivalent to the XML ID attribute (`xml:id`).
2. Create an `OnActivation` element as a child of the `Widget` element.
3. Create an element as a child of the `OnActivation` element that specifies the desired action to take when the widget is selected. Valid child elements are `SubmitForm`, `ResetForm`, `ImportData`, and `JavaScript`. See "Specifying a submit-form action" on page 164.
4. Optionally, create an `AppearanceCharacteristics` element as a child of the `Widget` element. This element provides appearance characteristics for the widget. Specify the attributes as follows:
   - `BackgroundColor` attribute with three or more numbers that specify a color for the widget annotation's background. The number of array elements determines the color space. For example, three numbers specify the DeviceRGB color space.

- `BorderColor` attribute with three or more numbers that specify a color for the widget annotation's border. The number of array elements determines the color space.
- `Caption` attribute with a text string that specifies the widget annotation's normal caption, which is displayed when the widget is not interacting with the user.
- Specify other attributes as needed.

5. Create an `Appearance` element as a child of the `Widget` element.
6. Create a `Normal` element as a child of the `Appearance` element.
7. Create a `Graphic` element as a child of the `Normal` element. This element describes an Appearance (see "Display properties of annotations: basic properties, appearances and default appearances" on page 146). Specify its attributes as follows:
   - `src` attribute (required) that references SVG graphic content used in the appearance
   - `Matrix` attribute (optional) that describes transformations from the current user space to a new user space. The attribute contains six numbers that define a transformation matrix that maps the form coordinates into the page coordinates. Default value: the identity matrix [1 0 0 1 0 0].
   - `BBox` attribute that specifies a location on the page to use as the destination for the appearance. An array of four numbers in the form coordinate system, giving the coordinates of the left, bottom, right, and top edges, respectively, of the Appearance's bounding box. These boundaries are used to clip the Appearance and to determine its size for caching.
   - Other attributes and child elements as needed
8. If the widget includes a down or rollover appearance, create the corresponding `Down` or `Rollover` elements as children of the `Appearance` element.

Specifying Acrobat form field actions

A widget annotation can specify a form field action for the viewer application to perform, such as submitting a form, resetting a form, importing data into a form, or causing scripts to execute. In addition, annotations can specify the non-form actions described "Specifying actions" on page 76.

About submit-form actions

Submit-form actions transmit the names and values of selected interactive form fields to a server specified by a uniform resource locator (URL), presumably the address of a Web server that will process them and send back a response.

The set of fields whose names and values are to be submitted is defined by the `Fields` array in the action dictionary together with the `Include/Exclude` and `IncludeNoValueFields` flags in the `Flags` attribute. Each element of the `Fields` array identifies an interactive form field, either by an indirect reference to its field element (via a URI) or by its fully qualified field name. If the `Exclude` flag is not specified, the submission consists of all fields listed in the `Fields` array, along with any descendants of those fields in the field hierarchy. If the `Exclude` flag is set, the submission consists of all fields in the document's interactive form `Fields` element except those listed in the `Fields` element that is part of the submit forms action.

Fields that specify the `NoExport` flag are never included in the data sent by a submit-form action. Field names and values may be submitted in any of the following formats, depending on the settings of the action's `ExportFormat`, `SubmitPDF`, and `XFDF` flags:

- HTML form format (described in the HTML 4.01 specification)
- Forms Data Format (FDF), which is described in Section 8.6.6 of the *PDF Reference, version 1.7*
- XFDF, a version of FDF based on XML. XFDF is described in the Adobe technical note *XML Forms Data Format Specification, Version 2.0*. XML is described in the W3C document *Extensible Markup Language (XML) 1.0*)
- PDF (in this case, the entire document is submitted rather than individual fields and values).

The name submitted for each field is its fully qualified name that uses dot syntax. For example: `MyTopContainer.MySubContainer.MyTextfield`. The value is the current field value at the time of the submit action.

About other form field actions

In addition to the submit-form action, form fields and widget annotations can include the actions listed below:

- Reset-form actions reset selected interactive form fields to their default values.
- Import-data actions import Forms Data Format (FDF) or XML Forms Data Format (XFDF) data into the document's interactive form from a specified file that is not part of the document.
- JavaScript actions cause a script to be executed.

In addition to actions, an interactive form field or widget can specify an additional-actions element that extends the set of events that can trigger the execution of an action.

Specifying a submit-form action

To specify a submit-form action, make the following modifications to the content annotations file for the page (/page/*page_number*/pg.can):

1. In the `OnActivation` element of the widget annotation to be associated with the form submission,
2. Create the `SubmitForm` element and set the `Flags` attribute to the flags desired. For a list of flags, see Table 8.8.2 and section 8.6.4 in the *PDF Reference, version 1.7*.
3. Create the `File` element as a child of the `SubmitForm` element.
4. Set the `File` element's `FSType` attribute to a string indicating the file specification. In this example, the `FSType` was set to `URL`, which indicates a URL file specification.
5. Set the `File` element's `Name` attribute to specify the URL of the script at the web server that will process the submission. In this example, the URL invokes a `mailto` script and provides an email address to that script.

Note: The `mailto` script is handled locally on the client. The script sends email to the URL, rather than directly sending the form data to a server.

The following example creates a submit-form action that submits the document as PDF, using the MIME content type `application/pdf` to an email address specified by the `mailto` script.

Example 17.12*Specifying a submit-form action (/page/page_number/pg.can)*

```
<Annotations xmlns="http://ns.adobe.com/pdf/2006">
  <Widget Rect="431 142 560 125" Flags="Print" ID="0">
```

```
      <OnActivation>
         <SubmitForm Flags="SubmitPDF">
            <File Name="mailto:aname@adobe.com" FSType="URL"/>
         </SubmitForm>
      </OnActivation>
      <AppearanceCharacteristics BackgroundColor="0.7529"
         Caption="Submit by email"/>
      <Appearance>
         …
      </Appearance>
   </Widget>
   …
</Annotations>
```

Specifying Acrobat form predefined appearances

To create predefined appearances for a form, perform the following tasks:

1. Create a `Normal`, `Rollover`, or `Down` subelement within an `Appearance` element to indicate the type of appearance being created.
2. Create the `Graphic` element and set the `src` attribute value to the URI for the file that contains appearance related commands for the element.
3. Set the `Matrix` attribute to specify the mapping from form space units to user space units.
4. Create the font and image resources required for the appearance of the element. For more information on fonts, see "Referencing and Embedding Fonts" on page 35. For more information on images, see "Adding Images to SVG Content" on page 46.
5. Create the `BBox` attribute, setting its value to four numbers in the form coordinate system. The (`x1`,`y1`) coordinates represent the upper-left corner of the rectangular region for the `BBox` attribute and the (`x2`, `y2`) coordinates represent the lower-right corner. These boundaries are used to clip the appearance.

The example below specifies a normal appearance for the widget annotation. Since no special behavior is required when the mouse pointer is rolled over or pressed down on the widget annotation, those appearances were not specified.

Example 17.13*Specifying a submit-form action (/page/page_number/pg.can)*

```
<Annotations xmlns="http://ns.adobe.com/pdf/2006">
   <Widget Rect="431 142 560 125" Flags="Print" ID="0">
      <OnActivation/>
      <AppearanceCharacteristics BackgroundColor="0.7529"
         Caption="Submit by email"/>
      <Appearance>
         <Normal>
            <Graphic src="/page/0/form_613-2.svg"
               Matrix="1 0 0 1 0 0" BBox="0 0 128 16">
               <Resources/>
            </Graphic>
         </Normal>
      </Appearance>
   </Widget>
```

```
   ...
</Annotations>
```

The following is the SVG content that is used in the normal appearance state in the previous example. It creates the graphic shown at right.

Example 17.14*SVG content that provides the content for the appearance (/page/0/form_613-2.svg)*

```
<svg fill="none" stroke="none" width="128" height="16">
   <defs>
      <clipPath id="cl-1">
         <path d="M1,1h126.6939v14.9537H1z"/>
      </clipPath>
      <font-face font-family="F1">
         <font-face-src>
            <pdf:font-information xlink:href="/font/0493BD58.fd"/>
            <font-face-name name="Arial"/>
         </font-face-src>
      </font-face>
   </defs>
   <g transform="matrix(1 0 0 -1 0 16.9537)">
      <path fill=
         "rgb(191.9971,191.9971,191.9971) device-color(DeviceGray,0.7529)"
         d="M0,0h128.6939v16.9537H0z"/>
      <g clip-path="url(#cl-1)">
         <text transform="matrix(1 0 0 -1 0 3.821)"
            font-size="12" font-family="F1"
            fill="rgb(0,0,0) device-color(DeviceGray,0)" fill-rule="evenodd">
            <tspan x="18.387 26.3909 33.7228 41.0547 51.7226 55.0586">
               Submit
            </tspan>
         </text>
         <text transform="matrix(1 0 0 -1 0 3.821)" font-size="12"
            font-family="F1" fill="rgb(0,0,0) device-color(DeviceGray,0)"
            fill-rule="evenodd">
            <tspan x="62.331 69.6629">by</tspan>
         </text>
         <text transform="matrix(1 0 0 -1 0 3.821)" font-size="12"
            font-family="F1" fill="rgb(0,0,0) device-color(DeviceGray,0)"
            fill-rule="evenodd">
            <tspan x="79.647 86.319 96.9869 103.6589 106.9949">email</tspan>
         </text>
      </g>
   </g>
</svg>
```

Creating Acrobat form data

In a Mars document, the form field values are represented in a separate data file, conventionally called `form_data.xfdf` file in the /form directory. If there is no initial form data, this file is not required.

To create Acrobat form data, perform the following tasks.

1. Create the `form_data.xfdf` file in the /form directory that contains the data for the form.

2. Add elements that represent the contents, as described in the XFDF Specification, located at www.adobe.com/go/acrobat_developer. The data is separated into name/value pairs, which enables the extraction of data and creation of reports.

If the user entered the value `John Smith` for the `Name` element, it is saved in `form_data.xfdf`. The corresponding XML taken from `form_data.xfdf` is shown below.

```
<field name="Name">
   <value>John Smith</value>
</field>
```

Note: Mars form data is saved in XFDF format and is in XML; it does not require any special processing.

The example below shows an excerpt from `form_data.xfdf` from the Conference Registration example. It shows the format the data is stored in for Acrobat forms.

Example 17.15*Saving data in XFDF format*

```
<?xml version="1.0" encoding="UTF-8"?>
<xfdf xmlns="http://ns.adobe.com/xfdf/" xml:space="preserve">
<fields>
   <field name="Name">
      <value>John Smith</value>
   </field>
   <field name="= $">
      <value> 2385</value>
   </field>
   <field name="Address">
      <value> 789 Maple Street</value>
   </field>
   <field name="City">
      <value> San Jose</value>
   </field>
   <field name="Company Name">
      <value>ABC Incorporated</value>
   </field>
   …
</fields>
…
</xfdf>
```

18 Adding Foreign Data (Page Piece Dictionaries)

This chapter describes how foreign data can be added to a Mars documents.

This chapter contains the following information.

Topic	Description	See
Understanding foreign data	Explains where foreign data can appear in a Mars document and the structures and conventions for representing that data.	page 168
Adding private data to application datasets	Explains how to add private data to an application dataset	page 171
Adding private data to logical-structure	Explains how to add private data to the nodes in a page-level logical structure or to a class map	page 173

Understanding foreign data

Mars provides a mechanism that allows applications to store foreign data in a Mars document. Such data is meaningful to the application that produces it but is typically ignored by general-purpose Mars viewer applications. Foreign data usually includes identifiers that associate it with specific applications.

Foreign data includes data other than the named elements and attributes described by Mars. Depending on how the foreign data is used in the Mars document, it can be formatted as XML, binary, or some other format.

Foreign data falls in the following categories:

- Data expressed using the conventions defined in the Mars `default_dict_element` pattern. Such data is known as private data because it does not conform to a public specification. Private data provides a way to specify the type and other identifiers for each data item. For example, the private portion of page parts (/page/*page_number*/info.xml) contains foreign data. Here is the XPath expression for such an element:

  ```
  Page/ApplicationDatasets/ApplicationData/Private
  ```

  The private portion of markup annotations (/page/*page_number*/pg.svg.ann) contains private data:

  ```
  Annotations/Redact/OverlayAppearance/ApplicationDatasets/
    ApplicationData/Private
  ```

- Data expressed using conventions defined in other specifications. The 3D annotation (/page/*page_number*/pg.can) can include a reference to a file within the Mars package that contains custom data. Currently, the only external data that can be specified must conform to the specification *Universal 3D File Format (U3D), editions 1 and 3*. Such data is referenced from the 3D annotation, as shown in the following XPath expression:

  ```
  Annotations/A3D/ExData/
  ```

- Data associated with an XML form, as described "Creating XML form data" on page 159.

The remainder of this chapter addresses only private data.

About private data specifically allowed by the Mars schema

This section describes examples of more common uses of private data, where the private is stored within the Mars document where private data, and how the data must be formatted to satisfy Mars conventions.

Application datasets

The `ApplicationDatasets` element is used to represent application specific information. This application specific extension data can be associated with the document, individual pages, and fragments of pages such as form `XObjects`. The following list shows the various places the information can be stored within the document:

- Backbone file as a child of the `PDF` element (/backbone.xml)
- Page information file as a child of the `Page` element (/page/page_number/info.xml)
- Resource file as part of a Form XObject (/res/name.xml or /page/*page_number*/name.xml)
- Various other files and locations where a page fragment can be specified

Marked content properties

The marked content and logical structure features can include private data. For marked content, private data can be provided in the SVG grouping elements as the value of the `pdf:Props` attribute. In this context, the private data is called *user properties*. For logical structure, private data can be provided in the `Attribute` or `Attributes` element, which can appear in the page-level structure file (/page/*page_number*/struct.xml). These attributes are known as custom added extensions.

Logical-structure attribute objects

Applications or plug-in extensions that process logical structure can attach private data to the nodes in a structure tree. This private data is added to the attribute objects associated with the structure elements that define the nodes.

The attribute information is held in one or more attribute objects associated with the structure element. Each attribute may contain values for various attributes. If the attribute object contains private data, the `Owner` attribute must be set to `UserProperties`. This setting distinguishes the attribute object from the set of standard attribute owners defined in Table 10.28 in the *PDF Reference, version 1.7*.

Other

Mars also enables the representation of new types of actions and annotations.

Format for private data

Mars specifies a syntax you can use to specify private data. The *Mars Reference* defines this syntax in the pattern called `default_dict_element`. This format is used in application datasets and in logical structure. It can represent a complex hierarchy of dictionaries, arrays, streams and simple types.

The syntax used to express array entries differs slightly from the syntax used otherwise.

Private data other than array entries

This table describes the format for entries in private data other than array entries. The *KEY* is a placeholder for the element name as follows:

- If the element is a top level entry, the context establishes the value to use for *KEY*. For example, if the private data appears in an application dataset, the element name is `Private`; if it appears in a structure-file attribute object, the element name is `UserProperty`.
- If the element is a member of a dictionary, replace *KEY* with a name associated with the entry. Any illegal XML characters in the name must be escaped. The name is application-dependent.

Format for private data other than array entries

Data type	Format
null	<*KEY* type="null"/>
integer	<*KEY* type="integer" Value="123"/>
number	<*KEY* type="number" Value="123.45"/>
string	<*KEY* type="string" Value="Here is text"/> or <*KEY* type="string" Value="Here is text"/ Value_enc="UTF-16"/>
boolean	<*KEY* type="boolean" Value="true"/> or <*KEY* type="boolean" Value="false"/>
name	<*KEY* type="name" Value="a_name"/> or <*KEY* type="name" Value="a_name" Value_enc="BASE64"/>
dictionary	<*KEY* type="dictionary"> Nested dictionary values </*KEY*>
array	<*KEY* type="array"> Nested array elements </*KEY*>
stream	<*KEY* <File/> <FileFilters/> <FDecodeParms/> </*KEY*>

Entries in private data arrays

The following table describes the format used to express private data that appears as entries in an array.

Format for private data expressed as entries in an array

Data type	Format
null	`<Null/>`
integer	`<Int Value="123"/>`
number	`<Number Value="123.45"/>`

string	`<String Value="Here is text"/>`
	or
	`<String Value="Here is text"/`
	`  Value_enc="SHIFT-JIS"/>`

boolean	`<Bool Value="true"/>`
	or
	`<Bool Value="false"/>`

name	`<Name Value="A Name"/>`
	or
	`<Name Value="encoded name" Value_enc="BASE64"/>`

dictionary	`<Dict>`
	`  Nested dictionary content`
	`</Dict>`

array	`<Array>`
	`  Nested array content`
	`</Array>`

stream	`<Stream`
	`  <File/>`
	`  <FileFilters/>`
	`  <FDecodeParms/>`
	`</Stream>`

Adding private data to application datasets

An `ApplicationDatasets` element can be used to hold private application data. The `ApplicationDatasets` element can appear as a child of the `Page` element in the page information file (/page/*page_number*/info.xml) or as a child of the document catalog (/backbone.xml) or as a child of various annotation and appearances elements.

Registering with Adobe

Before producing Mars documents that include private data, you should register with Adobe, as described in Appendix E of the *PDF Reference, version 1.7*.

Adding an application dataset

To add an application dataset, amend the file in which the application dataset is being added as described in the following tasks. "About private data specifically allowed by the Mars schema" on page 169 describes the parent elements that can house an application dataset and the file in which those elements exist.

1. Create an `ApplicationDatasets` element in the parent element.
2. For each unique set of private data, add an `ApplicationDataset` element as a child of the `ApplicationDatasets` element.
3. In the `ApplicationDataset` element, add the `Owner` attribute, with a string that uniquely identifies the set of private data. To avoid collisions with other private data, register this name with Adobe, as described "Registering with Adobe" on page 172.
4. In the `ApplicationDataset` element, optionally add the `LastModified` attribute, with a string that identifies the date and time when the contents of the data was most recently modified.
5. Add a `Private` element as a child of the `ApplicationDataset` element.
6. Add a `type` attribute to the `Private` element that specifies one of the types listed in the table on page 170.
7. Augment the `Private` element with other attributes and elements as follows:
   - If the property type is integer, number, string, boolean or name, add a `Value` attribute whose value specifies the data.
   - If the property type is array, create one child element for each of the entries in the array, naming the element with one of the type names described in the table on page 171. These element names include `Int`, `Number`, `String`, `Name`, `Dict`, `Array`, and `Stream`. For each element created in the array, repeat the instructions in step 7, except augment the elements created here rather than the `Private` element. The entries can be a mix of different types.
   - If the property type is dictionary, create one child element for each of the entries in the dictionary, naming each element with the key-name and specifying the `type` attribute. For each element created in the array, repeat the instructions in step 7, except augment the elements created here rather than the `Private` element. Ensure that the key-names used to name the elements for each dictionary entry are legal in XML, as described in the *XML Specification*.

The following example shows what the XML looks like where the data is defined by another application. The example shows how to specify the following types: `integer`, `dictionary`, `stream`.

- The `Owner` for the first set of application data is `"Jones Formatter 1.2"` and the data was last modified on June 26, 2003 at 22:19:26. The private data includes a single entry of type name with a value of `"NoRecallSpellingHollars"`.
- The `Owner` for the second set of application data is `"Flex 1.1"` and the data was last modified on February 26, 2003 at 22:19:26. The private data includes a single dictionary that contains the entries `Server` and `Age`. `Server` is of type string and has the value `"Acroblab1"`; `Age` is of type integer and has the value `"23"`.

Example 18.1 ***Specifying private application data***

```
<ApplicationDatasets>
   <ApplicationData Owner="Jones Formatter 1.2" LastModified="20030626221926">
      <Private type="name" Value="NoRecallSpellingHollars"/>
   </ApplicationData>
   <ApplicationData Owner="Flex 1.1" LastModified="20030226221926">
      <Private type="dict">
         <Server type="string" Value="Acroblab1"/>
         <Age type="integer" Value="23"/>
      </Private>
   </ApplicationData>
<ApplicationDatasets>
```

Adding private data to logical-structure

As discussed in "Specifying the structuring state and class mapping" on page 133, private data can be added to the attribute objects specified in logical structure nodes in the page-level structure file (/page/page_number/struct.xml) or in the class maps in the backbone file (/backbone.xml). These user properties are provided in the attribute object's `UserProperties` element. Attribute objects can appear in the backbone file's `PDF/ClassMap` element or in the page-level structure file's `Structure/Elem/Attribute` or `Structure/Elem/Attributes/Attribute` element. "Creating a page-level structure file" on page 135

Indicating the logical structure contains private data

To indicate the Mars document logical structure that includes private data, change the backbone file as follows:

1. Add the `Marked` element as a child of the `PDF` element. (This is required for any tagged PDF document regardless of whether it has private data.)
2. Add the `UserProperties` attribute to the `Marked` element, setting its value to `true`. For example: `UserProperties="true"`.

Adding private data to the structure elements

To specify user properties in an attribute object, make the following additions to the backbone.xml or struct.xml file. These instructions assume you have already created attribute objects that include attributes and elements specified by the *Mars Reference*.

1. In the `Attribute` element, add an `Owner` attribute whose value is `UserProperties`. For example: `Owner="UserProperties"`
2. Add a `UserProperties` element as a child of the `Attribute` element.
3. For each property, create a `UserProperty` element as a child of the `UserProperties` element.
4. Add a `type` attribute to the `UserProperty` element that specifies one of the types listed in the table on page 170.

5. Augment the `UserProperty` element with other attributes and elements as follows:
   - If the property type is integer, number, string, boolean or name, add a `Value` attribute whose value specifies the data.
   - If the property type is array, create one child element for each of the entries in the array, naming the element with one of the type names described in the table on page 171. These element names include `Int`, `Number`, `String`, `Name`, `Dict`, `Array`, and `Stream`. For each element created in the array, repeat the instructions in step 5, except augment the elements created here rather than the `Private` element. Entries can be of mixed types.
   - If the property type is dictionary, create one child element for each of the entries in the dictionary, naming each element with the key-name and specifying the `type` attribute. For each element created in the array, repeat the instructions in step 7, except augment the elements created here rather than the `Private` element. Ensure that the key-names used to name the elements for each dictionary entry are XML-compatible.
   - If the property value should be formatted, add a `FormattedValue` attribute whose value specifies the formatted presentation.

The following example defines two classes with the names `"Vertical"` and `"Horizontal"`. Both of these classes must define their `Attribute Owner` as `"UserProperties"`.

Example 18.2 ***Specifying UserProperties***

```
<ClassMap>
   <Class Name="Vertical">
      <Attribute Owner="UserProperties">
         <UserProperties>
            <UserProperty FormattedValue="90 deg." Name="Angle" Value="90"
               type="number"/>
            <UserProperty Hidden="true" Name="Extension Lines" Value="Both"
               type="string"/>
            <UserProperty Name="Precision" Value="0.00" type="string"/>
            <UserProperty Name="Units" Value=".Feet-Inch
               (1'-0&amp;quot;)"type="string"/>
            <UserProperty Name="Units Display" Value="Show Units"
               type="string"/>
         </UserProperties>
      </Attribute>
   </Class>
   <Class Name="Horizontal">
      <Attribute Owner="UserProperties">
         <UserProperties>
            <UserProperty FormattedValue=" deg." Name="Angle" Value="0"
               type="number"/>
            <UserProperty Hidden="true" Name="Extension Lines" Value="Both"
               type="string"/>
            <UserProperty Name="Precision" Value="0.00" type="string"/>
            <UserProperty Name="Units" Value=".Feet-Inch
               (1'-0&amp;quot;)"type="string"/>
            <UserProperty Name="Units Display" Value="Show Units"
               type="string"/>
         </UserProperties>
      </Attribute>
   </Class>
</ClassMap>
```

A Tasks Enabled by Mars

This appendix summarizes some of the tasks enabled by Mars. All of these tasks require some level of programmatic or XSL manipulation.

Document creation

- Assembling a basic document from a list of SVG pages, bookmarks, and metadata
- Using SVG to create simple text and images
- Adding a first page and a standard cover page
- Adding a standard header or footer to each page
- Inserting pages
- Adding text to each SVG page
- Creating pages, bookmarks, annotations, navigation elements, structure, and so on. You can optionally specify layers or more detailed specifications of the font properties.
- Using Adobe PDF Library to manipulate a PDF file and then saving the result to a Mars document

Form creation

- Creating an Acrobat form
- Creating an XML form that uses the Adobe XML Form Architecture (XFA)
- Adding an existing XML form to a Mars document

Content manipulation

- Adding form data
- Adding annotations
- Inserting pages
- Adding bookmarks
- Inserting navigation elements onto pages with scripts
- Adding headings and footings to pages
- Adding structure with content to a document
- Replacing images

Extracting and reporting

- General document information
- Listing fonts used
- Listing color spaces used
- Listing pages and page sizes, maximum page size
- Listing images with sizes, types, metadata

- Extracting text from tagged PDF
- Extracting pages
- Extracting form data
- Extracting annotations
- Extracting metadata

Index

Symbols

?xpacket (XMP metadata) 118

A

A3D (content annotation element) 82
Acrobat forms
    about 140
    actions 163
    creating 160–167
    creating button fields 152
    creating choice fields 154
    creating text fields 153
    data associated with 141
    form data 166
    form field widgets 140
    form fields 140
    predefined appearances 165
    submit-form actions 163, 164
AcroForm (backbone element) 142, 150
Action (common Mars element) 64, 69
actions
    GoTo 76
    JavaScript 78
    representing 74–81
    role in XFA forms 74
    Sound 77
    URI 77
Adobe Font Development Kit for OpenType 45
Adobe Illustrator 35
Annotations (backbone element) 87, 90
annotations (content and markup)
    appearances and appearance states 85, 147
    basic display properties 85, 147
    default appearances 86, 147
    omitted appearances 85, 147
Appearance (backbone element) 89
Appearance (common Mars element) 163
AppearanceCharacteristics (backbone element) 89
AppearanceCharacteristics (common Mars element) 162
ApplicationDatasets (common Mars element) 169, 172
articles
    creating 100–102
ArticleThread (articles element) 101
Ascent (font descriptor attribute) 41
Attribute (page structure element) 173
AvgWidth (font descriptor attribute) 41

B

backbone file
    about 15
    specification of page placement 28
BaseFont (font descriptor attribute) 40
BaseState (backbone element) 105
Bead (articles element) 101
BeforeClose (backbone element) 79
Bookmark (bookmark element) 64
bookmarks
    creating 62–73
    explicit destinations for 66
    named destination caches 71
    named destinations for 67
Bookmarks (bookmark element) 63
Border (markup annotation element) 94
Button (backbone element) 142, 151, 152

C

Cache (structure cache element) 137
cache files
    named destinations 72
CapHeight (font descriptor attribute) 41
Caret (markup annotation element) 83
CFFFont (OpenType CFF property) 41
CharSet (font descriptor attribute) 42
Choice (backbone element) 142, 151, 154
Choices (backbone element) 155
Circle (markup annotation element) 83
Class (backbone element) 134
class maps (logical structure) 132
ClassMap (backbone element) 133
Collection (backbone element) 113
Collection (embedded files element) 112
collections, creating 111–115
color profiles
    defining in SVG 58
    purpose of 47
    specifying for images 47
color spaces
    associating with text or graphics 59
    associating with text or graphics (ICC) 58
    specifying 50–60
Compact Embedded Font (CEF) format 35
Configs (backbone element) 106
container (Adobe Container Format) 17
Container (backbone element) 151
content annotations
    A3D 82
    creating 87–89
    Link 82
    Movie 82
    popup 83
    PrinterMark 82
    Screen 82
    TrapNet 82
    types of 82
    Watermark 82
    Widget 82
ContentGroup (backbone element) 109

Contents (markup annotation element) 91
Contents (page structure element) 30
coordinate system. *See* SVG coordinate system
CS (markup annotation element) 94

D

Data (named destination cache) 72
Default (backbone element) 105
default_dict_element (common Mars element) 169
Desc (backbone element) 96
Descent (font descriptor attribute) 41
rdf:Description (XMP metadata) 119
Dest (common Mars element) 76
Dest (named destination cache) 73
document structure 15
DocumentID (backbone element) 43
Down (common Mars element) 163, 165

E

Elem (page structure element) 135
embedded fonts
    obfuscation of 37
    optimizing appearances of 38
EmbeddedFile (backbone file) 96
EmbeddedFile (embedded files element) 112
EmbeddedFiles (backbone element) 96
ExclusiveGroups (backbone element) 105

F

Field (backbone element) 113
Fields (backbone element) 142, 151
File (common Mars element) 164
FileAttachment (markup annotation element) 83, 97
FileData (backbone element) 96
Files (structure cache element) 72, 137
FirstChar (font descriptor attribute) 36, 40
Fit (common Mars element) 66
FitB (common Mars element) 66
FitBH (common Mars element) 67
FitBV (common Mars element) 67
FitH (common Mars element) 66
FitR (common Mars element) 66
FitV (common Mars element) 66
Flags (font descriptor attribute) 41
Font (font descriptor element) 36, 40
font characteristics
    applying to text 39
font descriptors 37
    optimizing appearances of 38
    specifying 40
font permissions, checking 42
FontBBox (font descriptor attribute) 41
FontDescriptor (font descriptor element) 36, 41
font-face (SVG element) 36, 38
font-face-name (SVG element) 38
font-face-src (SVG element) 36
font-face-uri (SVG element) 36, 38
FontFamily (font descriptor attribute) 41
font-family (SVG attribute) 36, 38, 39
pdf:font-information (SVG element) 38
FontName (font descriptor attribute) 41
font-related structures 36
fonts
    font face specifiers 37, 38
    ligature formation 40
    obfuscation of embedded 36, 43
    OpenType 35
    referencing and embedding 37–45
    required OpenType tables 43
    supported 35
    SVG 35
FontStretch (font descriptor attribute) 41
FontWeight (font descriptor attribute) 41
foreign data
    about 168
FormPart (backbone element) 142, 150
forms. *See* Acrobat form or XML form
FreeText (markup annotation element) 83

G

GoTo (action element) 75, 76, 153
GoTo3DView (action element) 75
GoToE (action element) 75
GoToR (action element) 75
Graphic (common Mars element) 89, 165
Group (backbone element) 104
Groups (backbone element) 104

H

Hide (action element) 75, 153
Highlight (markup annotation element) 83

I

image (SVG element) 48
images
    adding to SVG content 48
    supported formats 46
images and color profiles
    adding to SVG content 46–49
ImportData (action element) 75
Ink (markup annotation element) 83
IsMap (action attribute) 77
ItalicAngle (font descriptor attribute) 41

J

java.util.zip (Java class) 17
JavaScript (action element) 75
JavaScript (javascript element) 80
JavaScript expressions 79
JavaScripts (backbone element) 80

K

Key (backbone attribute) 80

L

Language (backbone element) 105

LastChar (font descriptor attribute) 40
Launch (action element) 75
Leading (font descriptor attribute) 41
ligature formation 40
Line (markup annotation element) 83, 93
Link (content annotation element) 82, 88
logical structure
about 122
node-naming syntax 127–129
private data 169, 173
specifying 131

M

pdf:Mark (Mars extension to SVG 108, 130
Marked (backbone element) 130, 173
marked content
about 121
private data properties 169
specifying 130
markup annotations
Caret 83
Circle 83
creating 89–99
file attachment 95
FileAttachment 83
FreeText 83
Highlight 83, 92
Ink 83
Line 83
Polygon 83
popup 83
Sound 83
Square 83
Squiggly 83
Stamp 83
StrikeOut 83
Text 83
types of 83
Underline 83
Mars extension to SVG
font-information 38
Mark 108, 130
Membership 108
Props 108, 169
MaxWidth (font descriptor attribute) 41
MCR (page structure element) 135
pdf:Membership (Mars extension to SVG) 108
metadata
about 117
associated with document 117
associated with document resources 117
document-level 118
MissingWidth (font descriptor attribute) 42
Movie (action element) 75, 153
Movie (content annotations) 82

N

Name (font descriptor element) 40
name (SVG attribute) 38
Named (action element) 75
Normal (common Mars element) 89, 163, 165

O

obfuscation of embedded fonts 43, 44
OCProperties (backbone element) 104
OFF (backbone element) 105
ON (backbone element) 105
OnActivation (backbone element) 88, 162, 164
OnCursorEnter (action element) 153, 154
OnEvent (action element) 154
OnEvent (backbone element) 153
OnMouseDown (action element) 153, 154
OnPageVisible (action element) 153, 154
OpenType fonts
about 35
converting to 45
optional content
marking non-SVG files for 109
marking SVG files for 108
optional content, creating 103–110

P

packaging a Mars document 16–17
Page (page structure element) 30
page placement 28
pages, developing 27–34
Param (embedded files element) 112
Params (backbone element) 96
Pattern (structure cache element) 72, 137
PDF packages. *See* collections
Polygon (markup annotation element) 83
Popup (content annotation element) 83
Popup (markup annotation element) 83, 91
portable collections. *See* collections
PresentationOrder (backbone element) 105
PrinterMark (content annotation element) 82
Private (common Mars element) 172
private data
about 168
logical structure 169, 173
marked content 169
pdf:Props (Mars extension to SVG) 108, 130, 169

Q

Query (structure cache element) 72, 137

R

rdf:RDF (XMP metadata element) 118
relationships (Adobe Container Format) 17
Rendition (action element) 75
ResetForm (action element) 75
Role (backbone element) 133

role maps (logical structure) 133
RoleMap (backbone element) 133
Rollover (common Mars element) 163, 165
root file. *See* backbone file
RTContents (markup annotation element) 91

S

sCapHeight (OpenType OS/2 property) 41
Schema (backbone element) 113
Screen (content annotation element) 82
Script (action element) 78
Script (javascript element) 80
SetOCGState (action element) 75
Sort (backbone element) 113
SortField (backbone element) 113
Sound (action element) 75, 77
Sound (markup annotation element) 83
SoundFile (action element) 78
Square (markup annotation element) 83
Squiggly (markup annotation element) 83
Stamp (markup annotation element) 83
StemH (font descriptor attribute) 41
StemV (font descriptor attribute) 41
StrikeOut (markup annotation element) 83
string 16 (OpenType name property) 41
string 6 (OpenType name property) 41
Structure (page information element) 136
Structure (page structure element) 135
structure cache 137
structure of a Mars document. *See* document structure
structure tree 122
StructureTypes (backbone element) 133
sTypoLineGap (OpenType OS/2 property) 41
SubFields (backbone element) 151
SubmitForm (action element) 75, 164
SVG
    coordinate system 29
    fonts 35
    page content 28
SVG coordinate system. *See also* user space
sxHeight (OpenType OS/2 property) 41

T

tagging
    about 129
    specifying 131
Text (markup annotation element) 83, 90
TextField (backbone element) 142, 151, 153
Thread (action element) 75
ThreadInfo (articles element) 101
Threads (articles element) 101
Threads (backbone element) 100
Title (bookmark element) 64
Trans (action element) 75
transform (SVG attribute) 29
TrapNet (content annotation element) 82
TrueType (font descriptor element) 40
Type0 (font descriptor element) 40
Type1 (font descriptor element) 36, 40

U

Underline (markup annotation element) 83
URI (action element) 75, 77
Usage (backbone element) 105
UsageApplications (backbone element) 105
user space 29
UserProperties (page structure element) 173
UserProperty (page structure element) 173
usWeightClass (OpenType OS/2 property) 41
usWidthClass (OpenType OS/2 property) 41
usWinAscent (OpenType OS/2 property) 41
usWinDescent (OpenType OS/2 property) 41

W

Watermark (content annotation element) 82
Widget (content annotation element) 82
Widths (font descriptor element) 36, 40
WinZip 17

X

x:xmpmeta (XMP metadata) 118
xAvgCharWidth (OpenType OS/2 property) 41
XDP 149
XFA (backbone element) 142, 150
XFA template field elements 143
XHeight (font descriptor attribute) 41
xMax (OpenType head property) 41
xMin (OpenType head property) 41
XML form
    dynamic 140
    static 140
XML forms
    about 140
    actions 158
    adding form parts 149
    choice fields 145
    coordinate translation 146
    creating 148–160
    creating button fields 152
    creating choice fields 154
    creating text fields 153
    data associated with 141
    dynamic 140
    form field widgets 140
    form fields 140
    referencing form parts 150
    relationship between parts 143
    static 143
    XDP 149
    XFA template 141
XMP. *See* metadata
XYZ (common Mars element) 66, 76

Y

yMax (OpenType head property) 41
yMin (OpenType head property) 41