Adobe FrameMaker

Developing Structured Applications

September 2022

Contents

Part III Working with an EDD

Part IV Translating Between Markup Data and FrameMaker

Before You Begin

This developer's guide and its associated reference manual are for anybody who develops structured FrameMaker® templates and XML or SGML applications. They are not written for end users who author structured documents that use such templates and applications.

XML and SGML

FrameMaker can read and write XML (Extensible Markup Language) and SGML (Standard Generalized Markup Language) documents. XML and SGML are both document markup languages, and FrameMaker handles these markup languages in similar ways. However there are differences between the two, and this manual covers these differences whenever necessary.

When discussing the similarities between them, this manual refers to XML and SGML data as *markup data* or *markup documents*. Otherwise the manual refers to XML and SGML specifically to draw attention to the differences between these markup languages. The majority of new structured documentation projects are XML based, therefore XML now takes precedence over SGML where necessary.

Developing structured FrameMaker templates

End users of FrameMaker can read, edit, format, and write structured documents—the structure is represented by a hierarchical tree of elements. Each structured document is based on a template that contains a catalog of element definitions. Each element definition can describe the valid contexts for an element instance, and the formatting of element instances in various contexts.

To support these end users, you create the catalog and accompanying structured template.

Developing XML and SGML applications

When FrameMaker reads markup data, it displays that data as a formatted, structured document. When the software saves a structured FrameMaker document, the software can write the document as XML or SGML.

For the end user, this process of translation between FrameMaker documents and markup data is transparent and automatic. However, for most XML or SGML document types the translation requires an XML or SGML application to manage the translation. You develop this application to correspond with specific document types. When your end user opens a markup document with a matching document type, FrameMaker invokes the appropriate structure application. If there is no

match for a document type, the user can choose the application to use, or open the markup document with no structure application.

A structure application primarily consists of:

- A structured template
- A DTD or schema
- Read/Write rules (described in this manual)
- XSLT style sheets for pre and post process transformations (if necessary)
- An XML and SGML API client (if necessary) which is developed with the Frame® Developer's Kit (FDK)

Prerequisites

The following topics, which are outside the scope of this manual, are important for you to understand before you try to create a structured template or structure application:

- Structured document authoring in FrameMaker
- XML or SGML concepts and syntax, including how to work with a *document type definition*
- FrameMaker end-user concepts and command syntax
- FrameMaker template design

In creating some XML or SGML applications, you may also need to understand the following:

- XSLT 1.0
- C programming
- FDK API usage

If your application requires only the special rules described in this manual to modify the default behavior of FrameMaker, you do not need programming skills. However, if you need to create an XML and SGML API client to modify this behavior further, you need to program the client in C, using the FDK. This manual does not discuss the creation of XML and SGML API clients. For this information, see the *Structure Import/Export API Programmer's Guide*.

Using FrameMaker documentation

FrameMaker comes with a complete set of end-user and developer documentation with which you should be familiar. You can access the FrameMaker guides from the FrameMaker help and support page, http://www.adobe.com/support/framemaker/.

If you use the Frame Developer's Kit in creating your structure application, you will also need to be familiar with the FDK documentation set.

Using this manual

This manual is divided into four major parts. Each part begins with a short introduction in which you can find information about specific chapters in that part.

If you are creating a structure application, you should read the first three chapters of Part II before reading the remainder of the manual. You will then find information you need in the other three parts.

If you are not working with markup data but are creating a structured template, you will need only Part III.

The parts are as follows:

- Part I, "FrameMaker Structure Application Quick Start"

  This part has been designed to give developers an easy way to create a simple structure application. Once these simple step-by-step instructions are followed it will be easier to get the best out of the remainder of this developer guide.

- Part II, "Developing a FrameMaker Structure Application"

  Part II is for developers of XML or SGML structure applications. This section contains

  –Introductory information

  –An overview of the steps in creating a structure application

  –A comparison of FrameMaker concepts and XML and SGML

  –Details of assembling the pieces of an application into a whole

- Part III, "Working with an EDD"

  Part III is for developers of FrameMaker structured templates. It describes how to use an element definition document (EDD) to define elements and determine their formatting for your documents. Use this part together with chapters in the *Using FrameMaker* that describe other aspects of template creation.

- Part IV, "Translating Between Markup Data and FrameMaker"

  Part IV is for developers of structure applications. This part describes the model and the rules you use to modify the default translation between markup documents and FrameMaker documents

Typographical conventions

`Monospaced font`	Literal values and code, such as XML, SGML, read/write rules, filenames, and path names.
Italics	Variables or placeholders in code. For example, in `name="`*`myName`*`"`, the text *`myName`* represents a value you are expected to supply. Also indicates the first occurrence of a new term.
Blue underlined text	A hyperlink you can click to go to a related section in this book or to a URL in your web browser.
Sans-serif bold	The names of FrameMaker *User Interface* objects (menus, menu items, and buttons). The **>** symbol is used as shorthand notation for navigating to menu items and sub menus. For example, **Element > Validate...** refers to the **Validate...** item in the **Element** menu.

Using other FrameMaker documentation

The *Using FrameMaker* makes up the primary end-user documentation for this product. It explains how to use the FrameMaker authoring environment for both structured and unstructured documents. It also explains how to create templates for your documents.

In creating a structured template, you can refer to this manual for information on how your end user interacts with the product and how to create a formatted template.

You will also find a range of other online documents from the FrameMaker help and support page, http://www.adobe.com/support/framemaker/.

Using FDK manuals

If you create an XML and SGML API client for your XML or SGML application, you'll need to be familiar with the FDK. FDK documentation is written for developers with C programming experience.

- *FDK Programmer's Guide* is your manual for understanding FDK basics. This manual describes how to use the FDK to enhance the functionality of FrameMaker and describes how to use the FDK to work with structured documents. To make advanced modifications to the software's default translation behavior, refer to the *Structure Import/Export API Programmer's Guide*.)
- *FDK Programmer's Reference* is a reference for the functions and objects described in the *FDK Programmer's Guide*.

- An *FDK Supplement* adds information about FDK features that have been changed or added since the release of FrameMaker 8.0.
- *Structure Import/Export API Programmer's Guide* explains how to use the FDK to make advanced modifications to the software's default behavior for translation between markup documents and FrameMaker documents. This manual contains both descriptive and reference information.

For information on other FDK manuals, see "Using Frame Developer Tools" in the *FDK Programmer's Guide*.

Part I FrameMaker Structure Application Quick Start

Part I provides beginners information for developing structure applications. The chapters in this part are:

- Chapter 2, "Quick Start overview" provides an overview of the Quick Start chapters for typical application development scenarios.
- Chapter 3, "Create a new structure template" describes how to create a simple EDD and a structured template. This is the easiest way to start with structured documentation.
- Chapter 4, "Create your first XML application" explains the conversion of your first structured template into a real XML application.

1 Quick Start overview

Developing a FrameMaker structure application can appear complex at first. True, there is ultimately a lot to learn, but the overall process is easily broken down into manageable topics.

This Quick Start section presents the first two stages in FrameMaker structure application development. Choose the development process that matches your requirements then follow the simple instructions in the following chapters.

When you have completed the Quick Start chapters, move on to the main parts of the Developer Guide.

Create a new structure template

This is the quickest way to get started with FrameMaker structure development. You will create an *Element Definition Document (EDD)* that defines a document structure and its formatting rules. This EDD is then imported into your first structured template. The documents that you create are in FrameMaker format, so although you will not be saving the results to markup, you will be able to be benefit from the many advantages of guided authoring.

To learn how to create a new structure template see Chapter 3, "Create a new structure template."

Create your first XML application

Adding the ability to open and save as XML is the next step in developing FrameMaker structure applications. You will export a DTD from your EDD. You will also create an XML application definition along with the read/write rules that are used to map FrameMaker's objects to and from XML markup.

To learn how to create your first XML application see Chapter 4, "Create your first XML application."

> **Important:** The step-by-step instructions in the following QuickStart chapters assume that FrameMaker is set up to automatically insert child elements.

2 Create a new structure template

This Quick Start chapter guides you through the first stages of structured application development. After following these instructions you will understand the FrameMaker structure model and how to build a structured FrameMaker template.

The intention here is to introduce the basic of the features of a structured template, rather than to build the perfect structure application. After working through this chapter you should be able to adapt the results to fit your own requirements.

Prerequisites

The following topics, which are outside the scope of this manual, are important for you to understand before you begin to work through this chapter:

- Structured document authoring in FrameMaker
- FrameMaker end-user concepts and command syntax
- FrameMaker template design

The Structured Template

All FrameMaker structure applications require a structured template. There is actually very little difference between a structured template and an unstructured template.

The template document

A FrameMaker template is just another FrameMaker document. Before you start to customize it to your requirements its the same as any other new document that you create using **File > New > Document...** and then select **use blank paper** as required. At this stage the template is unstructured.

The following sections explain some of the features of a structured template.

Element catalog

The most obvious difference between unstructured and structured templates is the Element Catalog. The Element Catalog stores the structure definitions and context based formatting rules that you will design and create.

The structured template's end-users use the element catalog dialog to select valid elements to insert within the structure they have created.

As the developer of the structured template you will use the *Element Definition Document* or *EDD* to create the structure definitions and formatting rules. This EDD is then imported into your template creating, or updating the element catalog. The creation of a simple EDD is explained in "The Element Definition Document" on page 26.

Master Pages

Master pages in a structured template are identical to their unstructured equivalent. In a normal structured FrameMaker document only the main flow—typically flow A—makes use of the element catalog. All other text frames should be formatted using unstructured paragraph formats and character formats.

> **Note:** It is recommended that the formatting on a master page is always applied with paragraph and character formats without any overrides. This becomes important if the template is eventually used in a markup application. Format overrides are often lost on import from markup.

Running Header/Footer variables

Running header footer variables in structured documents allow the use of element content and attribute value building blocks in addition to the unstructured paragraph based building blocks.

Paragraph Catalog

As mentioned in the previous section, paragraph formats are still very important in the design of a structured template. Paragraph formats are also one of the ways that you can apply formatting in an EDD.

Format Catalog

Character formats are the only way to apply formatting to autonumber text in a structured template.

Tables

FrameMaker table formats are only partially integrated with the structure model. The content of a FrameMaker table is always fully structured, but the control over cell border ruling is limited and there is no structure format control over cell shading.

The Element Definition Document

This section will guide you through the various stages in the development of an EDD. It is not a definitive guide, that information can be found in Part III of this manual.

> **Note:** An EDD is itself just another structured document, although more complex than the EDD you are about to create. You should already be familiar with structured editing techniques.

A simple document structure

To demonstrate the construction of an EDD we will use the simple document structure shown below. In a real project you would need to analyze your requirements and design a structure to suit them.

Create an EDD

You will now build an EDD for a simple document. Follow these steps to build the structure definitions, the formatting will be added later:

Build the structure

1.Select **StructureTools > New EDD**. A new empty EDD is created which will look like this:

EDD Version is <*version*>
Automatically create formats on import.

Element:

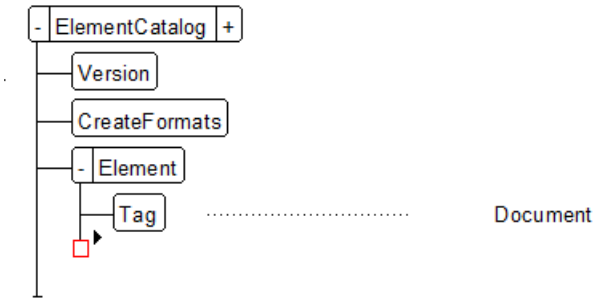
2.Save the new EDD with a relevant file name.

3.Add the *Tag* of the top level element which will be `Document`:

EDD Version is <*version*>
Automatically create formats on import.

Element: Document

4.After the `Tag` element insert the `Container` element which will also add its child element `GeneralRule`.



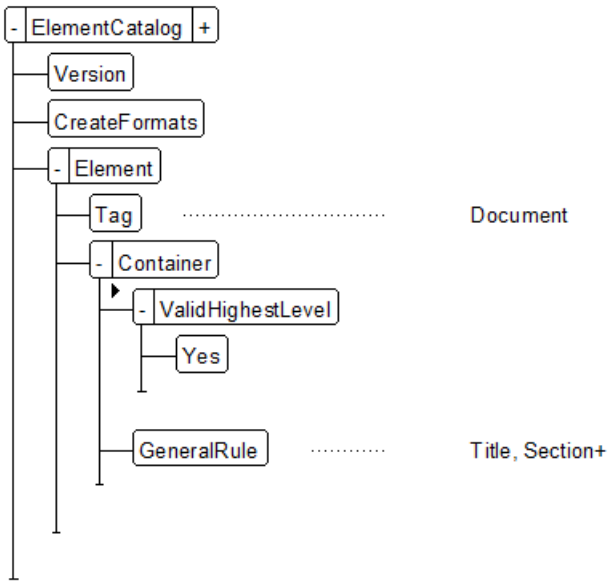
The EDD will now look like this:

Element (Container): Document
General Rule:

5. The `GeneralRule` element is where you define the structure content rules for the element. In the structure diagram you can see that the `Document` element must contain a single `Title` element followed by one or more `Section` elements. In a general rule this is represented as shown here:

   Element (Container): Document
   General Rule: Title, Section+

6. The `Document` element is at the top level of the structure, so you must add a `ValidHighestLevel` element as the first child of `Container`.



Important: Making an element valid at the highest level means that it will be listed in the Element Catalog dialog for an empty structured document. This provides a starting point for the author of a structured document.

7. Add an element definition for each element that you defined in the `GeneralRule`. Start by adding an `Element` element.

8. Add the element's name to the `Tag` element.

9. Insert the `Container` element.

10. For the `GeneralRule` of your `Title` element definition type *<TEXT>* including the angle brackets.

Important: <TEXT> means that the end user can type text directly into this element.

11. For the `GeneralRule` of your `Section` element definition type

    `Title, (Paragraph | Table | Graphic)*, Section*`

    This General Rule states that each Section must start with a Title, which is followed by zero or more `Paragraph`, `Table` or `Graphic` elements in any order. It ends with zero or more `Section` elements.

Your EDD will now look like this:

EDD Version is *<version>*
Automatically create formats on import.

Element (Container): Document
Valid as the highest-level element.
General Rule: Title, Section+

Element (Container): Title
General Rule:<TEXT>

Element (Container): Section
General Rule: Title, (Paragraph | Table | Graphic)*, Section*

12.Add the remaining element definitions as given in the following table:

Element name	Type	General Rule	Notes
Paragraph	Container	(<TEXT> | List | Emphasis | Xref)*	
List	Container	Item+	
Item	Container	(<TEXT> | Emphasis | Xref)*	
Emphasis	Container	<TEXT>	
Xref	CrossReference	*Not Applicable*	A CrossReference is a FrameMaker object element that cannot include any child elements.
Table	Table	TableHead?, TableBody	A Table object element may only contain TableHeading, TableBody and TableFooting object elements
TableHead	TableHeading	Row+	TableHeading and TableBody object elements may only contain TableRow object elements.
TableBody	TableBody	Row+	
Row	TableRow	Cell+	TableRow object elements may only contain TableCell object elements.
Cell	TableCell	<TEXT>	TableCell object elements may contain any type of element except other table part elements

Element name	Type	General Rule	Notes
Graphic	Graphic	*Not Applicable*	A Graphic is a FrameMaker object element that cannot include any child elements.

Add attribute definitions

Attributes can provide a wide range of additional functionality in a structured application. For this simple structured template attributes will enable the robust ID/IDREF system for internal cross references.

A single `idref` attribute will be added to the `Xref` element, which will store the value of a referenced element's `ID` attribute.

1.Find the element definition for `Xref`.

2.Add the `AttributeList` element as a child of `CrossReference`.

Element(CrossReference): Xref
Attribute list
1. **Name:**

3.Add a suitable `Name` for the attribute, such as "`IDref`".

4.Add the `IDReference` element after `Name`, then add the `Required` element. The element definition should now look like this:

Element(CrossReference): Xref
Attribute list
1. **Name:**IDref **ID Reference** **Required**

As the developer you must choose which elements can be cross referenced. This is done by adding an `ID` attribute to each potential source element. Let's start by adding the `ID` attribute to the `Document`, `Section`, `Paragraph` and `Table` element definitions.

5.Add the `AttributeList` element as the last child of `Container`.

Element(Container): Document
Valid as the highest-level element.
General Rule: Title, Section+

Attribute list
1. **Name:**

6.Add a suitable `Name` for the attribute, such as "`ID`".

7.Add a `UniqueID` after Name, then add the `Optional` element. The attribute list should now look like this:

Attribute list

1. Name:ID **UniqueID** **Optional**

Note: When an end user creates a cross reference FrameMaker automatically generates a unique value for the source element's `UniqueID` attribute.

If you prefer that all ID values are provided, define the attribute as `Required`. The end user will then be prompted for an ID value when the new element is created. FrameMaker will ensure that the `ID` values provided are unique by rejecting any duplicates.

8.Repeat steps 2 to 4 of this Add attribute definitions section for each of the other element definitions that are to have an `ID` attribute (the `Document`, `Section`, `Paragraph` and `Table` element definitions).

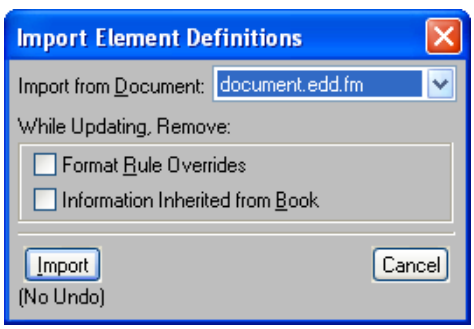
Note: Instead of building the structure element by element, simply copy and paste the `ID` attribute definition into the correct position in each element definition.

9.The initial structure is now complete, so you can test it by importing the EDD into your first template.

10.If you already have a template, open it. Otherwise open a new empty FrameMaker document.

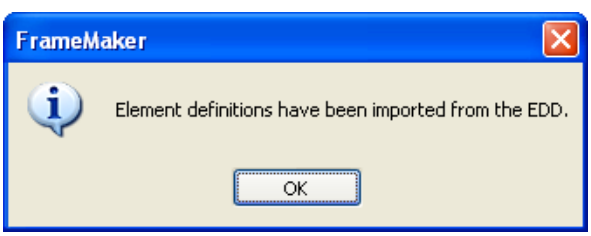
Test the structure

1.Import the element definitions from the EDD: **File > Import > Element Definitions...**



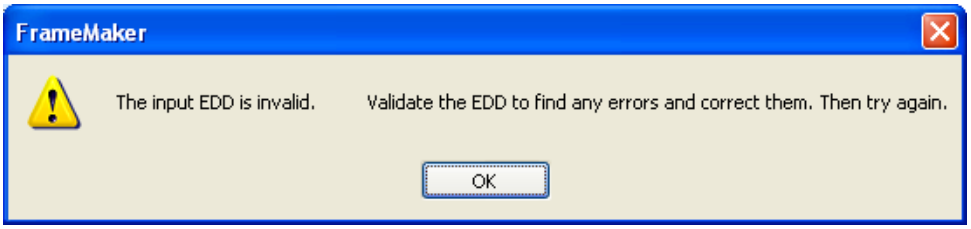
2.Click **Import** which will add the element catalog to the template document.

3.If your structure is valid you will see the information message:



4.If there is a problem with the EDD you will see an error message. The error message will either indicate that the EDD is structurally invalid, or that there are errors in the General Rules.

A structurally invalid EDD displays a warning message dialog. Go back to the EDD and fix any errors: select **Element > Validate...** fix the problems then try again.



Other EDD errors are presented as an Element Catalog manager report which will list each problem.

Element Catalog Manager Report

August 1, 2009
Imported EDD: C:\QuickStart\document.edd.fm
Destination Document: C:\QuickStart\document.tpl.fm

Elements that are defined but not referenced.
title

Elements that are referenced but not defined.
Title

Element Catalog Manager completed.

In the example above two errors are reported because the element definition for the `Title` element has been written without an initial capital letter. You must correct the error, then re-import the EDD. To make it easier to find the errors, the Element Catalog Manager Report provides active links to the source of each problem.

5.Once you have a valid template, use normal FrameMaker editing techniques to create a working test document. This will include at least one of each element in each required context.

The document that you create won't look very interesting because it is yet to include any formatting information. For the simple EDD you created earlier the document could look something like the following example...

Formatting a FrameMaker document¶
Formatting for containers¶
Here is some ¶
emphasized¶
text in a paragraph.¶
This is the second paragraph in the first section¶
Formatting for FrameMaker objects¶
This is a cross reference to "This is the second paragraph in the first section" on page 1¶
This section includes an anchored frame for a table followed by a graphic§

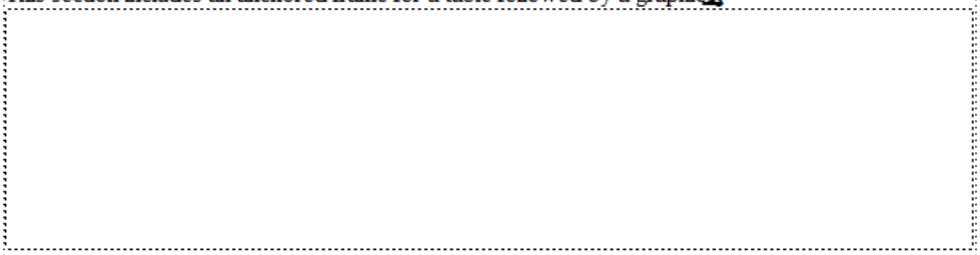
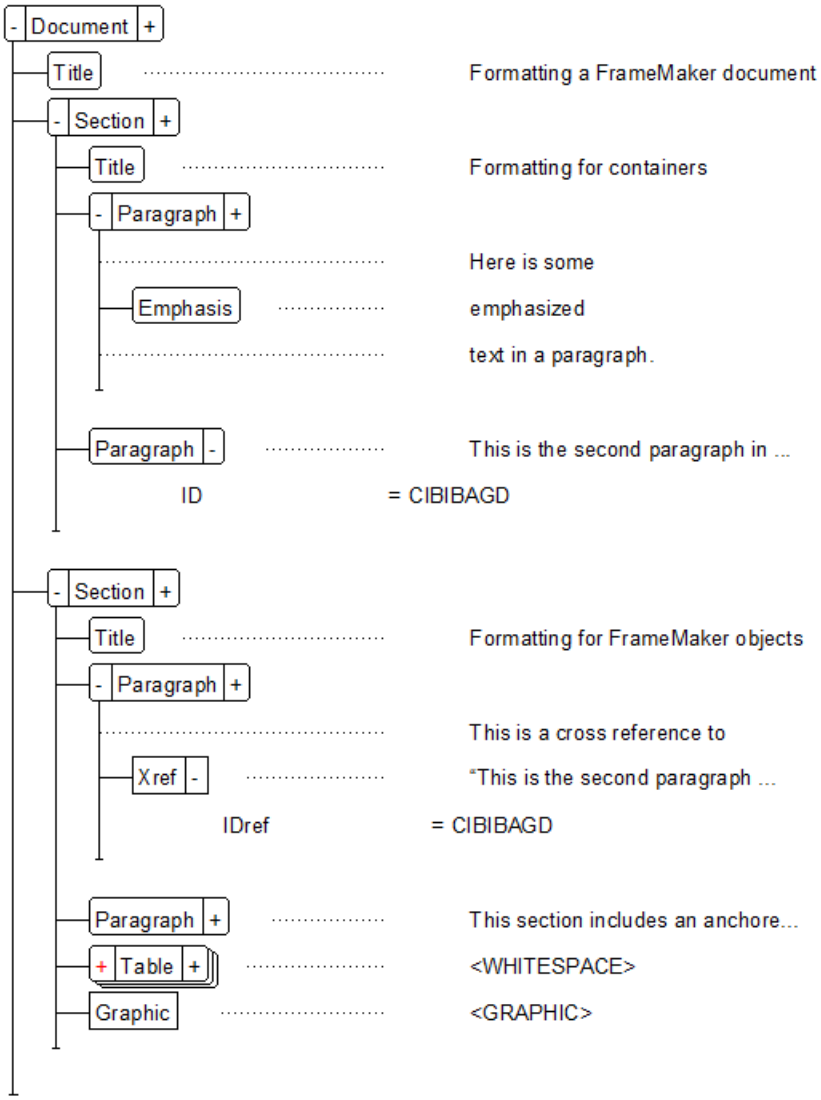


Table 1: §

Column1 Head§	Column2 Head§	Column3 Head§
Row1 Column1§	Row1 Column2§	Row1 Column3§
Row2 Column1§	Row2 Column2§	Row2 Column3§
Row3 Column1§	Row3 Column2§	Row3 Column3§

The structure view for this test document is presented on the next page. You may notice that the Table element already has a structural error. Also notice that the positions of the anchored frame and the table are reversed when compared to the document view above. These problems will be investigated and fixed in the following sections.



Add the format rules

Applying formatting to a FrameMaker structured document is also handled by the element catalog. In the EDD, cascading formatting rules are applied to some elements. If you apply the *Arial* font family to the top level, your entire document will use that font. However if the font family is changed for a lower level ancestor element that change then cascades down to its ancestors. In this way it is possible to apply complex and visually attractive formatting with the minimum of effort.

Basic Formatting

It's normal to start formatting at the top level element then move down the tree adding formatting rules where necessary until you have achieved the desired results.

1.In the EDD that you created earlier find the top level element `document`.

2.Add the `TextFormatRules` element as the last child of `Container`.

3.You now have the choice of four elements; `AllContextsRule`, `ContextRule`, `ElementPgfFormatTag` and `LevelRule`. Add the `AllContextsRule`, the other elements will be investigated later.

4.You will be formatting the `Document` element with properties that are equivalent to those found in the **Paragraph Designer**, so you must now add the `ParagraphFormatting` element.

> **Note:** The similarities between the **Paragraph Designer** and the `ParagraphFormatting` element are also reflected in its child elements:
>
> `-PropertiesBasic`
>
> `-PropertiesFont`
>
> `-PropertiesPagination`
>
> `-PropertiesNumbering`
>
> `-PropertiesAdvanced`
>
> `-PropertiesTableCell`
>
> `-PropertiesAsianSpacing`
>
> Each of these elements can contain child elements that are equivalent to the various formatting properties such as paragraph alignment, font weight, autonumber.

5.Insert the `PropertiesFont` element. The **Element catalog** will display a list of all font formatting property elements.

6.Add the `Family` element, then type "*`Arial`*"as its `<TEXT>` content. If Arial is not available choose a suitable alternative sans serif font.

7.Add the `Size` element, then type "*10pt*"as its `<TEXT>` content. The element definition for `Document` should now look like this:

Element(Container): Document
Valid as the highest-level element.
General Rule: Title, Section+

Attribute list

1. **Name:**ID **UniqueID** **Optional**

Text format rules

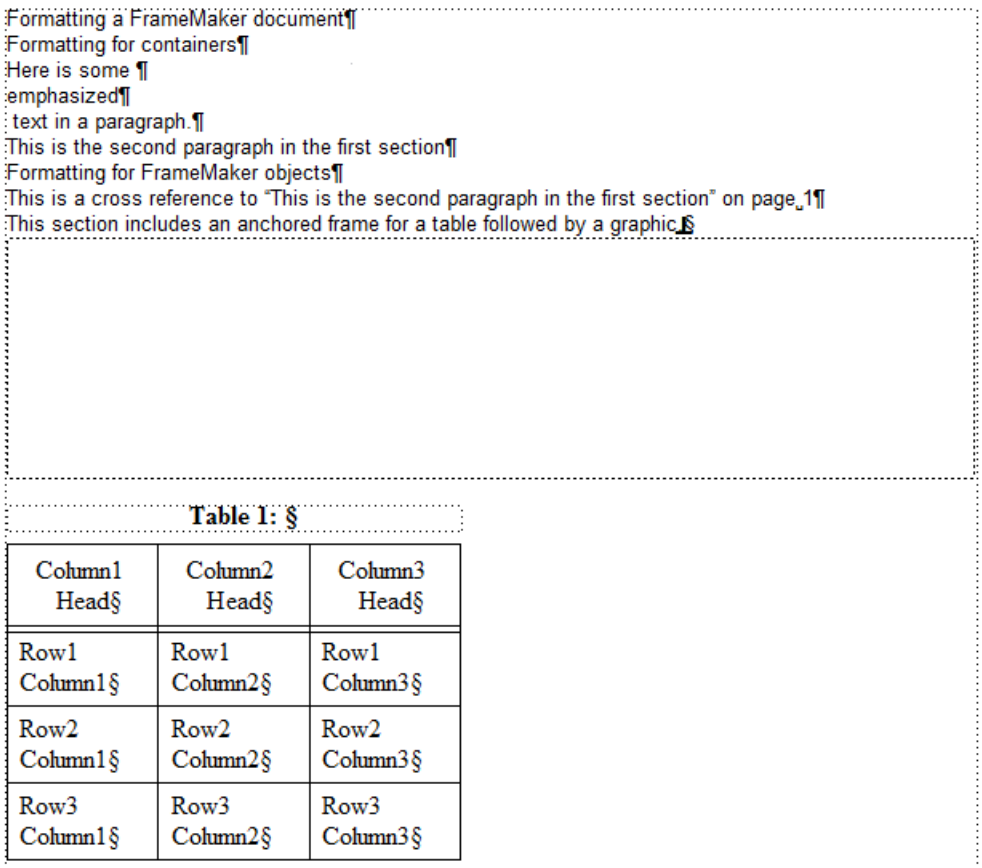
1. **In all contexts.**

Default font properties

Family: Arial
Size: 10pt

8.Save the EDD then switch to your test document.

9.Import the EDD. If there are no errors the document will now look like this:



10.Notice that the `Table` has not picked up the formatting that was defined for `Document`. This is normal behavior for FrameMaker tables. Copy the formatting rules from `Document` and paste them as the last child of `Table`.

11.Save the EDD then switch to your test document.

12.Import the EDD. If there are no errors the table formatting will now match the rest of the document:

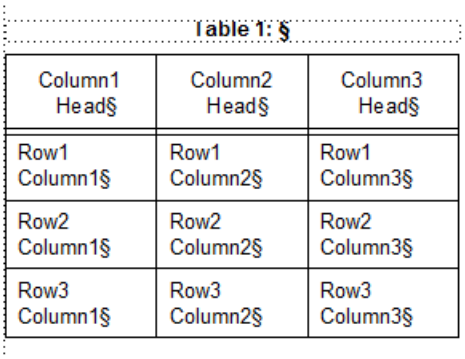


Table 1: §

Column1 Head§	Column2 Head§	Column3 Head§
Row1 Column1§	Row1 Column2§	Row1 Column3§
Row2 Column1§	Row2 Column2§	Row2 Column3§
Row3 Column1§	Row3 Column2§	Row3 Column3§

Format the Titles

Each title element needs to be formatted according to its context. The table below shows how formatting will be applied for each context.

Context	Format properties
Child of the `Document` element	Arial, Bold, 18pt
Child of a `Section` element that is a child of the `Document` element	Arial, Bold, 14pt, AutoNumber numeric (1.)
Child of a `Section` that is the child of a `Section` that is a child of the `Document` element Deeper nesting will not be formatted.	Arial Bold, 12pt, AutoNumber numeric (1.1)

You will notice that some formatting—`Family` and `Weight`—is used for all contexts. While `Size` and `Autonumber` depend on the element's context. You start by adding the common formatting.

1.Locate the element definition for `Title`.

2.Create an `AllContexts` rule as explained previously.

3.Add the `Weight` element and then its `Bold` child element.

> **Note:** There is no need to add a formatting rule for the font `Family` element as that has already been provided on the `Document` element.

4.Add a ContextRule element as the last child of TextFormatRules.

> **Important:** A ContextRule element can contain a list of mutually exclusive context specifications which are built as a series of If, ElseIf and Else statements. If is always the first statement, it can be followed zero or more ElseIf statements, then an optional Else statement. It works like this:
>
> If context A is true, apply format rule W
>
> ElseIf context B is true, apply format rule X
>
> ElseIf context C is true, apply format rule Y
>
> Else, apply format Z
>
> So, if contexts A, B or C are true the relevant format rules W, X or Y are applied. However if none of the supplied contexts are true, format Z is applied.

5.Add a context specification for the Title as a child of Document, simply type "*Document*" as the <TEXT> content of the Specification element.

6.Add the Size element, then type "*18pt*"as its <TEXT> content. The element definition for Title should now look like this:

Element(Container): Title
General Rule:<TEXT>

Text format rules

1. **In all contexts.**
   Default font properties
   Weight: Bold
2. **If context is:** Document
   Default font properties
   Size: 18pt

7.The next context for the Title element is as a child of Section. Remember that a Section can also be a child of Section, so the formatting rules will need to deal with nested Sections. Add an ElseIf element after the If.

> **Important:** The order of context specifications must always be to start with the most specific rule and end with the most general rule. If you do not follow this order for context rules, the most general rule will be applied for all contexts.

8.Add the context specification "*Section < Section*".

For this context add these formatting rules:

Property	Value
PropertiesFont < Size	*12pt*
PropertiesNumbering < AutonumberFormat	*<n>.<n+>\t*

9.Add another `ElseIf` element.

10.Add the context specification "*`Section`*".

For this context add these formatting rules:

Property	Value
`PropertiesFont < Size`	*`14pt`*
`PropertiesNumbering < AutonumberFormat`	*`<n+>\t`*

The text format rules for Title should now look like this:

Text format rules

1. **In all contexts.**
   Default font properties
   Weight: Bold
2. **If context is:** Document
   Default font properties
   Size: 18pt
   Else, if context is: Section < Section
   Default font properties
   Size: 12pt
   Numbering properties
   Autonumber format: <n>.<n+>\t
   Else, if context is: Section
   Default font properties
   Size: 14pt
   Numbering properties
   Autonumber format: <n+>\t

Important: Trouble-shooting context specification problems.

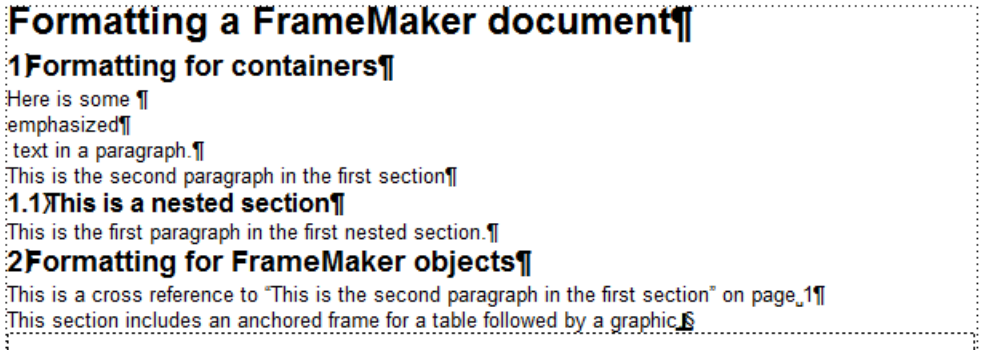
In the example above the context specification `Section < Section` will also be true for any number of additional nested Sections. To prevent that you must make the rule more specific. In this case adding the top level element makes the rule specific to a single context: `Document < Section < Section`.

Note: The EDD provides the `Level` rule as an alternative way to define contexts for nested sections. See Chapter 17, "Context Specification Rules," for detailed instructions.

11.Save the EDD then switch to your test document.

12.Import the EDD. If there are no errors the document will now look like this:

Note: A nested section has now been added to show the formatting for that context.



Indenting the Sections

The document is looking better, but it still needs some work to make it easy to read. In this section you will add an indent so that the section numbers are clear of the body text. Some extra paragraph spacing will also be added.

It is important that you select the most appropriate element to carry the formatting information. Choose the wrong element and you could find that additional rules have to be added to account for the unwanted side effects.

In this simple document you could add the required 0.3" indent to the `Document` element. That would work, but the document's Title element does not need the indent, so you would need to add another rule to remove the indent.

The indent could also be given to the `Paragraph` element which would be an acceptable choice for the current state of the EDD structure. However, as the document template evolves you may need to add other elements to the `Section` content rule. For this reason you will add the indent rule to the `Section` element.

1.Add these All Contexts Rules to the `Section` element definition.

Property	Value
`PropertiesBasic < Indents < FirstIndent`	*0.3″*
`PropertiesBasic < Indents < LeftIndent`	*0.3″*
`PropertiesBasic < TabStops < TabStop < TabStopPosition`	*0.3″*

2.The `Title` element will now pick up the indent that has been applied to `Section` which is not the desired result, so the first indent must be corrected. Add this All Contexts Rule to the `Title` element definition.

Property	Value
`PropertiesBasic < Indents < FirstIndentRelative`	*`-0.3"`*

Note: The EDD provides more control over some formatting properties than the unstructured Paragraph format. In the example above a relative value has been provided for the First indent property. The value is relative to the formatting that is inherited from the element's parent or an ancestor element.

3.You will now add some space above each `Title`. You could simply add the necessary formatting rules to the `Title` element definition. That would be adequate for the requirements of the current document, but there is a useful alternative, the `FirstParagraphRules`.

Locate the element definition for `Section`.

4.Add a `FirstParagraphRules` element as the last child of `Container`.

5.`FirstParagraphRules` accepts the same child elements as `TextFormatRules`, so use the same techniques to add the formatting:

Property	Value
`PropertiesBasic < ParagraphSpacing < SpaceAbove`	*`10pt`*

Important: The first paragraph rule applies formatting to the first child element, whatever that element may be. This can significantly reduce the amount of formatting rules required if an element has many element that could be the first child.

`LastParagraphRules` provides similar formatting control for the last child element.

6.Save the EDD then switch to your test document.

7.Import the EDD. If there are no errors the document will now look like this:

Formatting a FrameMaker document¶

1〉 Formatting for containers¶

Here is some ¶
emphasized¶
text in a paragraph.¶
This is the second paragraph in the first section¶

1.1〉This is a nested section¶

This is the first paragraph in the first nested section.¶

2〉 Formatting for FrameMaker objects¶

This is a cross reference to "This is the second paragraph in the first section" on page 1¶
This section includes an anchored frame for a table followed by a graphic.§

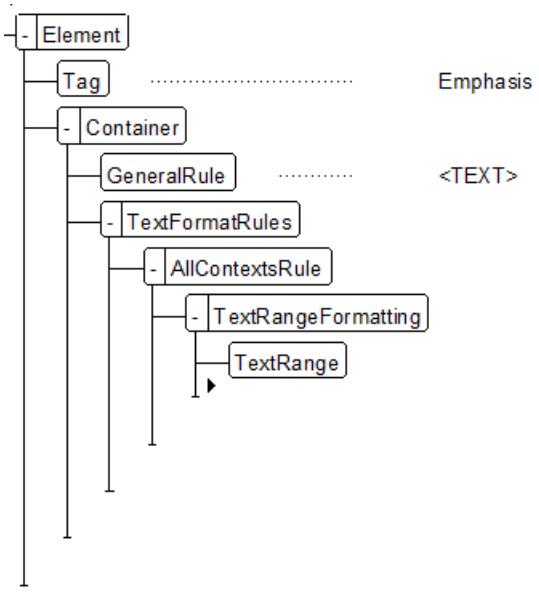
Text range formatting

You have already seen how the EDD's text format rules are equivalent to Paragraph formats. When character formatting is needed the EDD provides `TextRangeFormatting`. In the test document the `Emphasis` element needs to be formatted as a text range.

1.Locate the element definition for `Emphasis`.

2.Insert the `TextFormatRules` element as the last child of `Container`.

3.Add an `AllContextRule`, then a `TextRangeFormatting` child element.



4.After the TextRange element you can add the character formatting rules in one of three ways:

-CharacterFormatTag

-FormatChangeListTag

-PropertiesFont

Note: The CharacterFormatTag lets you apply a character format tag to the element. The character format will need to be defined in the structured template or document. If the character format is not available in the document when you import the EDD FrameMaker will create a format with that name, however it will not yet carry any specific formatting information. You will need to edit the character format using the Character designer.

A FormatChangeListTag is a reference to collection of formatting properties in the EDD. Its use is very similar to a character format tag. A format change list can include all formatting properties, but if it is referenced from TextRangeFormatting only the font properties take effect.

PropertiesFont lets you specify the font properties for the text range directly. You will use this method in the following step.

5.Insert the PropertiesFont element after TextRange.

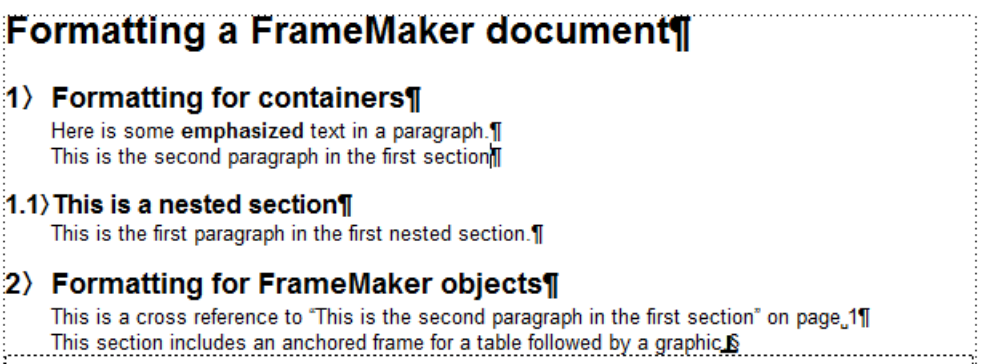
6.Add the Weight then the Bold elements. The element definition should look like this:

Element(Container): Emphasis
General Rule:<TEXT>

Text format rules
1. **In all contexts.**
Text range.
Font properties
Weight: Bold

7.Save the EDD then switch to your test document.

8.Import the EDD. If there are no errors the document will now look like this:



Formatting cross-references

So far you have been formatting FrameMaker container elements. A cross-reference is a FrameMaker element object and as such there are limitations to what can be done with the EDD.

You can specify the Initial cross-reference format, but no other properties. In this section you will set up two new cross-reference formats that can be selected by attribute value when an `Xref` element is inserted in the test document.

1.In the EDD add a new attribute definition to the element definition for `Xref`.

2.Add a `Name` for the attribute "*`Format`*".

3.Add a `Choice` after `Name`, then add the `Required` element. Finally add the `Choices` element then type "*`Para | Section`*". The attribute list should now look like this:

Attribute list

1. **Name:**IDref **ID Reference** **Required**
2. **Name:** Format **Choice** **Required**
   Choices: Para | Section

4.The initial cross-reference format will be based on the value of the `Format` attribute. Add the following `ContextRule`.

Context Specification	Cross-Reference Format
[Format="Para"]	Para
[Format="Section"]	Section

Note: These context specifications use the value of an attribute on the current element. You can also combine element and attribute context specifications. For more information see, Chapter 17, "Context Specification Rules,"

The Initial cross-reference format will now look like this:

Initial cross-reference format

1. **If context is:** [Format="Para"]
   Use cross-reference format: Para
   If context is: [Format="Section"]
   Use cross-reference format: Section

5.Save the EDD then switch to your test document.

6.Import the EDD. The first time you import it you will see an Element Catalog Manager Report:

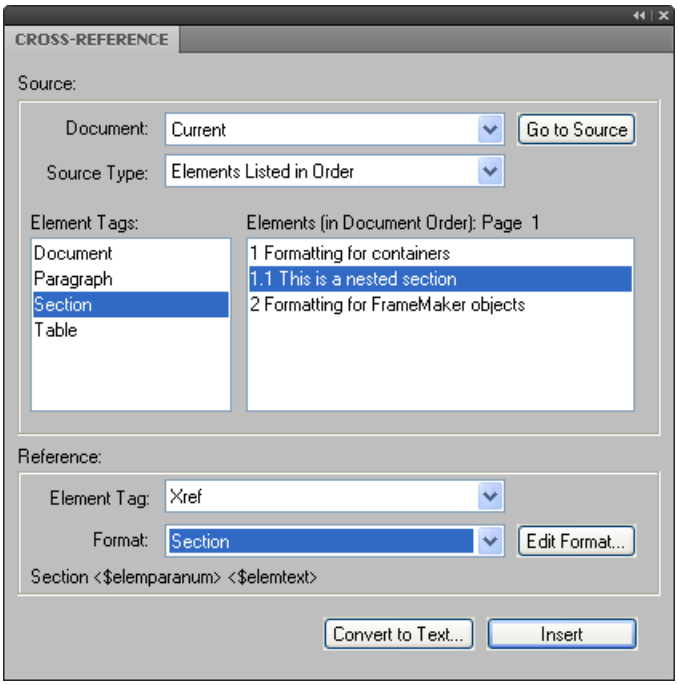
Element Catalog Manager Report

August 1, 2009
Imported EDD: C:\QuickStart\document.edd.fm
Destination Document: C:\QuickStart\document.tpl.fm

Messages...
Creating new cross-reference format (Para).
Creating new cross-reference format (Section).

Element Catalog Manager completed.

The new cross-references formats need to be completed. You can use element or attribute based building blocks to create a suitable format.



Anchored objects

Anchored objects such as Tables, Graphics and Equations may not always appear in the expected location. An example of this behavior can be seen in the test document where the position of the Table and the Graphic are reversed.

One way to ensure that anchored objects are positioned correctly is to use an anchor element. This is simply a container element that can contain the anchored object. By formatting the anchor with minimum line height and font size it has no effect on the design or pagination of the document.

1.Create a new element definition for the `Anchor` element. It should look like this:

Element(Container): Anchor
General Rule:(Table | Graphic)

Text format rules
1. In all contexts.
Basic properties
Line spacing
Height: 0pt
Default font properties
Size: 2pt

2.The content rule for the `Section` element must be changed so that the `Anchor` element replaces the `Table` and `Graphic` elements:

Element (Container): Section
General Rule: Title, (Paragraph | Anchor)*, Section*

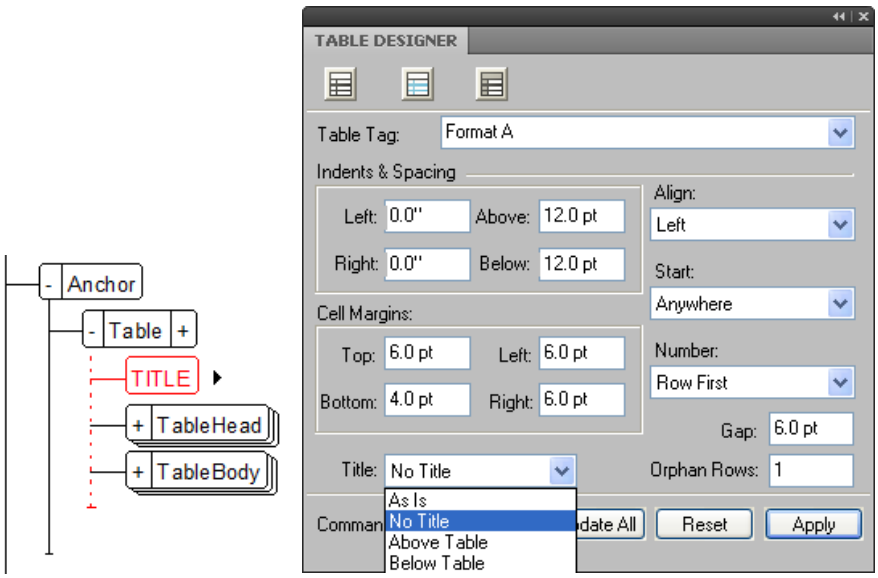
3.Save the EDD then switch to your test document.

4.Import the EDD. There will now be structure errors which can be fixed by wrapping the `Table` and `Graphic` elements in the new `Anchor` elements. The `Table` and `Graphic` elements will now be in the correct document order.

The EDD and table formats

As explained earlier the EDD has only limited control over FrameMaker table formats. You can use context rules to select the Initial Table Format, but once the table has been inserted into the document there is no further interaction.

If the table element definition does not include a table title element but the table format does, you will see an illegal `TITLE` element in the structure. To fix the problem ensure that the table format title position property matches the table element definition.



Formatting rules for Graphics

You can set up an attribute definition for the `Graphic` element that will allow the user to choose to insert either an anchored frame or an imported graphic file.

1.Create a new attribute definition for the `Graphic` element definition that looks like this:

Attribute list

1. **Name:** Type **Choice** **Optional**
   Choices: Import | AnchoredFrame

2.Create the context rules for Initial graphic element format:

Initial graphic element format

1. **If context is:** [Type="Import"]
   Insert imported graphic file.
   Else, if context is: [Type="AnchoredFrame"]
   Insert anchored frame.
   Else
   Insert imported graphic file.

> **Note:** The `Type` attribute has been defined as optional so that the user does not need to choose a value. When no value is chosen an imported graphic will be inserted. An alternative method would be to define a default value for the `Type` attribute. The `Else` statement would then be unnecessary.

Three ways to format

The EDD gives you three ways to apply formatting. In this Quick Start chapter you have used direct formatting for individual properties within context rules for `TextFormatRules` and

FirstParagraphRules. This method works very well for small to medium size EDDs, however it does not re-use any formatting definitions.

The Format Change List is a named group of formatting property elements. You can reference a Format change list as many times as you like from any context for TextFormatRules, FirstParagraphRules, LastParagraphRules and TextRangeFormatting. For more information see "Defining a format change list" on page 234.

As an alternative to specifying the formatting within the EDD, you can reference paragraph format tags for any context. This method can be used to good effect when converting an existing unstructured template. The document will look exactly the same as its unstructured predecessor and there will be no need to re-work cross-reference formats, generated lists or any of the other paragraph tag related items.

In practice it is normal to combine all three methods of formatting.

Deploy the template

Now that you have a viable structured template it can be used to create new documents. For each user, copy the template into the Templates folder in the FrameMaker installation folder. When a user needs to create a new structured document they will be able to easily find and select the template from the **New** dialog.

Next steps

This structured template development exercise shows some of the power and flexibility that FrameMaker delivers. There are of many more FrameMaker features that have not been mentioned here. Take the time to read the more detailed chapters in this Developer Guide then add features to the template that enhance its functionality.

3 Create your first XML application

In this Quick Start chapter you will convert your first structured template into a fully functional XML application capable of saving and opening XML with full *round-trip* ability.

The term *XML round trip* means that when a structured FrameMaker document is saved to XML an identical FrameMaker document is created when that XML instance is opened.

Define the XML application

A FrameMaker XML application is defined in the Structured Application Definition Document, which is also known as `structapps.fm`. Each XML application definition provides the configuration options which allow you to fine-tune an application's behavior. For this Quick Start XML application we will not need to use all of the available options. However when you are ready a comprehensive reference can be found in the Developer Reference, Chapter 1, Structure Application Definition Reference.

1. Open the Structured Application Definition Document by selecting **StructureTools > Edit Application Definitions**.
2. Insert an `XMLApplication` element immediately after the `Version` element.
3. Type a suitable name for the XML Application, for example "*`QuickStart`*".
4. Add the `DOCTYPE` element. You must type in the name of the EDD's top level element "*`Document`*".
5. Add the `Template` element. Type the path and file name for the structured template that you created previously, for example "*`C:\QuickStart\document.tpl.fm`*".

   Application Definition Version 9.0

   Application name: QuickStart
   DOCTYPE: Document
   Template: C:\QuickStart\document.tpl.fm
6. Select **StructureTools > Read Application Definitions** to enable the new XML application.
7. Save the `structapps.fm` file.

You will add more to the XML Application definition later.

Define the read/write rules

Read/write rules perform a variety of roles in a FrameMaker XML application. You will start by adding the essential rules for element type mapping.

1.Create a new read/write rules file by selecting **StructureTools > New Read/Write Rules**.

A read/write rules file is an unstructured text file. The file initially looks like this:

```
fm version is "<version>";
/*
 * Include all ISO entity mapping rules.
 */

#include "isoall.rw"
```

2.The #include directive for “isoall.rw” is not required for XML, so it may be removed.

3.Add a mapping rule for the `Xref` element. This rule identifies the element as a FrameMaker cross-reference then maps the cross reference format property to the `formatprop` attribute.

```
element "xref" {
is fm cross reference element "Xref";
attribute "formatprop" is fm property cross-reference format;
}
```

4.The mapping for a table is more complex because each table part needs a separate rule. The rule for `table` will ensure that cell border ruling override properties, number of columns and the width of each column are written to XML.

```
element "table" {
is fm table element "Table";
attribute "frame"
{
     is fm property table border ruling;
     value "top" is fm property value top;
     value "bottom" is fm property value bottom;
     value "topbot" is fm property value top and bottom;
     value "all" is fm property value all;
     value "sides" is fm property value sides;
     value "none" is fm property value none;
}
   attribute "colsep" is fm property column ruling;
   attribute "rowsep" is fm property row ruling;
attribute "numcols" is fm property columns;
   attribute "colwidths" is fm property column widths;
}
```

```
element "tablehead" {
is fm table heading element "TableHead";
}

element "tablebody" {
is fm table body element "TableBody";
}

element "row" {
is fm table row element "Row";
}

element "cell" {
is fm table cell element "Cell";
}
```

5.The rule for the Graphic element identifies the FrameMaker element type and provides instructions for the export of the contents of an anchored frame. In this case graphics will be saved with a file name based on the document's file name. The default behavior is retained for imported graphic files.

```
element "graphic" {
is fm graphic element "Graphic";
writer anchored frame {
   notation is "CGM";
   export to file "$(docname).cgm";
}}
```

6.Save the new read/write rules file into the same folder as the other application components.

> **Note:** In each of the read/write rules above, the element names have been slightly changed so that the XML saves as all lowercase while Capitalized names are used in FrameMaker.

Create a DTD

An XML application needs a DTD or schema for validation. In this section you will create a DTD from the EDD. FrameMaker cannot save an EDD as an XML schema.

1.Open the EDD that you created in the previous chapters.

2.Add the `StructuredApplication` element then type in the name of the XML application that you created in the previous section.

3.Select **StructureTools > Save as DTD...**

4.Enter a file name in the **Save As DTD** dialog. It is recommended that you use the *.dtd* extension.

5.If your EDD is valid you shouldn't see an error log. However, it is possible to build SGML style General rules in FrameMaker that are not permitted in XML.

6.A message dialog will tell you that FrameMaker has finished writing the DTD.

The DTD is now ready to be used.

Update the application definition

Now that you have a DTD for validation it can be referenced in the `structapps.fm` file.

1.Add the DTD element and type the path and file name for the DTD, for example "*`C:\QuickStart\document.dtd`*".

Update the structured template

The EDD has just been changed, so you need to update the application's structured template. Remember to do this whenever you change the EDD.

1.Open the structured template.

2.Open the EDD if its not already open.

3.Select **File > Import > Element Definitions...**

4.Select the EDD, then click **Import**.

5.Save the Template.

XML error logs

If the FrameMaker document was not valid when the file was saved you will see a Save as XML Log.

Note: This error was deliberately created by deleting a required `Title` element.

Save as XML Log
August 1, 2009
Source Document: C:\QuickStart\QuickStart1.fm
Destination Document: C:\QuickStart\QuickStart1.xml

XML Parser Messages (Document Instance)
Error at file C:\QuickStart\QuickStart1.xml.26F, line 24, char 43, Message: Element 'Paragraph' is not valid for content model '(Title,((Paragraph|Anchor))*,Section*)'

How to interpret the XML Log message The file name referenced in the parser message `QuickStart1.xml.26F` is a temporary file that FrameMaker creates while writing the real XML file. By the time you read the error message the temporary file will have been deleted. However,

the line number and character numbers are normally accurate if you need to check the XML output.

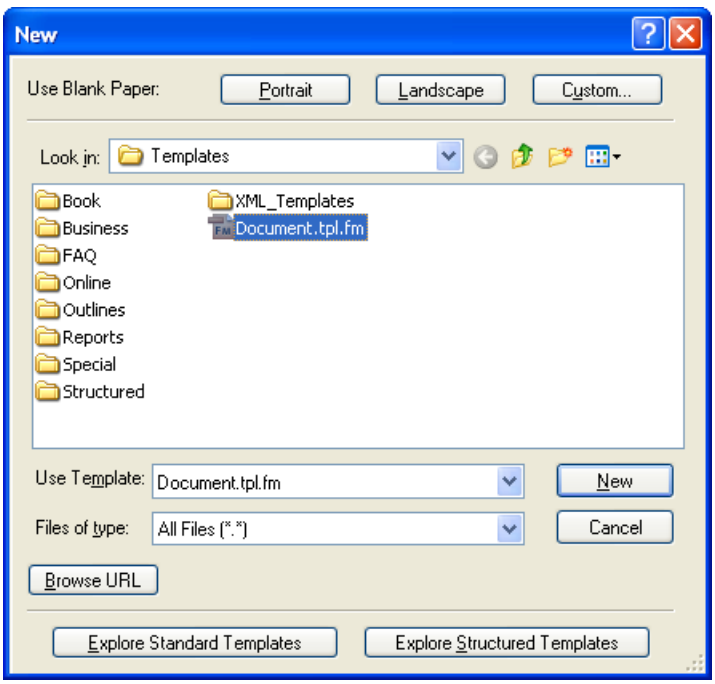
Paragraph is not valid for content model means that the first element encountered was `Paragraph` although the content model requires `Title` to be the first element.

Save as XML

Your XML application is now ready to open and save XML. Yes it's that easy to get started with XML!

1.Create a new document based on the structured template. Select **File > New > Document...**

2.In the **New** dialog select `Document.tpl.fm`, click **New**.



3.An empty structured document is created. Using normal FrameMaker editing techniques add text, tables, and graphics.

4.Validate then **Save** the document. A this stage the document is still in FrameMaker format.

5.To save the document as XML, simply select **File > Save As XML...** Provide a suitable name for the XML file, click **Save**. You will receive no feedback for a successful save.

XML error logs

If the FrameMaker document was not valid when the file was saved you will see a Save as XML Log.

Note: This error was deliberately created by deleting a required `Title` element.

Save as XML Log
August 1, 2009
Source Document: C:\QuickStart\QuickStart1.fm
Destination Document: C:\QuickStart\QuickStart1.xml

XML Parser Messages (Document Instance)
Error at file C:\QuickStart\QuickStart1.xml.26F, line 24, char 43, Message: Element 'Paragraph' is not valid for content model '(Title,((Paragraph|Anchor))*,Section*)'

How to interpret the XML Log message The file name referenced in the parser message `QuickStart1.xml.26F` is a temporary file that FrameMaker creates while writing the real XML file. By the time you read the error message the temporary file will have been deleted. However, the line number and character numbers are normally accurate if you need to check the XML output.

Paragraph is not valid for content model means that the first element encountered was `Paragraph` although the content model requires `Title` to be the first element.

Working with XML files

When you open an XML instance FrameMaker uses XML as its native format for that document. Native XML documents retain their `.xml` extension in FrameMaker and to save the file simply click **Save**, no need to use **Save As...** or **Save As XML...**

This behavior can be changed:

Edit the `maker.ini` file. In the `Preferences` section the setting is called `TreatXMLAsXML` it can have values of `On` or `Off` with a default of `On`.

Next steps

This XML application development exercise shows that round trip XML is easy to develop. There are of course many more FrameMaker features that have not been mentioned here. Take the time to read the more detailed chapters in this Developer Guide then add features to the application to suit your specific needs.

Many FrameMaker features are already enabled in the XML application. Try an XML round trip with **Track Text Edits** or **Filter by Attribute** enabled.

Part II Developing a FrameMaker Structure Application

Part I provides basic information for developing structure applications. The chapters in this part are:

- Chapter 5, "Developing Markup Publishing Applications" provides an overview of how you go about developing a FrameMaker structure application. Discusses the features of markup applications, and typical application development scenarios.
- Chapter 6, "Structure Application Basics" describes situations that require a structure application, and the basics of how to create a structure application.
- Chapter 7, "A General Comparison of Markup and FrameMaker Documents" compares relevant markup language and FrameMaker concepts. You should read this chapter even if you are already familiar with both the markup language (SGML or XML) and FrameMaker, since translation between the two is its own distinct topic. The chapter deals with counterpart constructs in the two representations, and also with constructs in one that have no real counterpart in the other.
  - –Chapter 8, "The XML and FrameMaker Models" compares XML and FrameMaker concepts.
  - –Chapter 9, "The SGML and FrameMaker Models" compares SGML and FrameMaker concepts.
- Chapter 10, "Creating a Structure Application" describes typical application creation workflow. Also discusses the types of files used in your final SGML application.
- Chapter 11, "Working with Special Files" tells where to find special files used by FrameMaker with XML and SGML documents. Also explains creation of the file that defines the pieces of your structure application.

4 Developing Markup Publishing Applications

FrameMaker provides a robust and flexible set of tools for creating markup editing and publishing applications. All general-purpose markup systems require application-specific information. You can use FrameMaker to develop such an application in a set of steps, each of which is straightforward.

The major parts of an application correspond to components in which the information is specified, as follows:

Application Component	File
Element and attribute definitions	EDD (can be derived automatically from a DTD or an XML schema)
Format rules	EDD
Page layouts	Template
Markup representation of elements and attributes	Read/write rules and structure API client
Markup structure transformation	Structure API client or XSLT for XML

The structure application bundles this information so that it can be easily accessed. Application development involves creating these files along with data analysis, documentation, and maintenance activities.

This chapter provides an overview of the development process. It contains these sections.

- "Markup applications," next, examines the features and components of markup applications.
- "Developing Adobe FrameMaker structure applications" on page 62 summarizes the process of developing a structure application, covering the steps common to many editing and publishing tools, and those unique to FrameMaker.
- "Technical steps in FrameMaker application development" on page 67 enumerates the tasks involved in developing a FrameMaker application and defines the files containing modules of the application that the application developer maintains.

Markup applications

A markup application is an application that manages and stores critical business information in a markup format. Here, markup refers to either XML (Extensible Markup Language) or SGML (Standardized General Markup Language).

There are two important aspects of markup that make it suitable for streamlining the creation, maintenance, and delivery of critical business information:

- Open standards
- Formal structure

If you can take advantage of these aspects, you will see improvements in your publishing process, and increased accuracy in the information you deliver. Because of open standards, you can share your information with more people using a wider variety of software applications. If you take advantage of the formal structure in markup, you can develop increasingly sophisticated ways to automate your processes.

Part of the formality of markup is that it separates content from appearance. The way markup separates a document's content from its appearance and intended processing makes it a natural format for single-source documents. However, the ability to use text in different ways requires that each use be defined. In markup terminology, such a definition is called an *application*.

More precisely, the SGML standard (ISO 8879, Standard Generalized Markup Language) defines a text processing application to be "a related set of processes performed on documents of related types." In particular, the standard states that an SGML application consists of "rules that apply SGML to a text processing application including a formal specification of the markup constructs used in the application. It can also include a definition of processing." This definition applies equally XML

Whether you are interested in XML or SGML, a publishing system that uses this technology is a general *markup application*. A markup application is the sum total of the tools in your XML or SGML publishing system. You work with these tools to author, process, and deliver your information in a variety of formats that can include print, PDF, HTML, or business transactions.

Understanding markup applications

You can see the need for markup applications by comparing general-purpose markup tools to traditional database packages. Just as database software is used to maintain and access collections of numeric data and fixed-length text strings, a markup system is used to maintain and access document databases. Both types of systems typify powerful, flexible products that must be configured before their power can be used effectively. In particular, both tools require the user to define the data to be processed as well as how that data will be entered and accessed.

Database	Markup
Depends on a schema to define record layout	Uses a type definition (DTD or XML Schema) to define structural elements and attributes
Input forms facilitate data entry	Configured XML or SGML text editor facilitates data entry
Reports present customized summaries and other views of information	Composed documents present information to end user

The *DTD* and accompanying input and processing conventions comprise the rules that define a markup application.

While markup applications need not include a visual rendering of documents—consider a voice synthesizer, word count, or linguistic analysis tool—many of them do involve publishing. Publishing is the process of distributing information to the consumer. Traditional publishing involves preparation of written materials such as books, magazines, and brochures. Computerized information processing facilitates the publishing of information in additional media, such as the Internet, mobile applications or CD-ROM, and also provides the possibility of interfacing with other applications, such as database distribution.

Applying formatting to markup

Often, the same information is published in multiple forms. In such cases, its appearance may change according to the purpose and capabilities of each medium. The appropriate rendering enhances the reader's comprehension of the text. Since markup prepares information for multiple presentations without making assumptions about its rendering, additional information is needed to control each rendering. This formatting information, although pertinent to an application, is outside the scope of the markup itself.

Consider, for instance, a repair manual stored in markup. It may include procedures such as the following:

```
<taskmodule skill="adv">
<heading>Tuning a 5450A Widget</heading>
<warning type="1">Do not hit a widget with a hammer. Doing so
could cause explosive decompression.</warning>
<background>The 5450A Widget needs tuning on a <emph>monthly
basis</emph> to maintain optimum performance. The procedure
should take less than 15 minutes.</background>
<procedure>
<step time="1">Remove the tuning knob cover.</step>
<step time="5">Use the calibration tool to adjust the setting to
initial specifications.</step>
</procedure>
</taskmodule>
```

The markup can be passed through a rendering application to present the mechanics with a formatted version:

Tuning a 5450A Widget

This procedure requires advanced certification.

Warning: Do not hit a widget with a hammer. Doing so could cause explosive decompression.

The 5450A Widget needs tuning on a *monthly basis* to maintain optimum performance. The procedure should take less than 15 minutes.

1. Remove the tuning knob cover.
2. Use the calibration tool to adjust the setting to initial specifications.

The application that prepares the formatted version applies rules indicating how the elements of the markup example are to appear. This example applies these rules:

- Put the `heading` element in a large, bold font.
- Generate the sentence, "This procedure requires advanced certification" from the value of the `skill` attribute.
- Insert the italic header "Warning:" before the text of the `warning` element.
- Number the steps in the `procedure`.

Notice that the values of the `time` attributes do not appear in the formatted text. The rules that drive a different application—a scheduling program, for instance—might use the `time` attribute to allocate the mechanic's time, for example. Or it might assemble the entire maintenance procedure (possibly thousands of such maintenance cards), and add up the times for each step to estimate the maintenance schedule.

Markup editing and publishing applications

Two important classes of markup applications involve creating (editing) and publishing documents. Editing tools are used to create markup documents; publishing tools produce formatted results from existing markup documents.

It is easiest to work with an editor if you have some visual indication of the document's structure, so markup text editors such as Adobe FrameMaker also require some formatting specifications. The editor can provides rules to determine, for example:

- How closely documents under development must conform to their type specification
- Options for listing available elements for the user
- Whether to automatically insert elements when the user creates a new document or inserts an element with required sub-elements
- Whether to prompt for attribute values as soon as the user creates a new element

Some tools address either the editing or the publishing application. FrameMaker is a single tool that addresses both. Although you can use FrameMaker for either activity alone, its strengths include the ability to create new structured documents (or edit existing ones) that are ready for publishing without additional processing. This pairing allows a single set of format rules to support both tasks. Thus, less effort is required to configure FrameMaker than to configure separate editing and publishing tools.

Developing Adobe FrameMaker structure applications

Developing a FrameMaker application shares many steps with developing a markup application for any other editing or publishing tool. This section summarizes the process, covering the steps common to many editing and publishing tools, and those unique to FrameMaker.

FrameMaker editing philosophy

Formatting, or visual clues—font variation, indentation, and numbering—help readers understand and use written information. FrameMaker is a *WYSIWYG* editor. While most *WYSIWYG* editors require authors to indicate how each part of the document is formatted, FrameMaker can use the rules in the markup application to format document components automatically in a consistent manner.

This combination of *WYSIWYG* techniques with the structured principles of markup is unique to FrameMaker. The goal in FrameMaker, is to streamline the process of creating and publishing documents through the use of markup. While authors manipulate elements and their attributes to create native markup documents, they work in a *WYSIWYG* fashion instead of directly with markup syntax.

Since the markup document model underlies edited material, the markup form of the document can be produced at any time during the *WYSIWYG* editing process. This process relies on the underlying element structure and hence has no need for the inaccurate filtering schemes that plague tools for generating markup from word processing formats.

Tasks in FrameMaker application development

The major tasks involved in the development of a FrameMaker application are:

- Defining the elements that can be used in a document and the contexts in which each is permitted. This task is analogous to developing a DTD. In fact, the element definitions can be automatically derived from a DTD if one exists.
- Determining which elements correspond to special objects, such as tables, graphics, and cross-references.
- Developing format rules that define the appearance of a structural element in a particular context.

- Providing page-layout information such as running footers and headers, margins, and so on—the foundation of every *WYSIWYG* document.
- Establishing a correspondence between the markup and FrameMaker representations of a document. For most elements, this correspondence is straightforward and automatic, but FrameMaker allows for some variations. For example, terse markup names can be mapped into longer, more descriptive FrameMaker names, and variant representations of tables, graphics, and other entities can be chosen.

Each of these tasks requires planning and design before implementation, as well as testing afterward.

The analysis phase

The specific features of a FrameMaker application largely depend on the tasks it will perform:

- Will users create new content, or will the application simply format existing markup documents?
- Whether or not they require further editing, are existing markup documents to be brought into the system? What about documents that are not XML or SGML (unstructured FrameMaker documents or, perhaps, the output of a word processor)? Will such legacy documents be imported on an ongoing basis or only at the beginning of the project?
- Will finished documents be saved as markup? Will authors themselves output markup, or will they pass completed projects to a production group that performs this post-processing step?
- Does the application involve a database or document management tool? Will portions of the documents be generated automatically? Are there calculations to be performed?
- Will there be a single DTD or a family of related DTDs used for different tasks (for example, a reference DTD used for interchange with a variant authoring DTD)?

Unless a project is based on one or more existing DTDs, one of the first outputs of the analysis phase is definition of the document structures to be used. The definition process usually involves inspecting numerous examples of typical documents. Even when there is a DTD to start from, the analysis phase is necessary to produce at least tentative answers to questions such as the previously listed ones.

Defining structure

A primary goal of the analysis phase is defining the structures that the end user will manipulate within FrameMaker. The bulk of this effort is often the creation of a DTD. When the project begins with an existing DTD, the DTD provides a foundation for the structure definition. FrameMaker, in fact, can use the DTD directly. Nevertheless, a pre-existing DTD does not eliminate the need for analysis and definition. As Eve Maler and Jeanne El Andaloussi explain in *Developing SGML DTDs: From Text to Model to Markup*, many projects use several related DTDs. If the existing DTD is intended for interchange, the editing environment can often be made more productive by creating an editing DTD. Several examples from the widely known DocBook DTD used for

computer software and hardware documentation illustrate the types of changes that might be made:

- Changing some element names, attribute names, or attribute values to reflect established terminology within the organization.

  For example, DocBook uses the generic identifier `CiteTitle` for cited titles. If authors are accustomed to tagging such titles as `Book`, an application might continue to use the element name `Book` during editing, but automatically convert `Book` to and from `CiteTitle` when reading or writing the markup form of the document.

- Omitting unnecessary elements and attributes.

  DocBook defines about 300 elements. Many organizations use only a small fraction of these. There is no need to make the remainder available in an interactive application. FrameMaker displays a catalog of the elements that are valid at a given point in a document. Removing unnecessary elements from the authoring DTD means the catalog displays only the valid elements that an organization might actually use in a particular context and avoids overwhelming the user with a much larger set of valid DocBook elements.

- Providing alternative structures.

  For example, DocBook provides separate element types—`Sect1`, `Sect2`, and so on—for different levels of sections and subsections. Defining a single `Section` element to be used at all levels (with software determining the context and applying an appropriate heading style) simplifies the task of rearranging material and hence provides a better environment for authors who may need to reorganize a document. Again, `Sections` can be automatically translated to and from the appropriate `Sect1`, `Sect2`, or other DocBook section element.

- Simplifying complex structures that are not needed by an organization.

  DocBook, for instance, provides a very rich structure for reporting error messages. If only a few of the many possibilities will actually be used, you can edit the DTD to eliminate unnecessary ones.

In addition to providing and editing the DTD, the analysis and definition of structure includes planning for the use of tables and graphics. Many DTDs provide elements that are clearly intended for such inherently visual objects. In other cases, however, such a representation of an element results from the formatting rules of the application. Consider again, for instance, the repair procedure on page 61. The formatted version did not display timing information for the procedure steps. This alternative version formats the procedure in a three-column table, showing the duration of each step:

Tuning a 5450A Widget

This procedure requires advanced certification.

Do not hit a widget with a hammer. Doing so could cause explosive decompression.

Step	Duration	Description
1	1 minute	Remove the tuning knob cover.
2	5 minutes	Use the calibration tool to adjust the setting to initial specifications.

Thus, defining structure involves defining the family of DTDs to be used in the project as well as deciding in general terms how the various elements will be used.

The design phase

The analysis phase of the project evolves into a design phase with two major goals: defining the desired results as well as the best way to accomplish those results. Defining the desired results includes the graphic design of the completed documents: page layouts as well as visual characteristics to be assigned to each structural element. Even when there is a rich body of sample documents and a tradition of consistent use throughout an organization, reexamination and systematic inspection may reveal unexpected inconsistencies and suggest new approaches. As a result, analysis frequently results in changes to existing processes.

Defining the best way to accomplish the desired results involves planning how to use FrameMaker—and any other relevant software—to meet the project goals. While this step includes planning how each structural element will be formatted in various contexts, it also must account for user interface aspects. Unfortunately, this process is frequently slighted in markup projects. As a result, SGML and to a lesser extent XML has acquired an undeserved reputation in some circles for being complex and difficult to use. In fact, poorly designed markup applications are often at fault.

Many things can be done to make markup easier and more appealing to the user. For example, FrameMaker menus can be customized, both by adding new application-specific commands defined through the FDK and by simplifying the menus by removing access to unneeded capabilities. In this part of the design phase, the project team must consider questions such as the following:

- What is the expected sequence of tasks that users will perform?
- Are there repeated steps that can be automated through the FDK?
- Will users work with complete documents or with fragments?
- Will all documents use the same formatting templates?

Implementing the application

Once the technical goals of the project have been determined, implementation begins. "Technical steps in FrameMaker application development" on page 67 describes the steps involved in implementing a FrameMaker application.

Testing

Although writing a DTD or a formatting specification does not use a traditional programming language, it is essentially a programming process. Therefore, even when there is no need for FDK clients, creating an markup application (for FrameMaker or any other tool) is a software development effort and, as such, requires testing. Two types of tests are needed: realistic tests of actual documents and tests of artificial documents constructed specifically to check as much of the application as possible.

All necessary processes must be tested: formatting, markup import, and markup export. Test documents must include FrameMaker documents to save as markup, and markup documents to open in FrameMaker. Testing must also include any processing of legacy documents, including the use of the FrameMaker conversion utility. Finally, once the application is used in production, continued testing should incorporate some actual production documents.

Training and support

Despite the intuitive nature of structured documents, end users cannot be expected to understand the intended use of different element types and attributes simply by working with them. They must be provided with training in the form of documentation, classes, or sample documents. Application-specific training can be combined with training on FrameMaker and other tools to be used in the project.

Provision must also be made for ongoing support. As they use the application, end users will occasionally have questions or encounter bugs. They may need to use structures that have not already been defined.

Maintenance

While the application development effort decreases over time, some maintenance should always be expected. In addition to needing bugs fixed, applications may be changed to accommodate new formatting, to encompass additional documents, and to track updates to the DTD.

The implementation team

So far, the different phases in the development of an markup application have been enumerated. Implementing those phases requires a team of people with different areas of expertise.

All large projects begin with an analysis phase. Participants must include both end users and developers, who will work on the eventual implementation. For an XML or SGML project, the end users include.

- Authors, editors, and graphic designers (production formatters) who will use the editing and publishing application
- Subject-matter experts familiar with the content requirements of the documents to be processed.

Developers include individuals with expertise in

- Markup
- FrameMaker
- XSLT
- Any other markup tools to be used
- Additional software tools, such as document management systems and database packages

Additional participants in the analysis may contribute significantly to the project definition even if they will neither develop nor use the completed application. For example, the organization's Web advocate may be able to identify some requirements that do not affect the rest of the team.

The skill sets required within the implementation team include document design, markup knowledge, setting up FrameMaker formatting templates, and setting up formatting rules that control automatic application of the desired graphic design to structured documents. If the FDK is used, programming skills are also needed. Of course, technical writing skills in documenting the finished application are a valuable contribution to the completed effort.

Technical steps in FrameMaker application development

This section enumerates the tasks involved in developing a FrameMaker application and defines the files containing modules of the application that the application developer maintains.

Element definition documents (EDD)

At the heart of every FrameMaker application is an element definition document (*EDD*). An EDD provides three types of information:

- The definitions of the elements that can occur in a document, including their allowable content and attributes. These definitions can be automatically extracted from a DTD or Schema.
- The formatting rules that define the visual characteristics of various document components.
- Other information governing behavior of elements, including rules for automatically inserting several elements in response to a single user action, preparing for the use of document fragments, and so forth.

An EDD is itself a structured document, and you use the structured editing features in creating and editing the EDD. You do not need not remember where to insert element definitions, context specifications, or format rules, because the Structure View and Element Catalog guide you in creating these constructs correctly.

The EDD formats the structure elements for readability: Different fields are clearly labeled, and spacing, indentation, and different fonts emphasize the organization. You can enhance the accessibility of information by grouping element definitions into sections and explaining them with extensive comments. These characteristics are illustrated by the following fragment:

Element Definition Document (EDD) for Reports
This EDD defines the structure rules for a report.

Element (Container): Report
Valid as the highest-level element.
General rule: (Title, Abstract, Contents, Chapter+, Appendix*)

Report and Chapter Structure
A report consists of Chapters and Appendices, preceded by a Title, Abstract, and Table of Contents. The Chapters and Appendices may be divided into Sections. Each Chapter, Appendix, and Section has a Head (or title).

Title of the entire report (the Head element is used for Chapter, Appendix, and Section titles).

Element (Container): Title
Valid as the highest-level element.
General rule: (<TEXT>)
Text format rules
1. **In all contexts.**
Basic properties
Alignment: Center
Line spacing
Height: 12pt
Line spacing is fixed.
Default font properties
Weight: Bold
Size: 36pt

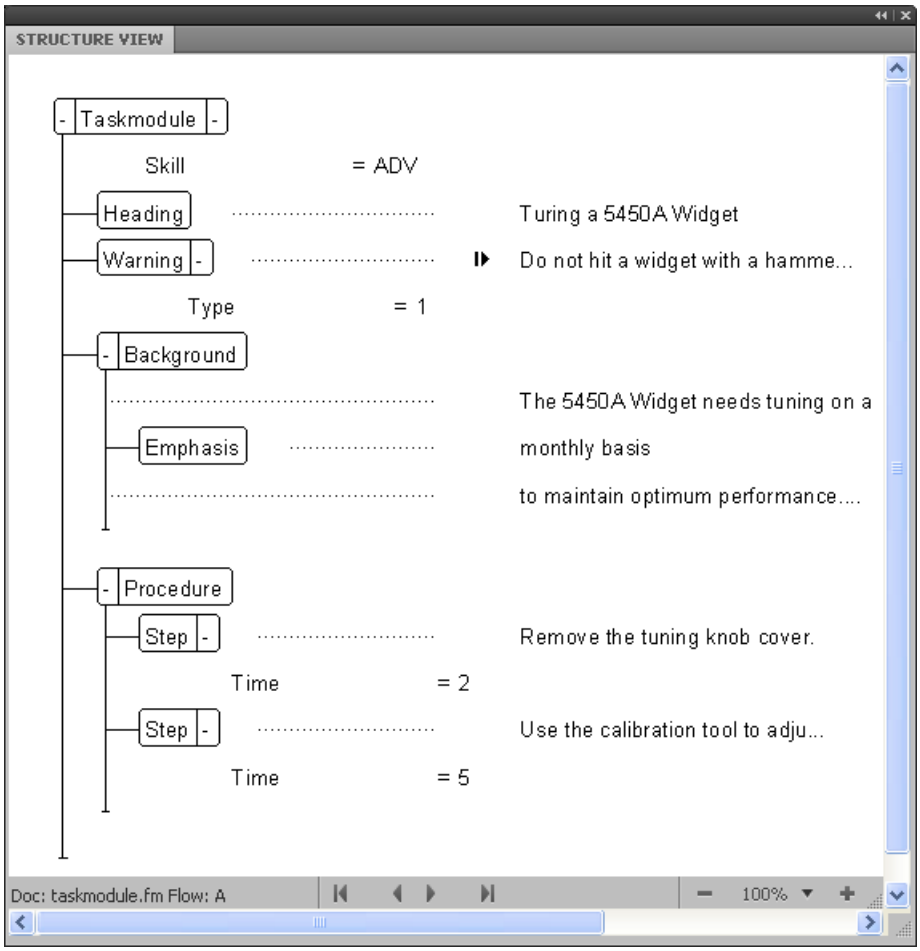
Structured templates

End users do not use EDDs directly. Instead, the definitions made in an EDD are extracted and stored in a template—that is, a FrameMaker document used as a starting point for creating other documents. Developers provide end users with a template that incorporates the structure and format rules from a particular EDD, as well as page layout information and formatting catalogs that define paragraph, character, cross-reference, and table styles. It may also contain sample text and usage instructions.

In some applications, information can be formatted in several ways. For example, the same text may need to be formatted for books with two page sizes. FrameMaker can accommodate such requirements. If an EDD's format rules refer to paragraph and character styles in the template's catalogs, new catalogs can be imported from another template to change the document's appearance. New page layouts, table styles, and cross-reference styles can also be imported. If the format rules incorporate specific formatting parameters, the appearance can be changed by importing a new EDD with different format rules.

Moving data between markup and FrameMaker

FrameMaker maintains the native element and attribute structure of markup in a *WYSIWYG* environment. The FrameMaker user can easily inspect this element structure in the Structure View. For the maintenance procedure example, the Structure View appears as follows:



Since both the FrameMaker rendition and the markup text file include content and element structure, data can move between the two forms automatically. The user interface is

straightforward. The FrameMaker **File > Open** command recognizes XML and SGML documents. When a user opens one, FrameMaker automatically converts the document instance to a structured *WYSIWYG* document and applies the format rules of the appropriate markup application. To write a structured *WYSIWYG* document to XML the user simply selects the **File > Save**, or **File > Save As XML...** command. For SGML, the user specifies SGML when executing the **File > Save As...** command.

Just as the form of a structured document parallels the form of a markup document instance, an EDD parallels a DTD or Schema. FrameMaker can also move definitions of possible structures between the two forms. Thus, if a new project is based on an existing XML DTD, FrameMaker can automatically create an EDD from the DTD. For example, given DTD declarations such as

```
<!ELEMENT step (#PCDATA)>
<!ATTLIST step time NUMBER #IMPLIED>
```

FrameMaker builds the following element definition:

Element (Container): Step
General rule: <TEXT>
Attribute list
1. **Name:** Time **Integer** **Optional**

Since there is no formatting information in the DTD, no format rules are included in the automatically generated EDD. You can manually edit this information into the EDD, or import a CSS style sheet into the EDD. You only need to do this once, even if the DTD is revised.

It is common for a DTD to undergo several revisions during its life cycle. When you receive an updated DTD, FrameMaker can automatically update an existing EDD to reflect the revision, and the update process preserves existing formatting information and comments in the EDD. The update incorporates changes to content models and attribute definitions, inserts new element types and removes discarded ones, then generates a short report summarizing the changes so that the application developer can review them.

If a project is not founded on an established DTD or Schema, the application developer can start implementation by creating an EDD. Once the EDD is finished, FrameMaker can automatically create the corresponding DTD.

Using XML Schema to create a DTD or EDD

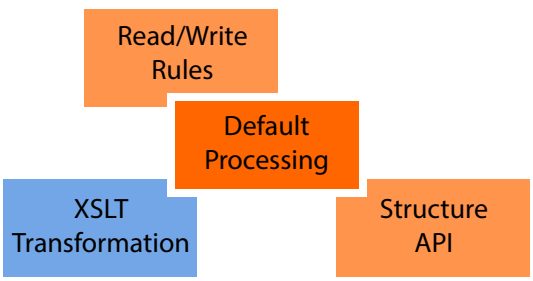
FrameMaker can create a DTD from the Schema, and the EDD from that DTD. This happens automatically when you import an XML file that uses Schema and does not refer to a DTD. You can also import a Schema directly, use it to create a DTD and EDD, and create a template or reference the resulting DTD in your XML documents. For example workflows, see Developer Reference, page 201: XML Schema to DTD Mapping.

Customizing markup import/export

You might want to customize the correspondence between a document's markup and FrameMaker representations for several reasons; for example:

- To base the FrameMaker application on an authoring version of an interchange DTD
- To integrate FrameMaker with database or document management tools
- To calculate portions of a document or display predefined text when certain elements or attribute values occur or particular entities are used
- To interpret certain elements, such as graphics or tables, as special objects

To support such requirements, FrameMaker offers three strategies for specifying how abstract markup structures are represented in the *WYSIWYG* publishing environment.

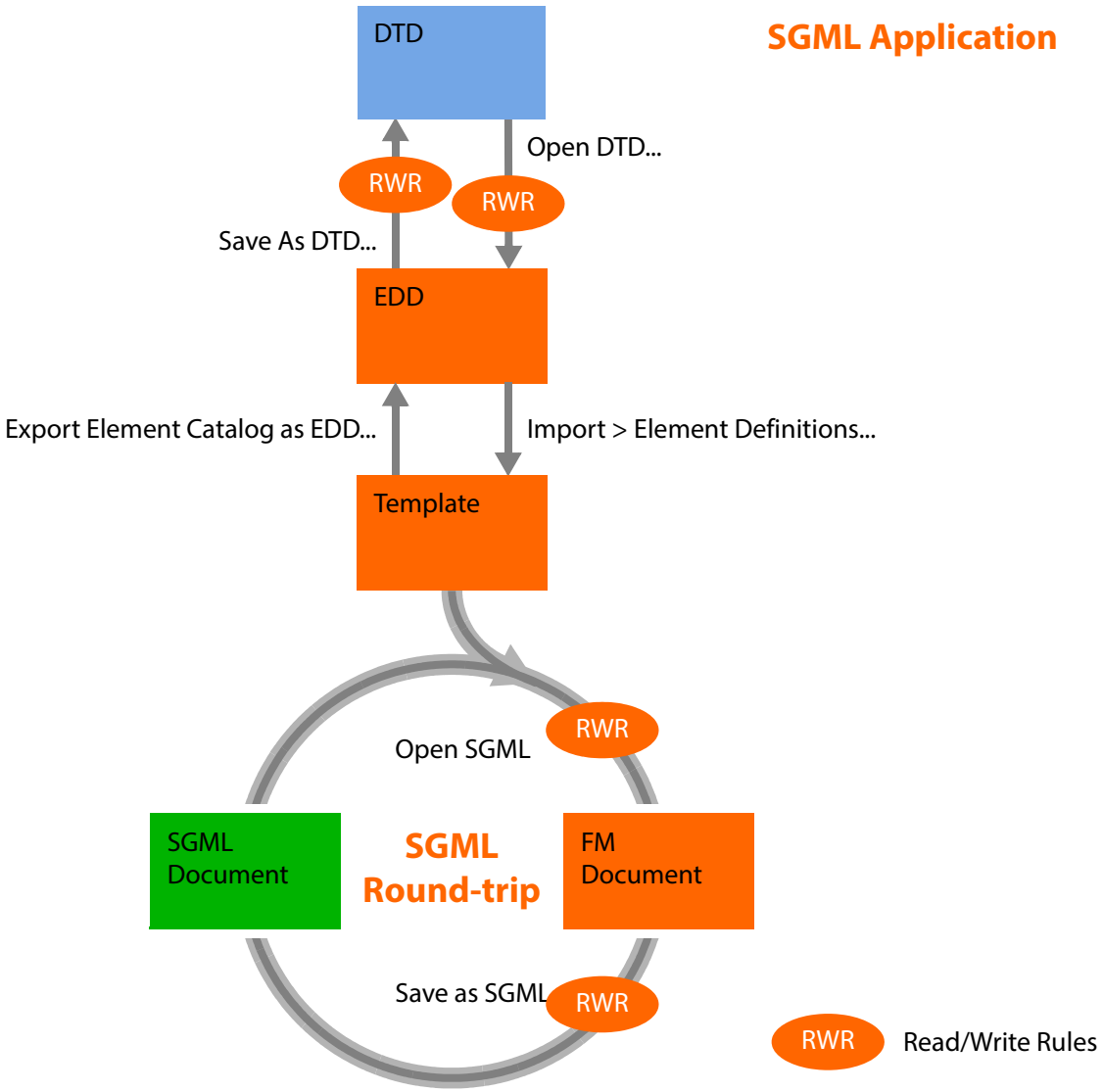


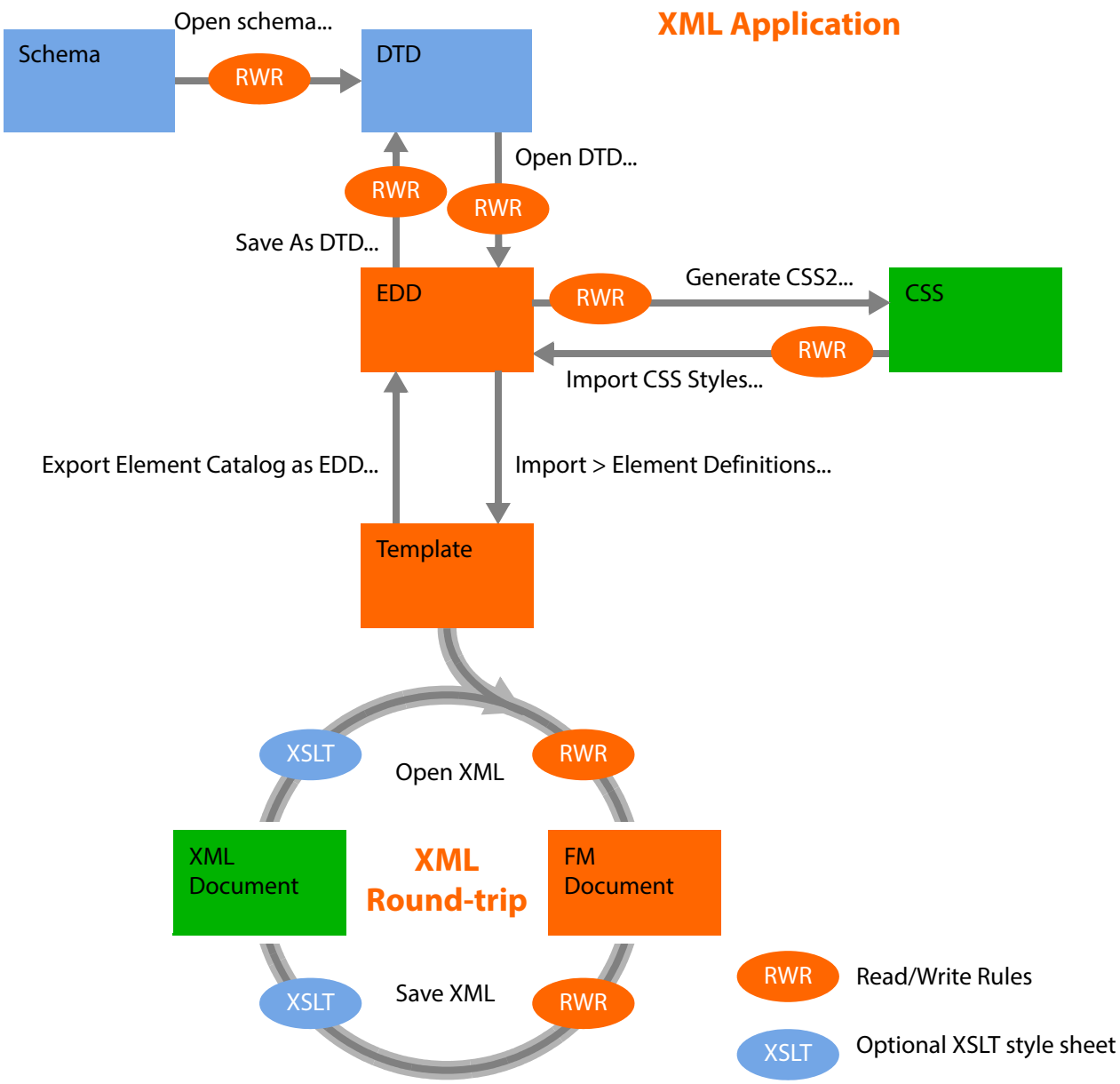
Most projects combine two or more of these approaches:

- By default, FrameMaker automatically translates a markup element into a FrameMaker element with the same name (and translates a FrameMaker element to a markup element with the same name); similar processing occurs for attributes. For many elements, the default processing is adequate.
- Some customizing can be accomplished through *read/write rules*. FrameMaker provides read/write rules for customizing that is common to many applications. For example, there are read/write rules for mapping markup elements into specific FrameMaker element objects such as table parts or graphics. The developer creates read/write rules in a separate file.
- For cases in which read/write rules are insufficient, the FDK includes a C function library called the Structure API, which enables more extensive customizing. The Structure API can be used, for instance, to extract part of a document's text from a database using attribute values as database keys.

- XSLT may be used to transform an XML structure into a structure that is more appropriate for editing or formatted output with FrameMaker. Export transformation converts the XML back into its interchange structure.

Read/write rules can be used with all four possible conversions—markup document to FrameMaker document, FrameMaker document to markup document, DTD to EDD, and EDD to DTD. These conversions are shaded in the following diagrams, which also show the FrameMaker commands that invoke them and their interaction with other data files.





Structure application files

Moving documents and type definitions between their markup and FrameMaker forms involves a number of files. The developer bundles the names of the files needed for a particular application—as well as other information—into a *structured application definition document*. Definitions for all markup applications available on a particular computer system are grouped in an application file. The FrameMaker configuration file, `structapps.fm`, is a structured

application definition document that FrameMaker reads whenever it starts. The following example is a fragment of a typical structure application file:

Application Definition Version <*version*>

Application name: Maintenance
- **DTD:** task.dtd
- **Read/write rules:** task.rw
- **SGML declaration:** namelen.big
- **DOCTYPE:** taskmodule

Application name: Report
- **Template:** report.tpl
- **Entity locations**
  - **Entity catalog file:** catalog

Because FrameMaker automatically reads the `structapps.fm` file, writers and editors do not need to know about the structure application file. To import or export markup documents, however, they may need to know the names of available applications—but even this information can be stored in a FrameMaker template or automatically selected from the document type name of an markup document. For example, given the preceding structure application file example, when reading an SGML document that begins:

```
<!DOCTYPE taskmodule SYSTEM "task.dtd">
```

FrameMaker uses the Maintenance application. Because that application specifies read/write rules, the indicated read/write rules control the import. Furthermore, the Maintenance application is stored in the resulting structured document so that the same read/write rules are used if the document is later exported back to markup.

The information in an application definition can include:

- One or more document type names that trigger use of the application on import.
- The name of the file containing the DTD to be used for export. This field is not needed for applications that only require import, since imported markup documents always include a document type declaration. When it does appear, the "DTD for export" field actually contains the name of a file containing DTD declarations (sometimes called an external DTD subset) suitable for use in document type declarations such as the one shown in the preceding example for the Maintenance application.
- Character encoding.
- The handling of conditional text for export to XML.
- Files containing external entities (identified by entity name or public identifier).
- An entity catalog.

- Search paths for subfiles of read/write rules and external entities.
- External cross reference handling for XML.
- Whether namespaces are enabled in an XML application
- The name of the file containing the application's read/write rules.
- An XML schema reference
- Identification of a structure API client to use during import or export.
- The handling of CSS style sheets for import and export.
- The names of XSLT style sheets for import pre-process and export post-process for XML transformation.
- The name of a file containing an SGML declaration to be used for export when no SGML declaration appears at the beginning of imported SGML documents.
- The name of a template to be used to create new documents during import. The template incorporates the element definitions imported from an EDD as well as page layout information and formatting catalogs. A template is needed only for applications that include markup import.
- XML Encoding for export or display

Legacy documents

In general, the conversion of unstructured documents to structured documents (or to markup documents) cannot be completely automated. FrameMaker offers a utility that adds structure to unstructured FrameMaker documents. This is done by mapping paragraph and character styles as well as other objects—such as footnotes, cross-references, markers, and tables—to elements. Some manual polishing of the resulting documents is expected; the amount of editing depends on the discipline with which formatting catalogs were used in the original and on whether a unique set of elements corresponds to each formatting tag. Since FrameMaker can read many common word processing formats, this utility can be used to structure word processor documents as well. Once any errors in the resulting structured document have been repaired, it can be exported to markup. Thus, FrameMaker provides a path for converting word processor documents to markup.

The rules for mapping between styles and elements are specified in a conversion table. Techniques for defining the mapping are provided in Developer Reference, Chapter 4, Conversion Tables for Adding Structure to Documents

Application delivery

Once an application has been completed, it must be delivered to the end users. The developer can either install the completed application on the end users' system or network, or provide instructions for doing so. At a minimum, end users need copies of all templates and FDK clients.

If end users will be structuring existing unstructured documents, they will also need copies of all conversion tables used. Finally, if end users will be exporting or importing markup documents, their structure application files must be modified for the new application. They need access to the read/write rules, entity catalog, DTD/Schema for export, XSLT style sheets, and (for SGML structure applications only) an SGML declaration.

Typical application development scenarios

Although creation of each component of a FrameMaker application is straightforward, novice developers may need some guidance as to the order in which to create these components. The strategy for creating an application can vary considerably with the requirements of a particular project. The following scenarios begin after the analysis and design phases have been completed. In all three, the developer implements a few elements at a time and stops to test the growing application frequently.

Starting from existing markup documents

In the first scenario, there is an existing DTD and sample markup documents. You select a short sample document and start development by supporting the element types that occur in it. Later, you add more complicated documents. Application development proceeds as follows:

1. Create a new file of read/write rules and enters rules that reflect any decisions made during the analysis phase of the project. In particular, enter rules to
   - –Identify elements used for graphics, tables and table components, and cross-references
   - –Discard elements and attributes that the organization will not use even if they are defined in the DTD
   - –Rename elements and attributes to reflect the organization's terminology
   - –Replace terse, abbreviated markup names with longer, more readable FrameMaker versions
   - –Specify FrameMaker names with multiple capital letters. For instance, you might map the element generic identifier `BLOCKQUOTE` to the FrameMaker element tag `BlockQuote`. Such rules will not be needed with SGML applications if the SGML declaration does not force general names to uppercase (that is, if case is significant in the SGML names).
2. Define a new markup application in the structure application file that invokes the read/write rules. If appropriate, the application definition specifies an XML or SGML application and maps any public identifiers used in the DTD to system filenames.
3. Using the new application, create an EDD from the DTD with the **StructureTools > Open DTD** command. The resulting element declarations may be grouped into sections, and some added comments in the process.
4. Open the selected sample document. The resulting FrameMaker document serves as the basis of a template. In this document, define page layouts, specifying the number of columns on the page, the width of any side-heads, page numbers, and so forth.

5. Return to the EDD and begin to define formatting rules. In general, formatting specifications are unnecessary for higher level elements, such as those for chapters and sections. Concentrate on elements that contain text and their immediate ancestors, such as elements for list items, notes, cautions, emphasized phrases, and cited titles.

6. After providing format rules for a few elements, test them by using the **File > Import > Element Definitions** command to import the element definitions from the EDD into the sample document. This command reports any formatting catalog entries mentioned in format rules but not defined in the sample FrameMaker document. Define all reported formats and glance through the sample document to verify that the formatted elements appear as intended.

   You can then return to the EDD, correcting any errors in the format rules and adding some more before testing them again, until the sample document is completely formatted. At that point you might want to test some other documents. After creating a base for further development, continue systematically through the DTD, making and testing context-sensitive format rules for all necessary elements.

7. You might need to add more read/write rules. If you change additional element tags, the original names must be replaced by the new versions throughout the EDD, in the definitions of the changed elements as well as in definitions that refer to those elements. To update the names in the EDD, use the **StructureTools > Import DTD** command, which reprocesses the DTD according to the current read/write rules and updates the existing EDD. It writes a report listing the changes made.

8. To create a template from the sample document, you can delete all content and save the result.

9. Use the template to test the editing environment by adding automatic insertion rules to the EDD, specifying, for example, that when the end user inserts a `List` element, FrameMaker should automatically insert an `Item` element within it. Consider whether an FDK client can provide input accelerators, such as automating frequent sequences of editing steps. You might also want to customize the FrameMaker menus for end users, perhaps removing developer-oriented commands or (to prevent the end user from overriding automatic format rules) removing formatting commands.

10. For markup import, add the template to the application definition in the structure application file, and add read/write rules for special characters and other entities. Once again, open the markup version of the sample document, test the result, and modify the application until the results are correct.

11. If read/write rules are insufficient to import any elements, use the structure API to develop a client with the requisite functionality.

12. If markup export is also required, test it and add other read/write rules or structure API functions, if necessary.

13. Once formatting, import, export, and editing functions have been tested, deliver the completed application, along with documentation, to the end user.

Working with Schema

You can import an XML document that references a Schema file, and you can specify a Schema file in your structure application, to use for validating a document upon export to XML.

1.For a specific XML document, you can include the path of the schema file in the XML using attributes - `noNamespaceSchemaLocation` or `schemaLocation` depending on whether your schema includes a target namespace or not.

2.To specify a Schema file for use in exporting XML, modify the `structapps.fm` file. Use the Schema element as part of the `XMLApplication` to provide the Schema file path for export.

3.Open the XML in Frame using a structured application. Edit it.

4.Save the XML using a structured application. The Schema element in the structapps.fm file is output in the file and validation is performed against it.

In this workflow, a DTD is generated automatically as an intermediary file from the Schema given in the XML document, and you do not modify it. However, you can also use a Schema file to generate an EDD directly, or you can modify the DTD and reference it from the XML document. When an XML document references both a Schema and a DTD, FrameMaker imports it using the DTD, although it still validates against the Schema.

Creating a DTD or EDD from Schema

You can create a new EDD from a Schema definition, or import the elements from a Schema definition into an existing EDD. FrameMaker converts the Schema definition to DTD first, and then creates or imports elements to an EDD. To do this, use these commands in the **StructureTools** menu:

- Open Schema: This command converts a specified Schema to DTD, and creates a new EDD from the DTD.
- Import Schema: This command converts a specified Schema to DTD, and imports elements from the DTD into an existing EDD.

Each command allows you to specify the Schema file, then asks you where to save the resulting DTD file. To create an EDD from Schema:

1.In Structured FrameMaker, click **Structure Tools > Open Schema**. Choose a Schema file.

2.Choose a path for the DTD to be output.

3.Examine the resulting DTD and make any modification you wish.

4.Create an EDD from the generated DTD.

5.Use this EDD to create a template that can be included in the Structured Application

6.Provide your DTD path along with the Schema Location in the input XML. This will make sure that FrameMaker works correctly with your template. Validation of input and output XML is still performed against the Schema.

Building a new application

Another typical use of FrameMaker involves developing the document type definition from scratch. In this scenario, FrameMaker is used to write documents that are then exported to markup. Again, development work relies on a thorough data analysis.

1. Instead of starting with a DTD, begin to develop an EDD. Guided Editing makes the mechanics of editing the EDD simple. You can choose to define both content rules and attribute definitions for one element at a time, or to complete all content rules before defining any attributes. You can also define content rules for numerous elements before adding any format rules.
2. Create a template and then use the **File > Import > Element Definitions...** command to interpret the EDD and store the resulting element catalog in the template.
3. Use the template to test sample documents. As in the scenario of the previous section, you may find it easier to keep track of the work by repeating the edit-import-test cycle a few elements at a time.
4. When the EDD is complete, use the **StructureTools > Save As DTD** command to create an equivalent DTD.
5. Create a structure application that uses the generated DTD for export.
6. Test the export of the sample document and add read/write rules and a structure API client as necessary. If you do not want to use the FrameMaker default representation for special object elements, edit the corresponding element and attribute definition list declarations, in addition to providing the associated rules.
7. You can edit the generated DTD by allowing for tag omission and providing appropriate short reference mapping, short reference use, and entity declarations. Since FrameMaker exports all comments in the EDD to the generated DTD, review comments carefully. Retain comments pertaining to the structures being defined, but discard comments specific to FrameMaker formatting, since they do not necessarily apply to all uses of the DTD.
8. If the application requires markup import as well as export, test conversion in the import direction as well, modifying the application to correct any errors.
9. Finally, deliver the application to end users and prepare for the project's maintenance phase.

Working with legacy documents

The final scenario assumes there are existing unstructured FrameMaker documents. The developer can take either of two general approaches: constructing a conversion table to structure the legacy documents and using the resulting structure to start an EDD; or creating an EDD based on an analysis of the legacy documents and then creating a conversion table to reflect the element definitions in the EDD. The following steps outline the first approach:

1. Select a typical unstructured document and use the **StructureTools > Generate Conversion Table** command to create a new conversion table based on the format tags of the

unstructured document. This command puts a row in the table for each type of FrameMaker object and format tag used in the sample document. The initial conversion table might appear as follows:

Wrap this object or objects	**In this element**
P:MainTitle	MainTitle
ChapterTitle	P:ChapterTitle
P:Body	Body
C:Emphasis	Emphasis
G:	GRAPHIC
F:flow	FOOTNOTE
T:Table	Table

2. Edit the conversion table, changing entries in the second column, adding entries in the third column, and adding new rows for higher level structures.

3. Use the conversion table to add structure to the sample document, ignoring the lack of formatting in the resulting structured document but correcting as many structural errors as possible by editing the conversion table and reapplying it.

4. With the **StructureTools > Export Element Catalog as EDD** command, extract an EDD that has an element definition for each element type that appears in the sample document. Although these element definitions define the content of the elements very generically—each is allowed to contain text as well as any defined element—the EDD provides a skeleton that you can further develop.

5. The exported EDD lists the element definitions alphabetically, according to their tags. You can rearrange them into a logical order if desired—easily moving them around using drag-and-drop in the Structure View—and then replace the content rules with more restrictive ones and add format rules.

6. The sample document (and other automatically structured documents) can be reformatted by using the **File > Import > Element Definitions** command from the EDD into the generated structured document

7. Continue to finish the application, as in the previous scenario. However, since page layouts and formatting catalogs were defined in the original unstructured document, the template development effort does not need to be repeated.

5 Structure Application Basics

This chapter provides an introduction to structure application development using FrameMaker. The chapter examines some of the reasons you might be using markup data and FrameMaker together and explains how those reasons affect the structure application you develop. The chapter also provides high-level information about creating that application.

Where to begin

If your end users do not need markup, but use structured documents only because they want the benefits inherent in a structured authoring environment, you only need to provide them an *element definition document (EDD)* and a structured template. In this case, the material you need is contained in Part III, "Working with an EDD."

If your end users need to read or write markup documents, you will probably need to develop a structure application to modify the default translation between markup data and FrameMaker documents. In that case, this entire manual is of use to you. The rest of Part I provides general information on creating a structure application. Part IV, "Translating Between Markup Data and FrameMaker," provides details of the default translation between markup data and FrameMaker and explains how you can change this default behavior with structure read/write rules.

Structure application scenarios

The specific features of your structure application will largely depend on how your organization intends to use it.

Translating in one or two directions?

You can write structure applications that translate documents in one direction—from XML or SGML to FrameMaker or from FrameMaker to XML or SGML. Or your application can translate in both directions, enabling information to make round trips between markup data and FrameMaker.

If your end users only translate the data in one direction, you can limit your structure application to only manage the translation in that direction. One-direction translations are:

- Reading markup data into FrameMaker—the structure application only needs to import specific doctypes into FrameMaker
- Writing FrameMaker documents as markup data—the structure application only needs to export FrameMaker documents as XML or SGML of a specific doctype

For example, your company may have a large XML document database of parts information from which it periodically needs to produce catalogs. You may want to deliver the information in FrameMaker to take advantage of its formatting capabilities while you continue to create and store the information only in XML. In this case, end users manipulate the catalog document in FrameMaker, but not the source XML database. Because of this, you set up an application to translate from XML to FrameMaker but don't worry about the reverse.

Or assume your company needs to publish documents in print, PDF, and XML. The authors can create the documentation in FrameMaker, paying attention to pagination and other formatting issues as they go. When they have a final version of the documents they can save them as PDF and XML, and the online documentation will be made from the same source as the printed documentation.

Or perhaps your end users work with FrameMaker to create their documents, but they collaborate with other writers who use a different authoring environment for XML documents. In this situation your application needs both to read XML documents into FrameMaker and to write FrameMaker to XML, so that your end users can collaborate effectively.

Can you simplify when translating in only one direction?

If your application only needs to translate in one direction, then you might be able to simplify some of the information you present. For instance, if your end users have existing FrameMaker documents to deliver in markup, they developed documents using the full power of FrameMaker. They may well have used such FrameMaker features as markers to facilitate document creation. But some of these features may not be relevant in markup data, so you can choose to omit them from the markup document. If you won't be re-importing the markup data back into FrameMaker, the resultant loss of information is not important. Therefore, you don't have to retain the use of those features in your structure application.

Do you have an existing element definition?

Another factor influencing the design of your application is the degree to which the document structure has already been defined for the markup data and the FrameMaker template. Is your starting point an established EDD, a DTD—for instance, one of the Department of Defense *CALS* DTDs—or an XML Schema? Will you be provided with both an established EDD and an existing body of documents that use that EDD? Or has no definition document as yet been written?

It is common to start application development with a DTD. In that case, your structure application must also address the issues of translating the DTD to an EDD. If you start with an existing XML Schema, you must create a DTD from it, and can then continue to create the EDD from the DTD. It is also possible that you begin with an existing EDD, and must translate the EDD into a DTD.

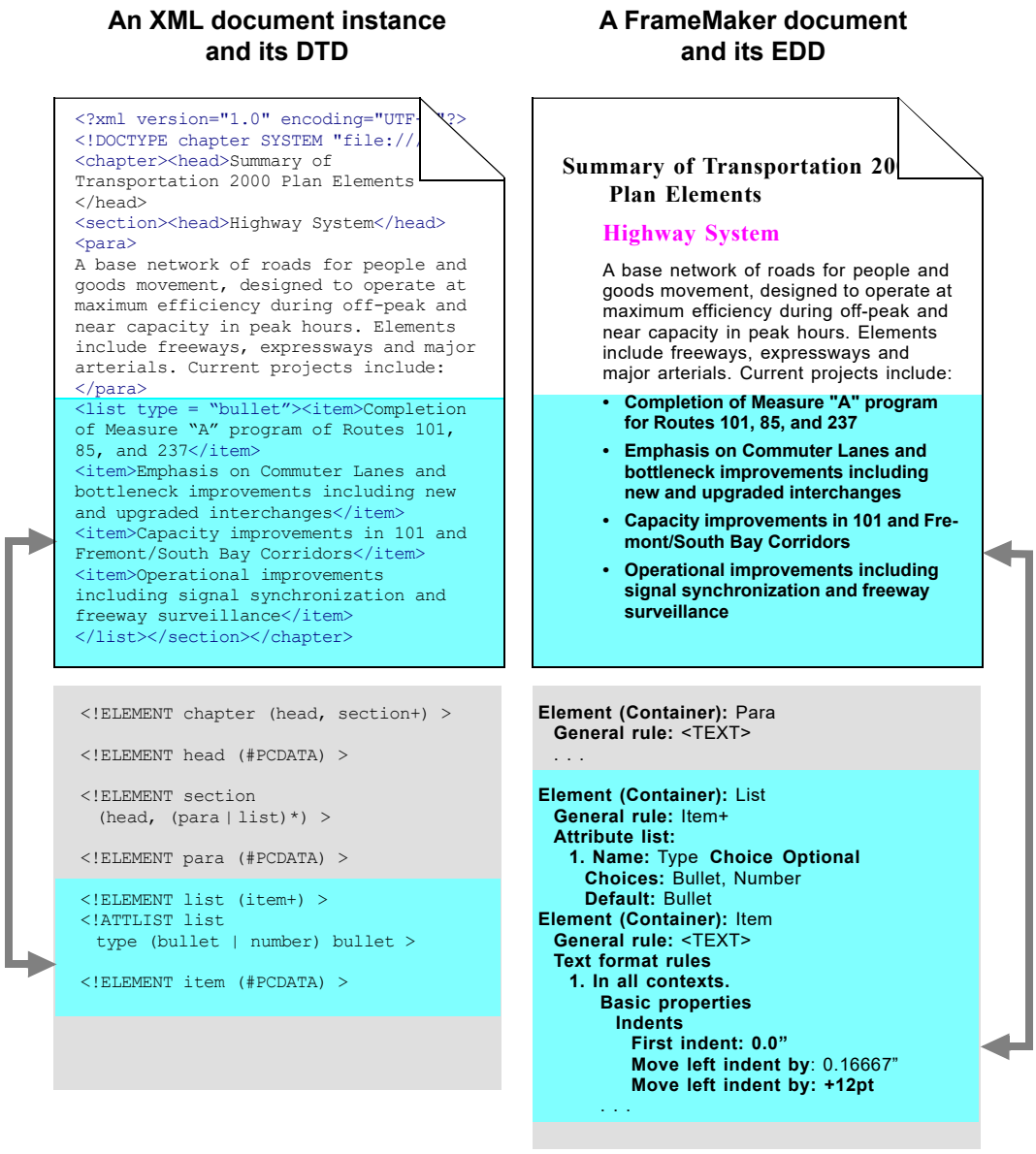
For example, assume your end users already have a library of documents based on a particular DTD. In that case, you'll be less inclined to make changes to the DTD that require modifying those documents. If you start without a model on either side, however, you have the freedom to design

the EDD and the DTD with its counterpart in mind. If you have this freedom, the application design process may be easier.

Structure application development

Markup data and structure in FrameMaker have many similarities. To describe the hierarchical nature of documents, both use special methods to define the elements that can occur in a *document instance*, and the relationships among those elements. In markup, these elements are defined in a DTD, and in FrameMaker they are defined in an EDD.

Within particular document instances, both use descriptive tags to identify each instance of a structural element. The tags in the document content correlate to the element definitions found in the DTD or the EDD, as shown in the following illustration. The top of the illustration shows pages of XML and FrameMaker documents, while the bottom shows a DTD and EDD, respectively. The highlighted portion of the XML document corresponds to the highlighted definition for `<list>` in the corresponding DTD. The FrameMaker page and EDD illustrate a similar correspondence.



Despite the similarities in their basic approach to the creation of structured documents, XML and SGML differ from FrameMaker in their methods of representing document content, and in the aspects of documents they represent. In this respect, markup and FrameMaker are like ordinary human languages that evolve from different cultures—they differ not only in the particular words they use, but in their rules for putting words together, and even in the thoughts that can be easily expressed with them. For documents to move between FrameMaker and markup data, the software needs instructions to help it translate between the two languages.

The starting point: an EDD, DTD, or Schema

Your end users need a starting point that describes the structure of documents they'll create or edit. You provide this starting point either with a DTD or XML Schema, or with a FrameMaker EDD.

If you start with a DTD, FrameMaker can create portions of a corresponding EDD for you. If you start with Schema, FrameMaker converts it to DTD first, then you can continue with creating the EDD.

If you start with an EDD, FrameMaker can create a DTD for you.

If a starting point has not yet been established, you can create your own DTD or EDD. In this situation, we recommend you to start by using FrameMaker's tools for creating an EDD (described in Part II of this manual) and then create a DTD from that EDD.

Translation between DTDs and EDDs

FrameMaker can automatically translate a DTD or EDD. If you start with a DTD, the software creates a default version of the corresponding EDD. Similarly, if you start with an EDD, FrameMaker creates a default version of the DTD.

Once your application is complete, your end users use standard commands to save individual FrameMaker documents as markup documents or to open individual markup documents in FrameMaker. In either case, the structure application works transparently with FrameMaker, making these automatic translations possible.

Formatting information in FrameMaker

You create a structured template to provide appropriate formatting for your documents. A DTD does not provide any formatting information. When reading markup data, FrameMaker combines any formatting that is defined in the document template with any formatting rules that are specified in the EDD. To vary the formatting of your documents, you may create more than one structured template for the same DTD. For example, your end users can use the same markup data for different purposes, such as a fully designed catalog and a brief parts list. They may want the same data formatted differently in these two situations.

With XML, formatting information can be saved as CSS, and further transformation can be specified in XSL.

- When reading XML, FrameMaker does not use any of the CSS formatting information, but you can export format information to CSS when exporting a document to XML. See Chapter 20, "Saving EDD Formatting Information as a CSS Stylesheet".
- You can apply XSL transformations to XML before importing the XML document into FrameMaker, and after exporting a document to XML. See Chapter 30, "Additional XSL Transformation for XML".

Changing the default translation

The structure of markup data varies widely, and that can affect the way it maps to FrameMaker document objects. As a result, much of your job in creating a structure application is to change the default translation FrameMaker uses. You do so by providing information the software needs to recognize and process particular constructs. If FrameMaker automatically translates all the components of a document just as you want, you don't need to provide this extra information.

One of the differences between markup data and FrameMaker is that has explicit representations for items such as tables or graphics. With XML or SGML, an individual DTD defines arbitrary representations for tables and graphics. For this reason, the default translation in FrameMaker assume certain DTD representations of these items. For example, by default FrameMaker assumes a CALS/OASIS model for tables. These assumptions may or may not match your actual DTD. If they don't match, you need to develop an application that modifies the default translation accordingly.

How you modify the translation

To modify how FrameMaker translates documents, you start by using special *read/write rules*. In many cases these rules are sufficient to successfully translate a given doctype. A significant part of your effort in creating a structure application consists of specifying these rules.

You specify read/write rules in a special rules document. Most rules are relevant to all import or export functions, but a small number apply only to some. For example, there is a rule to rename an element on import and export. There is also a rule to tell the software how to treat FrameMaker non-element markers on export, but this rule doesn't apply to import or to creation of a DTD.

In some situations the representation you want in your DTD and your EDD may be radically different from what FrameMaker does by default. If so, rules might not be adequate to correctly translate your documents. For example, the FrameMaker model for tables assumes that a table is described in row-major order. But if your DTD describes tables in column-major order, then the FrameMaker read/write rules won't be able to translate between the markup data and FrameMaker representations. In other situations, your XML or SGML document may contain *processing instructions* that need to be interpreted but are unknown to FrameMaker. In such situations, you can customize FrameMaker via the FDK.

The FDK allows you to modify FrameMaker's import and export of individual markup and FrameMaker documents. You cannot use it to modify the creation of an EDD or a DTD.

What your end users do

Once you've created a structure application, your end users use that application to open markup documents in FrameMaker and to save FrameMaker documents as markup data. For this purpose, they'll work in a FrameMaker environment that includes your structure application. Depending on your application, the fact that files are stored as markup or as FrameMaker documents can be invisible to your end users. For example, the standard Open command opens either an XML or a FrameMaker document.

6 A General Comparison of Markup and FrameMaker Documents

Markup data and FrameMaker documents use models for structured documents that are sometimes similar and sometimes substantially different. This chapter describes the similarities and differences that you should understand before you can create a structure application.

Structure descriptions

In markup data, elements are defined in *element declarations* and *attribute definition list declarations* within a *document type definition (DTD).* In FrameMaker elements are defined in *element definitions* within an *element definition document (EDD)* and attributes are defined as part of an element.

FrameMaker EDDs

In FrameMaker, the EDD in which you create element definitions is a separate document. After creating the file of element definitions, you import the definitions into a template. A *template* is a FrameMaker document that stores common pre-defined information for a set of documents. This information includes element definitions and many other details such as paragraph formats, table formats, page layouts, and variable definitions.

The process of importing the EDD into a template stores the element definitions in the template's *Element Catalog*. After the EDD designer has imported the EDD, it is no longer needed as a separate document. Typically, the EDD designer retains the separate EDD document for maintenance. End users such as writers or editors, however, don't need direct access to the EDD. Instead, they work exclusively with the template file.

For detailed information on creating EDDs, see Part III, "Working with an EDD."

XML and SGML DTDs

For markup data, the process of providing declarations for a document is somewhat different. The DTD designer can create the declarations in a separate file. However, there is no step of transforming a DTD for use by a particular document. The form of the DTD remains constant.

In addition, a DTD is only concerned with the syntax of a document—that is, with legal ways to put together the pieces of a document, regardless of the intended purpose of the pieces. Markup has nothing to say about the semantics of the process, that is, the meaning of those pieces.

Before its document instance, a markup document always includes either all the declarations or a reference to them. Rather than requiring that all of the declarations in a DTD be included in each

markup document, the XML and SGML standards allow for the declarations to be stored separately and referenced from the document. The document type declaration in such a document includes the set of declarations stored separately by referring to them as an external entity.

A typical document type declaration has the form:

```
<!DOCTYPE name extid [ internal_declarations ] >
```

For XML, a document type declaration can have the form:

```
<!DOCTYPE name extid URL [ internal_declarations ] >
```

In these examples, *name* is the document type name and *extid* is the external identifier of an entity that consists of a list of declarations. For XML, *URL* refers to the location of the external declarations on the internet.

> **Important:** When reading and writing XML data, FrameMaker ignores the URL statement in the document type declaration. In FrameMaker, the structure application can include a specific DTD, or it can use a map to resolve the public identifier for the DTD on your system.

The declarations in the external entity are treated as though they appeared at the end of *internal_declarations*. The declarations that actually appear in *internal_declarations* are read before the ones in the external entity. Together, the declarations in the external entity and in *internal_declarations* are the *document type declaration subset* of the DTD.

There is an informal practice in the SGML community of using the term *external DTD subset* to refer to this external entity and using the term *internal DTD subset* to refer to the declarations shown here as *internal_declarations*.

In most places in this manual that use the term DTD, it can refer to either a complete DTD or to an external DTD subset. In the few places where the distinction matters, the manual clarifies which one is meant.

For XML, declarations can be specified in W3C's XML Schema. FrameMaker supports Schema by mapping it to DTD, from which the definitions are mapped into EDD. Detail of the mapping are provided in the Developer Reference, page 201: XML Schema to DTD Mapping

Elements

In both markup data and in FrameMaker, the basic building blocks of documents are *elements*. Elements hold pieces of a document's content (its text, graphics, and so on) and together make up the document's structure. FrameMaker distinguishes among several specific element types, corresponding to special FrameMaker document objects such as tables or cross-references. Markup does not make such a distinction.

FrameMaker element types

A large proportion of the elements in a structured FrameMaker document are defined to include text, child elements, or both. FrameMaker has several additional element types that represent constructs in a document for which the software has special authoring tools. These element types are for

- footnotes
- cross-references
- markers
- system variables
- equations
- MathML equations
- graphics
- tables
- table parts (such as table footings)
- rubi and rubi groups

If the element does not correspond to one of these constructs for which FrameMaker has special tools, the element is tagged as a *container* in the EDD.

FrameMaker provides flexibility in handling markers and system variables. As noted above, you can define elements that correspond to them. Alternatively, you can allow users to create *non-element* markers or system variables directly in an element that can contain text. In this case, you do not create a corresponding element. You choose the appropriate representation depending on your end-user's needs. For example, if your EDD includes a definition for a marker element, then when an end user inserts an element of this type in a document, FrameMaker uses the marker element's format rules to insert a marker of the correct type. When an end user inserts a non-element marker, on the other hand, the user must explicitly specify the marker type.

- For more information on element types, see Chapter 12, "Developing an Element Definition Document (EDD)."
- For information on translating between FrameMaker and markup elements of various types, see Part IV, "Translating Between Markup Data and FrameMaker."

XML and SGML elements

Markup data provides even more flexibility than does FrameMaker. Markup doesn't dictate the purpose of elements of any type. A construct that is represented in FrameMaker by a marker or system variable element can be represented by an element with a declared content of empty in markup. It can also be represented by a completely different element structure.

Element declarations and definitions

Each element definition or declaration in both markup data and in FrameMaker contains the element's name and information about appropriate content for that element. FrameMaker element definitions also contain appropriate formatting for the content. In markup data, an element's name is its *generic identifier*; in FrameMaker, the name is its *element tag*. The information about content is in the *declared content* or in the *content model* of a markup element declaration and in the *general rule* of a FrameMaker element definition. Formatting information is specified via the *format rules* of a FrameMaker element definition.

Element content

The content model or declared content of a markup element and the element tag and general rule of a FrameMaker element specify in precise terms what an element can contain. In FrameMaker, the part of the content rule that specifies the basic element content is the *general rule*.

A FrameMaker element's general rule can be the single specifier `<EMPTY>` to indicate no content, or a combination of text and child elements. Some of the special element types, such as cross-references, do not have a separate content rule, since an element of that type always has the same content (a single cross-reference in this case). A markup element's content model can include either the reserved name `ANY` ("anything") or a combination of child elements. In some element definitions in both markup and FrameMaker, the content rule also specifies what content is required or optional and the required order of the content.

Inclusions and exclusions

> **Note: XML:** The XML specification does not allow inclusions or exclusions.

Definitions and declarations can specify elements with constraints other than their general rule. An element that is an *inclusion* in another element can occur anywhere within that element, without the general rule explicitly listing the included element. An element that is an *exclusion* to another element cannot occur anywhere within that element, even if the content rule implies that the excluded element is allowed.

For example, you can define an element representing a chapter as a heading followed by a combination of paragraphs and lists, each of which has its own structure. If you want to allow footnotes anywhere in a chapter, you could specify the element representing a footnote element as an inclusion in a chapter element. You then would not need to explicitly mention a footnote element in the definitions of paragraphs or lists, but would be able to include footnotes in those elements.

Attributes

FrameMaker and markup both provide *attributes* to supply additional information about an element. For example, the DTD designer for a manual could use an attribute called `version` for its `book` element to allow the user to specify a book's revision status.

In FrameMaker, the attributes for an element are a part of the definition of the element itself. In XML or SGML, the attributes for an element occur separately in an *attribute definition list declaration (ATTLIST)* in the DTD.

By default, FrameMaker translates most attributes in markup data as attributes of the same name in the FrameMaker document. However, you may decide to supply rules to change this behavior. Some attributes in markup data represent information best represented by other means in FrameMaker. For example, you can write a rule to specify that an attribute corresponds to a particular property, such as the number of columns in a table, instead of to a FrameMaker attribute.

- For information on defining attributes in an EDD, see Chapter 14, "Attribute Definitions," in this manual.
- For information on translating between FrameMaker and markup attributes, see Chapter 21, "Translating Elements and Their Attributes."

Entities

In markup, an *entity* is a collection of characters you reference as a unit. Entities are used for many purposes. You can use an entity as

- shorthand for a frequently used phrase
- a placeholder for an external file containing a graphic in some special format
- a way to include multiple physical files in the same document.

FrameMaker provides several features for handling situations for which you use entities in markup data.

Entities can be classified in various ways. For example, they can be grouped into general or parameter entities or into internal or external entities. *General entities* usually occur inside a document instance. *Parameter entities* usually occur inside a DTD. *Internal entities* have replacement text specified directly in an entity declaration. *External entities* are stored in a separate storage object (such as a data file) often identified in the entity declaration by a system identifier, public identifier, or both.

While FrameMaker doesn't have a single construct called an entity, it uses various mechanisms to provide the functionality of many XML and SGML entities. Entity-like features include the following:

Text substitution You can represent frequently repeated sequences of characters as general entities in markup, and as variables or text insets in FrameMaker.

File management You can break large documents across multiple files and manage those files with a single document containing general entity references in markup, and with a book file in FrameMaker.

Graphics In markup data you often store graphics in separate files, then include them in the document with general entity name attributes. In FrameMaker, you can store graphics and equations externally or internally.

Special characters SGML doesn't allow you to enter certain characters or sequences of characters directly as data characters in an SGML document. This can happen, for example, if the character is not in the character set of the SGML document. In FrameMaker these might be either variables or characters in a particular font.

> **Note: XML:** The XML specification allows the use of UNICODE text, which makes this use of entities largely unnecessary unless required by a predefined DTD. The specification also identifies predefined character entities which FrameMaker can translate by default.

Markup In SGML, entities may contain actual markup. Because FrameMaker is a WYSIWYG tool, it has no concept of markup as such.

- For information on creating variables, text insets, and books, see the *Using FrameMaker*
- For information on translating entities to various FrameMaker constructs, see:
  - –Chapter 22, "Translating Entities and Processing Instructions,"
  - –Chapter 24, "Translating Graphics and Equations,"
  - –Chapter 26, "Translating Variables and System Variable Elements."
  - –Chapter 29, "Processing Multiple Files as Books."

Documents

The XML and SGML specifications use the term *document* differently than it is used in FrameMaker. A markup document has a particular set of parts in a particular order and can be spread across multiple physical files. A FrameMaker document is simply a single file in FrameMaker format.

Markup documents

According to the XML and SGML standards, an XML or SGML document contains an *SGML declaration* (SGML-only), a *prolog*, and a *document instance set*. (For more information about the SGML declaration, see Chapter 9, "The SGML and FrameMaker Models.")

The prolog and document instance set allowed with FrameMaker have the simplest form defined in the standard. For a document used with FrameMaker, its prolog can contain comments, processing instructions, and exactly one DTD. Its document instance set includes exactly one

document instance. A document instance is a particular collection of data and markup such as a memo or book—what most users informally think of as a document.

When you open the file or files comprising an XML or SGML document, you can clearly see the parts of the document corresponding to the SGML declaration, DTD, and document instance. Frequently, the bulk of the DTD actually resides in a separate file as an external DTD subset and is referenced in the document.

FrameMaker documents

Since FrameMaker is a WYSIWYG tool, a FrameMaker document is organized differently than a markup document. A FrameMaker document contains information specified in the template from which it was created, along with the data of the document. The template information is stored in various *catalogs*, such as the *Element Catalog* and the *Paragraph Catalog*, and in special nonprinting master and reference pages. Rather than having explicit markup that appears in the document, it uses commands for adding structure and formatting that take effect immediately.

Multiple-file documents

Frequently, your end user wants to divide the text for a document into multiple physical files. For example, a book may have a separate file for each chapter. Both markup documents and FrameMaker allow a single conceptual document to be divided into multiple physical files.

FrameMaker provides the *book* mechanism for this purpose. A book file contains a list of files that together form a complete work. Each file in the book is a complete FrameMaker document and can stand on its own.

In markup, you can use text entities for the same purpose—you can have a single file with references to text entities, each of which contains a portion of the document. In markup, each text entity isn't a complete document. That is, each entity doesn't have its own DTD, and document instance. Instead, the text entities are part of the document instance of a single markup document.

- For information on creating FrameMaker books, see the FrameMaker user guides.
- For information on creating books from text entities, see Chapter 29, "Processing Multiple Files as Books."

Format rules

Markup has no standard mechanism for representing the formatting of a document, although some DTDs use attributes for some formatting information. XML uses CSS and XSL to represent formatting.

FrameMaker provides format rules that allow an EDD to store context-sensitive formatting information. When you read an XML document into FrameMaker, the formatting specified in the

XSL file has no effect. However the CSS formatting can be used if required, otherwise FrameMaker uses the formatting specified in the EDD.

FrameMaker also uses the EDD format rules when you edit a document. As you insert an element with format rules, FrameMaker applies the appropriate format to the element's content on the basis of surrounding structure and attribute values. That is, FrameMaker can format the same element in different ways, in different contexts in a document. In addition, an end user can override formats for any portion of a document.

FrameMaker format rules can be defined hierarchically. For example, you can say that the font family and size for `Section` elements are Times 12pt and for `MyTab` table elements they are Helvetica 12pt. Later, you can say that the `Fnote` footnote element is 9pt. Since you did not specify the font family for `Fnote`, it is Times if it occurs in a `Section` element, but Helvetica if it occurs in a `MyTab` element.

- For information on creating format rules in an EDD, see Chapter 15, "Text Format Rules for Containers, Tables, and Footnotes," and Chapter 16, "Object Format Rules."

Graphics

There is no standard mechanism for representing graphics in markup data. There are several common methods in use, in each of which an entity holds the graphic object itself.

In markup, the entity can be in an external file, written in a particular format such as Sun™ raster format. The graphic data format is given a name called a *data content notation*. The entity declaration specifies its notation.

In XML, the graphic and its file format can be represented in a an unparsed entity. Then the XML document can use this entity as an attribute in a graphic element to include the graphic in the document.

FrameMaker provides tools for creating a graphic. Alternatively, your users can import an external graphic, either by copying it directly into your FrameMaker document or by referring to an external graphic. In the latter case, the graphic remains in an external file and the name of the file is associated with the document. FrameMaker recognizes a graphic in several file formats, such as MIF or Sun raster format. Because FrameMaker determines the format directly by reading the graphic, you don't need to tell it the format of a graphic explicitly. Hence, there is no need to attach names to the formats.

- For information on translating graphics, see Chapter 24, "Translating Graphics and Equations."

Equations

As with graphics, markup has no standard mechanism for representing equations, while FrameMaker has a complete tool for working with them. Once created, however, equations in FrameMaker have characteristics very similar to graphics. For this reason, FrameMaker treats

equations in essentially the same way as graphics for markup import and export, and this manual discusses them together.

- For information on creating graphics and equations in FrameMaker, see the *Using FrameMaker*.
- For information on translating equations, see Chapter 24, "Translating Graphics and Equations."

Tables

FrameMaker has a complex facility for creating and manipulating tables within a FrameMaker document, including several special element types for use in your EDD. Tables have parts such as a title, body, rows, and cells.

Markup data doesn't have a standard mechanism for representing tables. As a result, their representation is unique to each DTD. In practice, many DTDs use the *CALS or OASIS* table model, which is similar to the table description supported by FrameMaker. Other DTDs can have element structure that is not automatically recognizable as a table, but that needs to format as a table.

When you create an EDD from a DTD, FrameMaker automatically recognizes the CALS table model and creates elements of the appropriate type. If you have a different table model in your DTD, you'll need to supply rules to identify the table structure.

- For information on working with tables in FrameMaker, see the *Using FrameMaker*.
- For information on defining tables and table parts in an EDD, see Chapter 13, "Structure Rules for Containers, Tables, and Footnotes."
- For information on translating tables, see Chapter 23, "Translating Tables."

Cross-references

A *cross-reference* is a passage in one place in a document that refers to another place (a *source*) in the same document or a different document. While the XML and SGML standards do not explicitly support cross-references, they do provide the declared values `ID`, `IDREF`, and `IDREFS` for attributes, and attributes using these declared values customarily represent cross-references. FrameMaker can use this model for cross-references within a FrameMaker document.

FrameMaker provides several additions to the cross-reference model suggested by XML and SGML. You need to keep these possibilities in mind when you work with cross-references:

- For markup data, both the cross-reference and its source are elements. In FrameMaker the source of the cross-reference can be an element but can also be a paragraph or a spot in a paragraph.
- For markup data, attributes contain the information connecting a cross-reference to its source. In FrameMaker you can also store the information in markers instead.
- XML and SGML allow a single attribute value to link several sources. FrameMaker requires a separate cross-reference for each citation.

Keep in mind also that FrameMaker and markup have different views of what constitutes an internal or external cross-reference. In FrameMaker, a cross-reference to a different file is always an external cross-reference. In markup, cross-references to different entities in a single document are always internal cross-references. So cross-references between components in a FrameMaker book are considered external, but cross-references between the text entities that correspond to those components are internal, since the entire book translates to a single XML or SGML document.

The `ID/IDREF` mechanism is natural in markup data for *internal cross-references*, those in which the source and the cross-reference are in the same markup document. However, it cannot be used with *external cross-references*, those in which the source and the cross-reference are in different markup documents. FrameMaker provides a single mechanism for both internal and external cross-references.

FrameMaker represents external cross-references in XML with a variation of the `ID/IDREF` model. Instead of an `IDREF` attribute, it uses a `srcfile` attribute whose value identifies the file containing the source element as well as the value of its `ID` attribute.

- For information on creating cross-references in FrameMaker documents, see the *Using FrameMaker*.
- For information on translating cross-references to markup, see Chapter 25, "Translating Cross-References."

Processing instructions

Markup data can include *processing instructions* to modify the treatment of a document in some system-specific manner. FrameMaker translates most processing instructions as markers in your FrameMaker document. It also defines a small number of special processing instructions for handling FrameMaker markers, books, and book components. You can use XSLT or the FrameMaker Developer's Kit (FDK) to handle other processing instructions when reading markup documents, but not when creating an EDD from a DTD.

- For information on handling processing instructions, see Chapter 22, "Translating Entities and Processing Instructions," and Chapter 29, "Processing Multiple Files as Books."

Associate a structured application using processing instructions

You can specify the information about the `structapps.fm` and the application required to open an XML document in the processing instruction in FrameMaker. The processing instruction should appear before the root element in the XML document. The syntax of the processing instruction is as follows:

```
<?Fmwd AppLocation "structapps.fm" AppName "ApplicationName"?>
```

For example, in the following code snippet, FrameMaker reads the `structapps.fm` file specified in the processing instruction. It then identifies the 'SendMail' application from the

structapps.fm file and uses it to open the XML file. This way the structured application file is opened faster.

```
<?xml version="1.0" encoding="utf-8"?>
<?Fmwd AppLocation "http://cms-fm/FM9/structapps.fm" AppName
"SendMail"?>
<Mail xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="email.xsd">
<from>Asterix</from>
<to>Obelix</to>
<subject>Lost Dogmatix</subject>
<body>Please Help.</body>
</Mail>
```

FrameMaker uses the structapps.fm file to locate referenced files, such as read-write rules, templates, schemas, and DTDs. If the structured application referenced files are located on a web server FrameMaker downloads all the referenced files from the server to a temporary location on the local disk. For the referenced files, you can use absolute paths or paths relative to the XML file being edited. The structapps_path can point to HTTP or local paths.

You can specify the processing instruction in an XML file in one of the following three ways.

- `<?Fmwd AppLocation "structapps_path" AppName "Application"?>`

  FrameMaker silently reads the structapps.fm file from the location specified in the structapps_path and associates the specified application at the time of opening and saving an XML file. If the application is not specified in the structapps.fm file, FrameMaker opens or saves the XML files with no application.

- `<?Fmwd AppLocation "structapps_path" ?>`

  FrameMaker silently reads the structapps.fm file from location specified in the structapps_path at the time of opening or saving an XML file. Since the structured application is not specified, FrameMaker will prompt you to select one of the applications specified in the structapps.fm file.

- `<?Fmwd AppName "Application"?>`

  FrameMaker uses the current structapps.fm file but uses the application name specified in the processing instruction of the XML file.

Parameter entity declarations

A DTD can use parameter entities as references to declaration content that is repeated in different statements. When FrameMaker reads a DTD with parameter entities, it expands them out to the full content of the reference. Any declarations that use a parameter entity will translate with the correct content, but the resulting EDD does not retain the parameter entity declaration nor the parameter entity itself.

When you create an EDD, you can use FrameMaker variables as references to repeated content. When FrameMaker translates the EDD to a DTD, it does not convert these variables to corresponding parameter entities.

7 The XML and FrameMaker Models

Before building a structure application that is designed to work with XML, you should understand the specific differences between XML and the FrameMaker document model. For a discussion about markup in general, see Chapter 7, "A General Comparison of Markup and FrameMaker Documents."

Element and attribute definition

The XML specification includes the following features concerning the declaration of elements and attributes:

Supported characters in element and attribute names

In XML, various markup tokens (Element names, attribute names, etc.) can include characters from the upper range of UNICODE. While FrameMaker can read XML content that contains the full range of UNICODE, FrameMaker has limited support of the characters for markup tokens. This support is limited to characters of the current display encoding. This display encoding can be determined by the locale of the current operating system, or by specifying a display encoding for the XML structure application. The characters you can use are:

Character set:	For this language:
`FrameRoman`	Western European languages
`JIS X 0208:1997`	Japanese
`BIG5`	Traditional Chinese
`GB2312-80`	Simplified Chinese
`KSC5601-1992`	Korean

For a given encoding, character support in markup tokens is further limited by the set of reserved characters. To see which characters are reserved for FrameRoman encoding, refer to the online manual, *FrameMaker Character Sets* and note the reserved characters in the standard character set.

For more information, see Developer Reference, page 27: Specifying the character encoding for XML files.

Multiple attribute lists for an element

The XML specification allows definition of an element's attributes in more than one attribute list. FrameMaker does not support this feature.

Namespace declaration

FrameMaker supports the use of namespace declarations in XML. A namespace can be declared as an element attribute in the DTD, or it can be declared in a given element within the document instance.

FrameMaker does not require you to declare namespace attributes in the EDD.

In FrameMaker you can enable or disable namespaces in the XML structured application. To see namespace information, you can choose **Element > Namespaces** to display the NameSpaces dialog box. Or namespace information may appear in the structure view elements that declare namespaces. The display of namespace information in FrameMaker depends on whether namespaces are enabled, as well as how the namespaces are declared in the DTD.

For a structure application with namespaces enabled:

- When a namespace is declared in an element but not in the DTD, the namespace information appears in the Namespaces dialog box. The structure view shows an asterisk for the element that has the namespace declaration.
- When a namespace is declared in an element and is also declared in the DTD, the namespace information appears in the Namespaces dialog box. The structure view shows the namespace as a valid element attribute with no value. The structure view also shows an asterisk for the element.
- When a namespace is declared in the DTD, but is not declared in any element, the structure view shows the namespace as a valid attribute with no value.

For a structure application with namespaces disabled:

- When a namespace is declared in an element but not in the DTD, the structure view shows the namespace as an invalid attribute, with the URI as the attribute value.
- When a namespace is declared in an element and is also declared in the DTD, the structure view shows the namespace as a valid attribute, with the URI as the attribute value.
- When a namespace is declared in the DTD, but is not declared in any element, the structure view shows the namespace as a valid attribute with no value.

When you export the FrameMaker document to XML, the namespace information is written to the XML.

> **Note: XML Schema:** You must enable namespaces to allow FrameMaker to validate XML against a Schema definition upon import and export. Schema allows an XML document to reference multiple Schema locations in different namespaces. When this is the case, only the first namespace is used. See Developer Reference, page 201: Schema location for additional information.

Rubi text

Japanese text uses a form of annotation known as Rubi to express the phonetic representation of a word over the characters used to spell the word. The XML specification suggests a representation for Rubi as follows:

```
<ruby>
<rb>Oyamoji text</rb>
<rt>Rubi text</rt>
</ruby>
```

To import that representation you would need read/write rules similar to the following:

```
element "ruby" is fm rubi group element "MyRubiGroup";
element "rb" is fm rubi element "MyRubi";
element "rt" is fm rubi element "Oyamoji";
```

For more information, see "Defining a Rubi group element" on page 161.

Unicode and character encodings

XML supports any character encoding that is compatible with UNICODE. FrameMaker offers equal support for any #PCDATA, CDATA, and RCDATA content.

FrameMaker has limited support of characters in markup tokens such as GIs and attribute names—see "Supported characters in element and attribute names" on page 100 and Developer Reference, page 27: Specifying the character encoding for XML files.

Read/write rules do not support Asian characters, so you cannot use rules to manipulate elements or attributes whose names contain such text.

Supported encodings

FrameMaker ships with support for the following encodings, listed by their IANA names:

IsoLatin1	TXIS
Ascii	TASC
Ansi	TANS
Shift-JIS	TSJS
Big5	KSC_5601
	TBG5
GB2312	UTF-8
	TGB
Korean	TKOR
EUC-JP	Shift_JIS

EUC-KR	US-ASCII
EUC-TW	UTF-16
ISO-8859-1	windows-1252

Using these IANA names, you can specify any of these encodings for export to XML—see Developer Reference, page 27: Specifying the character encoding for XML files.

These encodings are created in the ICU (International Components for Unicode) format, and stored as .cnv files. The supplied encodings are stored in the following location:

```
$installdir\fminit\icu_data
```

You can add other ICU encodings to the FrameMaker installation—you must create ICU compliant mappings and save them as .cnv files, then store them in the `icu_data` directory. Once you install a new encoding, you can refer to it the same as you refer to the other export encodings.

FrameMaker associates a given ICU encoding to an internal display encoding that is appropriate for a given language. Because any number of languages can be used in an XML file, FrameMaker cannot make that association automatically. The XML document prolog specifies an encoding, but the document may contain elements or other constructs that override the language implied by that encoding.

As of this writing you can find more information about International Components for Unicode (ICU) at http://www-124.ibm.com/icu/.

FrameMaker display encodings

By default, FrameMaker uses the display encoding that matches the input locale of your operating system. However, you can specify a different encoding in your XML structure application. FrameMaker supports the following display encodings:

Display encoding:	For this language:
`FrameRoman`	Western European languages
`JISX0208.ShiftJIS`	Japanese
`BIG5`	Traditional Chinese
`GB2312-80.EUC`	Simplified Chinese
`KSC5601-1992`	Korean

On import, when FrameMaker encounters an unknown character in #PCDATA it imports the character as a marker of type `UNKNOWNCHAR`. The marker content is a text string representing the UTF-16 encoding for that character. It imports unknown characters in CDATA as XML character references.

Encoding for XML files

The XML specification states that an XML document must either specify an encoding in the prolog, or it must be UTF-8 or UTF-16. FrameMaker follows this specification by assuming UTF-8 by default if there is no encoding specified in the XML file.

If you read an XML file with character encoding that doesn't match either the declared encoding or the default encoding (if no encoding is declared), it is likely that the import process will encounter a character that doesn't match the encoding FrameMaker uses. In that case, you will get a parsing error that says the document is not well-formed due to a bad token.

8 The SGML and FrameMaker Models

Before building a structure application that is designed to work with SGML, you should understand the specific differences between SGML and the FrameMaker document model. For a discussion about markup in general, see Chapter 7, "A General Comparison of Markup and FrameMaker Documents."

SGML declaration

According to the SGML standard, an SGML document contains an *SGML declaration*, a *prolog*, and a *document instance set*. The SGML declaration includes information on the specific syntax in effect for the document. In the absence of an SGML declaration, your SGML applications can use the *reference concrete syntax* defined in the SGML standard.

SGML features with no counterparts

SGML features with no FrameMaker counterparts include:

- Short reference mapping or usage
- Markup minimization
- Content references

FrameMaker correctly interprets a document with markup using any of the features mentioned in this list. However, it does not retain the information that the document's markup used the feature. For example, your document might use *markup minimization* to omit certain start-tags. If it does, the parser interprets the document as though the omitted start-tags were present. The rest of FrameMaker's processing can't distinguish whether or not the start-tags were omitted.

FrameMaker also doesn't use any of the above features when writing SGML documents. If you want an SGML document written by FrameMaker to use one of the features, you must write a Structure API client. For information on writing a Structure API client, see the online manual that comes with the FDK, the *Structure Import/Export API Programmer's Guide*.

Marked sections and conditional text

Both SGML and FrameMaker have mechanisms for specifying portions of a document that can be included or left out as needed. In SGML, this mechanism is *marked sections;* in FrameMaker, it is *conditional text*. The details of these two mechanisms differ significantly. For this reason, when translating between FrameMaker and SGML, the software does not attempt to preserve the fact

that information was tagged as conditional text in a FrameMaker document or that it occurred in a marked section in an SGML document.

When reading an SGML document, if the SGML parser used by FrameMaker encounters a marked section declaration with the effective status `IGNORE`, it doesn't include that section. If the effective status is `INCLUDE`, `CDATA`, or `RCDATA`, the software appropriately interprets and translates the marked section. The software doesn't annotate marked sections in the resulting EDD or document. Since your modifications only affect documents after they have passed through the parser, you cannot modify this behavior.

Similarly, if FrameMaker encounters conditional text that is hidden when writing a FrameMaker document as SGML, it does not include that text in the SGML document. All other text, whether it is unconditional or conditional, is included in the SGML document. Conditional text is not annotated in any particular way in the resulting DTD or document. You can write a Structure API client to change the exported document instance to reflect condition tags.

For information on working with conditional text, see the *Using FrameMaker*.

Unsupported optional SGML features

The SGML standard defines some features as optional, meaning that a specific implementation does not have to accommodate these features to be considered a conforming SGML system.

The following optional SGML features are not supported by FrameMaker:

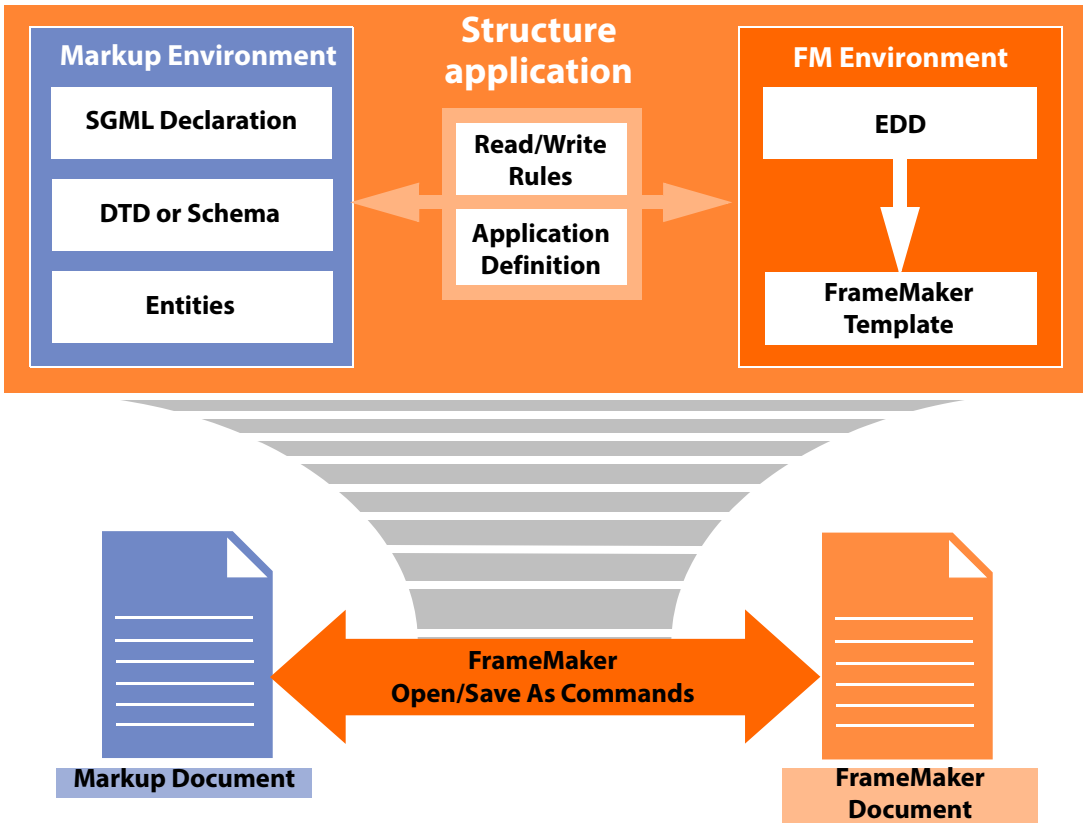
- `DATATAG`
- `RANK`
- `LINK`
- `SUBDOC`
- `CONCUR`

Your SGML documents cannot contain any of these features. If they do, FrameMaker signals an error and terminates processing. You cannot change this behavior with the FDK.

9 Creating a Structure Application

This chapter describes the major tasks you'll perform to create a structure application and describes the pieces your application might contain. Much of your work in creating an application involves writing read/write rules to modify the software's default behavior. Part IV, "Translating Between Markup Data and FrameMaker," describes specific default behaviors and explains how you can change them.

The diagram shows your completed structure application at work. Information in the application funnels into the Open and Save As commands in FrameMaker, adapting their default translation behavior to your markup and FrameMaker environments. Your structure application has information (such as a DTD) specific to the markup representation of documents, other information (such as an EDD) specific to the FrameMaker representation of documents, and rules and definition to bridge this information. The application pieces are described in more detail in "Pieces of a structure application" on page 120.



The development process

As a structure application developer, your primary task is to modify the default translation behavior of FrameMaker to fit the circumstances of your application. This section gives an overview of the steps used to create a structure application and to deliver that application to your end users.

At the highest level, there are four major application development tasks:

1.Get an initial version of both an EDD and a DTD.

You typically start application development with either an existing EDD or an existing DTD. In some situations, however, you have neither of these things. Your first task is to provide yourself with either an EDD or a DTD as a starting point and then use the software to create an initial version of a DTD if you started with an EDD or an EDD if you started with a DTD. If your element declaration is in XML Schema, FrameMaker can translate that to a DTD, and you can continue from there to create the EDD.

2.Get sample documents to test your application.

If you don't have sample markup documents and FrameMaker documents to test the progress of your application, you will need to create them.

3.Create read/write rules to modify the translation between the EDD and DTD.

Once you have an EDD, a DTD, and some sample documents, start an iterative process of analysis and application modification. Write read/write rules to modify how the software translates between markup and FrameMaker.

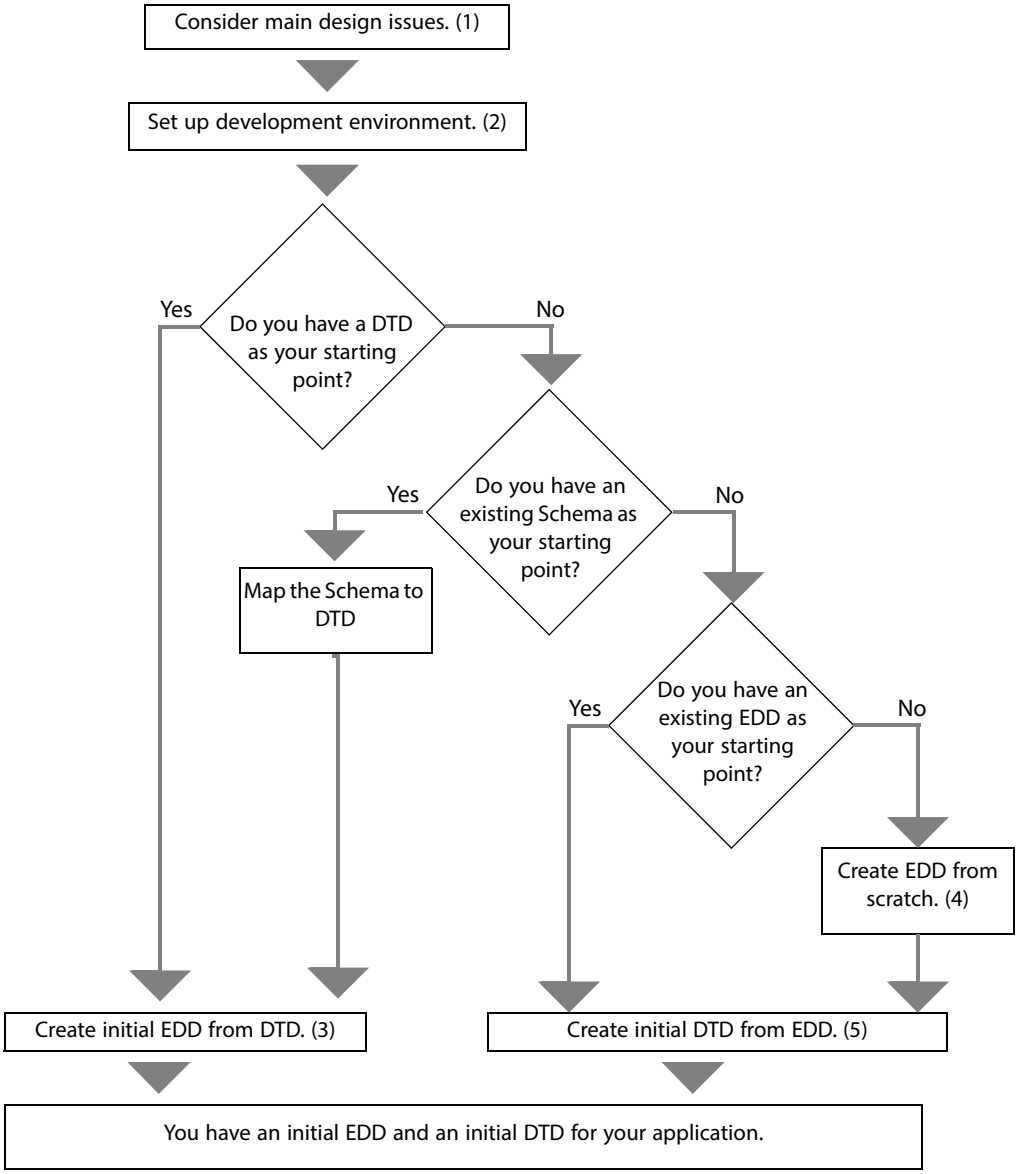
4.Use the FDK, XSLT and FrameMaker tools to finish your application.

Once you've done as much as you can with read/write rules, you may have to write a *structure API client* or XSLT style sheets to further modify the software's behavior. Consider other modifications you want, such as customizing the FrameMaker end-user interface to remove inappropriate commands or add new ones. Finally, assemble all the pieces of your application for delivery to your end users.

The four following sections provide more detail on the major tasks. To simplify the discussion, they have no pointers to more information on the various steps. For these pointers, see "For more information" on page 119.

Task 1. Producing an initial EDD and DTD

This flowchart shows the steps you follow to produce an initial EDD and DTD for your application, and the notes below the chart give more detail on some of the steps. What you do once you have the development environment set up depends greatly on the situation at your company. Numbers in the flowchart refer to the notes below the chart.



The following notes give more detail on some of the steps in the chart. Note numbers match the chart numbers.

(1) Consider the main design issues.

You need to think about:

- the kinds of documents you need to support

- what end users need in terms of access to both FrameMaker and markup versions of the same document
- the environment (XML, SGML, or FrameMaker) in which documents will be created and delivered
- whether end users work with only one structure application or multiple applications

(2) Set up the development environment.

You or your system administrator must install FrameMaker.

Once FrameMaker is properly installed, you need to tell it where to find rules you'll write and other information it may need. Collect this information in an application definition that you can associate with the EDD or the markup document element.

At this point, you can create a simple application file, giving the application a name and specifying the location of existing files, such as a, DTD or schema, and (for SGML applications only) the SGML declaration. Later in the development process, you'll need to include other information in the definition.

(3) Create an initial EDD from your DTD, if you have one.

If you're starting with a DTD, choose **StructureTools > Open DTD** to create an initial EDD to see how the software translates your DTD with no help from your application. You use this initial EDD during your analysis to see how you want to translate markup constructs into FrameMaker constructs.

In the absence of read/write rules, FrameMaker translates:

- Markup elements to FrameMaker elements of the same name—For SGML the FrameMaker element names receive initial capitalization; for XML the FrameMaker elements are the same case as the element and attribute declarations.
- Markup attributes to FrameMaker attributes, assuming that attributes contain extra information about their associated elements
- entities to various constructs such as FrameMaker variables

The software produces a log file if it encounters any problems while processing your DTD.

If your XML declarations are in Schema, you can specify the Schema file in the application definition, and FrameMaker automatically maps the information to DTD and creates the DTD file for you when you import a document. In general, types are converted on load to DTD-equivalent types. A general warning is shown if there are any unsupported constructs in the Schema that cannot be converted to DTD.

(4) Create an EDD, if you're starting from scratch.

If you're in the situation of having neither a preexisting EDD nor a preexisting DTD, you need to create one or the other before you can create a structure application. Because of the richer

semantics available in an EDD, you should first create an EDD and its associated FrameMaker template and then continue the development process from there.

If you're starting from scratch and the sample documents you intend to use are unstructured FrameMaker documents, you may want to do this step in conjunction with the next major task.

(5) Create an initial DTD from your EDD.

If you have a preexisting EDD, choose **StructureTools > Save As DTD** to create an initial DTD to see how the software translates your EDD with no help from your application.

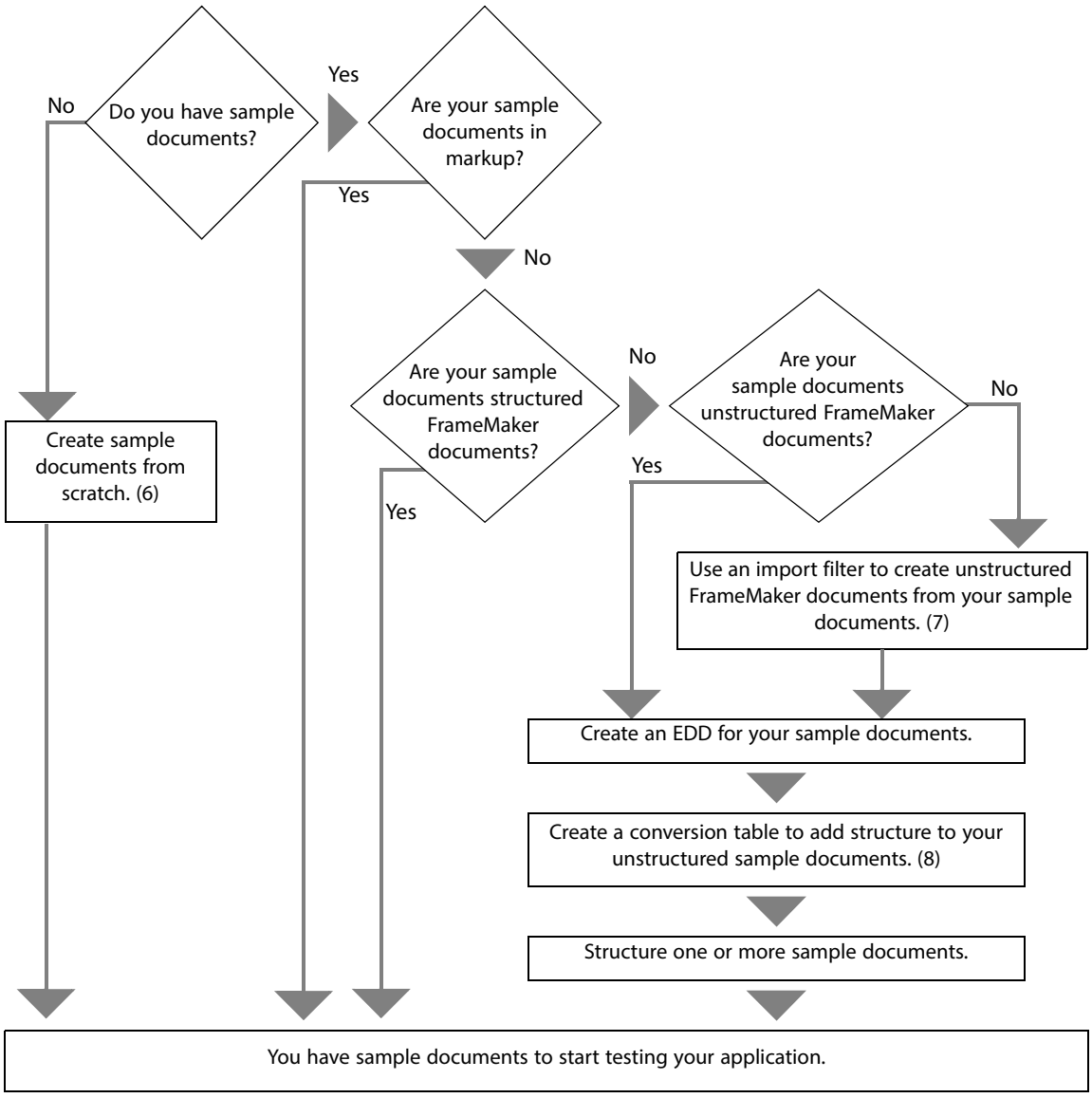
The DTD has element and attribute definition list declarations corresponding to the element and attribute definitions in the EDD. The software reads through the EDD, processing the elements and their attributes one at a time. For each definition, the software

- creates a markup element of the appropriate type
- produces an attribute definition list declaration for any element that has defined attributes
- writes notation declarations for use with graphics and equations and produces comments in the DTD corresponding to those in the EDD

An EDD includes more semantic information on the usage of its elements than does a DTD. For example, there are special element types corresponding to markers, system variables, and graphics, among other things. The declarations in the DTD created by FrameMaker reflect this information.

Task 2. Getting sample documents

This flowchart shows the steps you follow to get sample documents to test with your application, and the notes below the chart give more detail on some of the steps. Numbers in the flowchart refer to the notes below the chart.



The sample documents you need depend on your starting point. For example, if you already have markup documents, you probably won't yet have FrameMaker documents. If you have existing unstructured FrameMaker documents, you may need to structure them. Later in the process, you may decide to create more sample documents.

(6) Create sample documents if you have none.

Add to the collection as you develop the application, in order to test particular parts of the application. If you're starting with a preexisting DTD, create sample XML or SGML documents. If you're starting with a preexisting EDD, create structured FrameMaker documents.

(7) Use document import filters to get FrameMaker documents.

If your sample documents are in a file format other than XML, SGML, or FrameMaker, you should convert them to unstructured FrameMaker documents and then use the FrameMaker's conversion tools for structuring unstructured documents.

(8) Structure unstructured FrameMaker documents if necessary.

If you've reached this point, you have unstructured FrameMaker documents to use as your sample documents. FrameMaker provides tools to aid you in structuring unstructured documents.

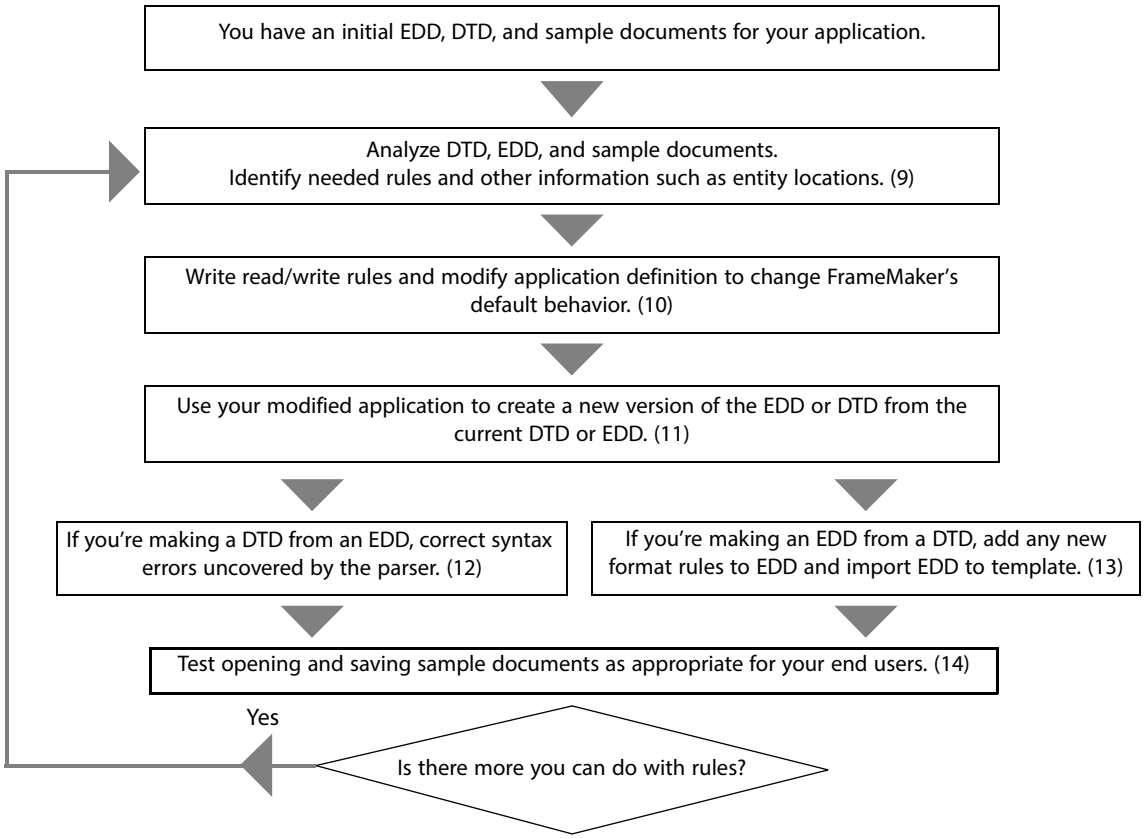
You create a conversion table that describes a set of unstructured documents and then use FrameMaker commands to structure your documents. Once the documents are structured, you also need an EDD and structured template that corresponds to the elements described in the conversion table.

If you use FrameMaker's conversion tools, you could use the resulting EDD as one of the starting points of your structure application.

Task 3. Creating read/write rules

Once you have both an EDD and a DTD, you can use read/write rules to refine how FrameMaker translates between them. This portion of application development is an iterative process.

The following flowchart shows the steps of this process. The notes below the chart give more detail on some of the steps, and numbers in the flowchart refer to the notes.



(9) Analyze the details of what your application must do.

If you are not already familiar with it, you need to study the original DTD or EDD, with the partially completed new EDD or DTD and sample documents, to determine what constructs are defined and how they are intended to be used. Sample documents are invaluable in understanding the meaning of the defined element structures; the partially completed EDD or DTD lets you know how well FrameMaker performs the translation so far.

If you start with a DTD, the steps of your analysis should be similar to the following:

1.Examine element declarations or definitions.

Determine the representation of elements and, from a high level, the element and document groupings. Determine which elements are used for text or as containers for other elements and which are used for special constructs such as tables or graphics.

For an SGML application, determine the level of minimization provided by the DTD. *Markup minimization* is irrelevant in FrameMaker's internal representation, but you may need to understand its usage in sample documents. During the early stages of development, you probably won't try to reproduce minimization in markup, but you may later write a structure API client to do so.

Determine the purpose of elements with a declared content of EMPTY. These will translate to the appropriate FrameMaker constructs, such as graphic elements.

2.Examine attribute use.

Examine attributes to distinguish attributes that control formatting from attributes that simply provide information about the associated element. Formatting attributes for special constructs such as tables or graphics may become formatting properties in a FrameMaker document.

3.Examine entity declarations.

Determine how entities are used and which become text, variables, graphics, book files, or special characters in FrameMaker. The SGML standard defines several sets of character entities. Check whether your DTD refers to any of these character sets. Some XML DTDs also use character entities although not necessary in most cases.

4.Examine notation declarations.

Determine how to represent non-SGML or non-XML data (NDATA) in FrameMaker. NDATA can often be represented directly as FrameMaker graphic or equation elements.

5.Examine elements and attributes used as cross-references.

FrameMaker assumes markup attributes with declared value of ID or IDREF refer to cross-references. Your DTD may also use other declared values, such as IDREFS, NAME, and NAMES, for cross-references.

(10) Write rules and modify the application definition.

While working on your rules, you may need to modify your application definition. For example, you need to tell the software how to locate entities that have *public identifiers*.

Some parts of the task of changing the translation between the two representations will require more work than others. For example, standard elements such as those representing text in paragraphs or lists may translate with no help from you. But other elements, such as those representing graphics, will require your assistance.

The steps described here suggest working on a portion of the representation only as far as possible with rules and waiting until you're finished with rules before attempting to write a structure API client or XSLT transformations. An alternate approach is to work on a particular representation until it is complete, before moving on to the next representation, even if that means creating a structure API client.

(11) Create a new version of the EDD or the DTD.

After you have written the rules your application requires, use the appropriate command to update your EDD or to recreate your DTD. Where they are applicable, FrameMaker uses your rules to modify its processing. Where it encounters no applicable rule, the software uses its default behavior.

(12) Correct syntax errors uncovered by the parser.

FrameMaker uses parsers to interpret XML and SGML markup in ways that comply with the associated standards. When FrameMaker processes a DTD or a markup document on import, the parser interprets markup and identifies the information in the document such as element declarations and element start-tags.

The parser may find problems that cause the DTD to be syntactically invalid. Some problems can be corrected with read/write rules, while others may require you to modify the DTD.

> **Note: SGML:** The SGML standard allows wide variations in the actual markup of specific documents. For example, it allows variations in the maximum length of the sequence of characters specifying a name, in the case-sensitivity of names, and in markup that may be omitted altogether. This is specified in the SGML declaration. If the parser encounters errors in the DTD, you may be able to fix them by changing the quantity and capacity limits defined in the SGML declaration used by the current structure application.
>
> Modify the SGML declaration using a text editor to correct these errors. Add the SGML declaration to the application definition. Rather than recreating the DTD with the modified application, you can manually invoke the SGML parser to validate your changes.

(13) Add format rules to the EDD and import it into the template.

If you started with a DTD, you (perhaps working with your document designer) now expand the EDD and FrameMaker template to include appropriate format rules.

FrameMaker supports context-sensitive formatting. An element definition can have one or more format rules that specify formatting applied to the element in a particular context in the document. For example, a `Head` element inside a single `Section` element might be formatted differently than a `Head` element inside a `Section` element that's nested in another `Section` element. (That is, a first-level heading probably looks different from a second-level heading.)

(14) Test with sample documents.

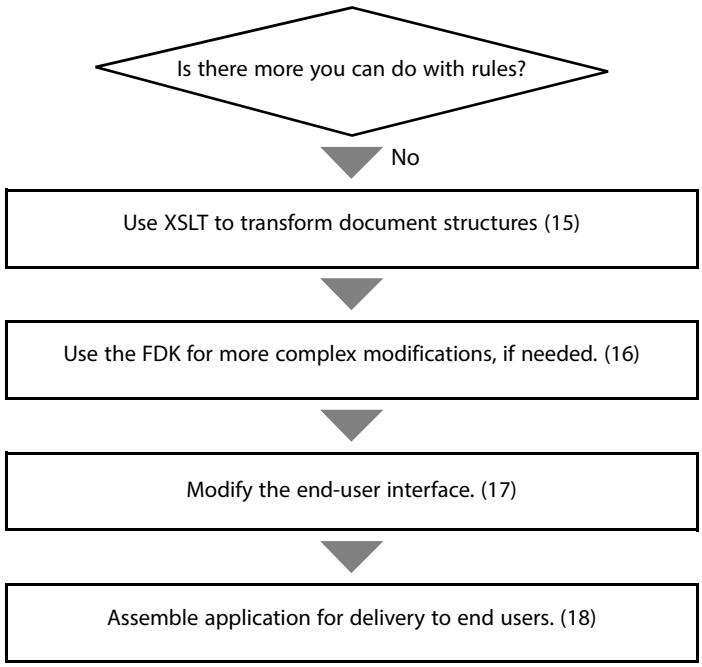
You should test your application at various stages of its development and you should test your read/write rules thoroughly. After you make a change to the read/write rules select **StructureTools > Check Read/Write Rules**.

If end users will be opening markup documents in FrameMaker, invoke the software with sample documents that test your modifications in that direction. If users will be saving FrameMaker documents as markup, invoke the software with sample documents to test those modifications. If your end users have existing documents for you to use, that can be very helpful. In any case, you may want to create sample documents that test most features of your application.

Task 4. Finishing your application

Once you've progressed as far as you can with read/write rules and the application definition, you may find that FrameMaker still does not translate your documents as completely as you wish. If so, you'll have to use the FDK to complete the translation. At this time, you may also decide to modify FrameMaker's end-user interface to be more appropriate to your end users. Finally, you put all the pieces of the application together to deliver to your end users.

The flowchart shows these steps. The notes below the chart give more detail on some of the steps, and numbers in the flowchart refer to the notes.



(15) Use XSLT to transform document structures

XML only: Some DTDs use element content models that are not compatible with FrameMaker's object elements. For example, your DTD or schema may permit child elements or text content in an `xref` element. FrameMaker cross-references are always `empty` elements. Using a pre-process XSLT you can transform the XML into a structure that is suitable for FrameMaker, while a post-process XSLT transform reverses the process.

(16) Use the FDK for more complex modifications.

In some situations, read/write rules are not sufficient to describe the appropriate translations between markup and FrameMaker. For example, your SGML DTD may use special markup minimization, and rules cannot specify creation of minimized markup during export of documents to SGML. In this case, you'd need to write a structure API client.

Even if you need to write a structure API client, you should use read/write rules for as much of the translation as possible. This will simplify what your client must do.

(17) Modify the end-user interface.

You can modify the menu items available to your end users. At the very least, you'll probably want to remove most of the developer only commands on the **StructureTools** menu. You may wish to remove other menu items as well. For example, if you want all formatting controlled by the format rules in your EDD, you can remove direct formatting commands from your users' menus. In addition, you might write FDK clients to help your end users in other ways and provide access to those clients on the menus.

(18) Assemble your application for delivery to end users.

You need to make sure the pieces of your application are appropriately packaged for your end users. You may even choose to write scripts to make it easier for your end users to access the application. You should also be prepared to update your application as end users encounter bugs or request enhancements. In addition, be prepared to handle revisions of the EDD or DTD.

For more information

You can find more information on topics in the previous section in other parts of this manual and in other manuals. The table lists the topics by the notes that introduced them.

Note	Topic	Location
2	Installing FrameMaker	*Installing* FrameMaker
	Creating an application definition	"Application definition file" on page 132
3, 4	Creating an EDD	Part III, "Working with an EDD"
	Working with log files	"Log files" on page 134
7	Using document filters available with FrameMaker	*Using Filters*
8	Creating an EDD and FrameMaker template	Part III, "Working with an EDD"
	Creating an EDD for an existing set of unstructured documents	Chapter 4, "Conversion Tables for Adding Structure to Documents"
9-14	How FrameMaker translates markup documents and how you can modify that with rules	Part IV, "Translating Between Markup Data and FrameMaker"
	Writing format rules	Chapter 15, "Text Format Rules for Containers, Tables, and Footnotes," and Chapter 15, "Object Format Rules"
	Importing an EDD into a FrameMaker template	Chapter 11, "Developing an Element Definition Document (EDD)"

Note	Topic	Location
15	Using the FDK to create a structure application	*Structure Import/Export API Programmer's Guide*
17	Modifying the user interface	*Customizing FrameMaker Products*
18	Assembling your application	"Pieces of a structure application," next

Pieces of a structure application

A structure application created with FrameMaker includes an external DTD subset, a FrameMaker template, and a read/write rules document. Other files may be included, depending on the particular application.

The following sections describe the possible parts of a structure application and your options for assembling those parts into a working application. Although you can include a structure API client as part of your application, this section does not talk about what to do with such a client.

Application definition file

Your application requires several files, search paths for these files, and other information. You provide FrameMaker with these paths and other information by placing them in the `structapps.fm` file. In this file, you create a definition for your application that includes the names of other needed files.

If you provide your users with several structure applications, put information about all of them in a single `structapps.fm` file. FrameMaker reads this file on startup, so your end users can choose any of the applications defined in this file. You need to deliver this `structapps.fm` file to your end users.

For information on the `structapps.fm` file, see "Application definition file" on page 132.

External DTD subset

A structure application pertains to a particular external DTD subset that the software uses when writing markup documents. You specify the external DTD subset in the application definition. While it is optional, to take full advantage of FrameMaker we recommend that you specify a DTD for your application. Your application will use it to:

- Export a FrameMaker document as markup.
- Import a partial markup instance as a text inset in your FrameMaker document.
- Import an entity reference to a markup file as a text inset in your FrameMaker document.
- Open an incomplete markup instance as a FrameMaker document.

SGML declaration

> **Note: XML:** The XML specification does not support user-defined SGML declarations. The following discussion is for SGML, only.

An SGML structure application can use a particular SGML declaration. If it does, you specify the SGML declaration in the application definition.

The SGML declaration is an optional part of an application. Individual SGML documents can begin with a declaration. If neither the SGML structure application nor a particular SGML document specifies a declaration, FrameMaker uses the SGML declaration described in the Developer Reference, page 225: SGML Declaration

If you do specify an SGML declaration, you can deliver it to end users as a separate file.

FrameMaker template

A structure application is associated with a particular FrameMaker template that you can specify in the application definition. The software uses the template to specify the structure and formatting of FrameMaker documents it creates from markup documents. If you do not specify a template, it uses the formats you would get if you used the **File > New** command and specified *Portrait*, rather than a particular template.

As you create your application, you can work with a FrameMaker EDD in a separate file. However, FrameMaker does not use a separate EDD file to find information on structure when opening a markup document; it gets the information directly from the FrameMaker template. At some point during the creation of the FrameMaker template, you must import element definitions from the EDD to the template. Because of this, you don't deliver the EDD to your end users as a separate file. For maintenance purposes, however, you should retain the EDD in a separate file.

Read/write rules document

You create read/write rules in a FrameMaker document and specify that document in the application definition. FrameMaker uses the rules document both when opening a markup document and when saving a FrameMaker as XML or SGML.

Because a read/write rules document can reference other documents, you can have more than one document containing rules. In the application definition, you specify only the primary file; the software locates the others through the `#include` mechanism in the primary file.

Your read/write rules document and any include files are separate files you deliver to your end users.

Entity catalogs

If your application requires special external entities or entity catalogs, you must deliver those as separate files to your end users. For information on entities and entity catalogs, see Chapter 22,

"Translating Entities and Processing Instructions," Chapter 11, "Working with Special Files," and Developer Reference, Chapter 10, ISO Public Entities

Documentation

As part of your structure application, you should write documentation to explain to your end users how to use the application. It is your responsibility to deliver such documentation either in online or printed form.

This manual is written for developers creating structure applications. Refer end users to the FrameMaker *User Guide*. You can remove this manual from your end users' installation directory.

Transformation file

An XSL transformation includes information such as XSL, Parser, and Output folder, for creating output from XML files. Application-specific transformations are stored in an XML file that you specify for the structured application in the `structapps.fm` file. A transformation file thus includes your application-specific XSL transformation scenarios.

Configuration file

A structured application may include an XML configuration file. This configuration file is optional and contains attributes with their suggested and default values. When opening a structured application, FrameMaker reads the corresponding configuration file if it exists, and automatically populates the attribute values. Users can choose to edit attribute values using the Configuration File editor.

MathML

Namespace prefix

You may choose to use a prefix for all MathML elements in your structured application. To use a namespace prefix for the MathML elements in your XML files, declare the MathML namespace prefix in your structured application.

Export entities as values

Prior to the FrameMaker 2015 release, MathML equations were saved as entities in XML. With the current release of FrameMaker, MathML equations are saved as Hex values. By default, the export entities as values construct is set to true. If you set the value of this construct to false, then FrameMaker might fail to load a file that contains a MathML equation.

Creating a FrameMaker template

You should deliver your FrameMaker template to end users as a separate file. The template is a FrameMaker document that includes element definitions and formatting information, but not the

ultimate content of your documents. By using a template, you ensure all documents for a specific application can have the same element definitions and formatting. For information about importing element definitions into your template, see "Creating an Element Catalog in a template" on page 166

For information on creating a FrameMaker template, see the FrameMaker *User Guide*. For information on creating the EDD for a template, see Chapter 12, "Developing an Element Definition Document (EDD)."

Following are descriptions of information you can put in your template that takes advantage of document structure. Included are descriptions of building blocks that define cross-reference formats, variable definitions, and format generated lists and indexes, as well as map FrameMaker elements for HTML export. For general information about cross-reference formats, variable definitions, generated lists and indexes, and HTML export, see the *Using FrameMaker*.

Cross-reference formats

Use the following building blocks to create cross-reference formats that refer to FrameMaker elements:

Building block	Meaning
<$elempagenum>	The page number of the source element
<$elemtext>	The text of the source element (up to the first paragraph break), excluding its autonumber, but including any prefix and suffix specified in the element definition
<$elemtextonly>	The text of the source element (up to the first paragraph break), excluding its autonumber and any prefix and suffix specified in the element definition
<$elemtag>	The tag of the source element
<$elemparanum>	The entire autonumber of the source element's first paragraph (or of the paragraph containing the source element), including any text in the autonumber format
<$elemparanumonly>	The autonumber counters of the source element's first paragraph (or of the paragraph containing the source element), including any characters between the counters
<$attribute[name]>	The value of the attribute with the specified name (or, if no value is specified, the default value)

You can specify building blocks to refer to the *ancestor* of the source element. Including the tag of an element in the building block indicates that it will refer to the nearest ancestor with the

specified tag. For example, a cross-reference to a subsection might also identify its parent section, as in the following: See “Types of plate boundaries” in “Plate tectonics.”

The following list shows how building blocks refer to ancestors of the source element:

Building block	Meaning
<$elempagenum[tag]>	The page number of the nearest ancestor with the specified tag
<$elemtext[tag]>	The text of the nearest ancestor (up to the first paragraph break) with the specified tag, excluding its autonumber, but including any prefix and suffix specified in the element definition
<$elemtextonly[tag]>	The text of the nearest ancestor (up to the first paragraph break) with the specified tag, excluding its autonumber and any prefix and suffix specified in the element definition
<$elemtag[tag]>	The tag of the nearest ancestor with the specified tag
<$elemparanum[tag]>	The entire autonumber of the first paragraph of the nearest ancestor with the specified tag
<$elemparanumonly[tag]>	The autonumber counters of the first paragraph of the nearest ancestor with the specified tag, including any characters between the counters
<$attribute[attrname:tag]>	For the preceding elements that are first siblings, then ancestors of the source element, the value of the attribute with the specified name (or, if no value is specified, the default value)

In each of the building blocks, enter the tag of the element to which you want to refer between brackets. For example, if you want to refer to the text of the source’s nearest ancestor tagged Section, you would use:

```
<$elemtext[Section]>
```

Variables

If you’re defining running header/footer variables that will refer to elements in structured FrameMaker documents, you can use building blocks that refer to elements or element attributes rather than to paragraphs.

The following building blocks in running header/footer variables refer to an element tag:

Building block	What it displays
<$elemtext[tag]>	The text (up to the first paragraph break), excluding its autonumber, but including any prefix and suffix specified in the element definition
<$elemtextonly[tag]>	The text (up to the first paragraph break), excluding its autonumber and any prefix and suffix specified in the element definition
<$elemtag[tag]>	The tag
<$elemparanum[tag]>	The entire autonumber of the element's first paragraph
<$elemparanumonly[tag]>	The first paragraph's autonumber counters, including any characters between them

Follow these guidelines for using element tag building blocks:

- Enter the tag of the element for which you want to display information between brackets.
- You can include a context label with the element tag to provide additional information about the element's location in the document structure.
- You can include more than one element tag in the brackets, separated with commas. With multiple tags, FrameMaker uses the first element it finds with one of the tags. An example building block is:

```
<$elemtext[Head(Level1),Head(Level2)]>
```

The following building blocks in running header/footer variables refer to element attribute values:

Building block	What it displays
<$attribute[name]>	The value of the attribute
<$highchoice[name]>	The highest value of the attribute that appears on the page (where highest means the value closest to the bottom of the pop-up menu on the right side of the Attributes window)
<$lowchoice[name]>	The lowest value of the attribute that appears on the page (where lowest means the value closest to the top of the pop-up menu on the right side of the Attributes window)

Follow these guidelines for using element attribute value building blocks:

- Enter the name of the attribute whose value you want to display between brackets. If a list of possible values is not defined for the attribute used in a <$highchoice> or <$lowchoice> building block, the building block will be ignored.

- You can specify elements to consider when searching for an attribute value to include in the running header or footer. To do so, place a colon after the attribute name, followed by one or more element tags separated by commas. For example, a variable with the following definition would display the highest value of the Security attribute of the first-level and second-level Section elements.

```
<$highchoice[Security:Section(Level1), Section(Level2)]>
```

Special text flows to format generated lists and indexes

Many formatting aspects of a list or index are controlled by a special text flow on a reference page in the generated file. The name of the reference page matches the default filename suffix, such as TOC for a table of contents or IX for a standard index. These reference pages contain building blocks to specify formatting of the entries in the generated files.

The following building blocks apply only to structured documents in FrameMaker. If you use an element-based building block to refer to an unstructured paragraph, the information won't appear in the generated list. If you use a paragraph-based building block to refer to an element, the information included in the generated list will come from the paragraph that contains the beginning of the element.

Building block	What it displays
<$elemtextonly>	Displays the text of the first paragraph of the element, excluding any autonumber, and the element's prefix and suffix, if any. Note that prefix and suffix are only excluded for the specific element; if the paragraph text is in a child element that has a prefix or suffix, the prefix or suffix will be included.
<$elemtext>	Displays the text of the first paragraph of the element, excluding any autonumber, but including the element's prefix and suffix, if any.
<$elemtag>	Displays the element tag

For general information about how FrameMaker generates and formats lists and indexes, see the FrameMaker user's manual.

HTML mapping for export

The *Using FrameMaker* includes instructions for converting a FrameMaker document to *HTML*. Converting structured FrameMaker documents to HTML uses nearly the same process, but there are some differences which take advantage of the structure in your documents.

When setting up an application, you should set up the mapping to HTML in advance so your users don't need to do that work. After you build a template, you can use the process described in the FrameMaker user's guide to set up the HTML mapping for it. The mapping information will be stored on the HTML reference page of your template.

You can also associate HTML mapping with an EDD. To do this, create a reference page in your EDD named EDD_HTML. Then copy the mapping information from the HTML reference page of your template to the EDD_HTML reference page. If your EDD includes the `ImportHTMLMapping` element, then the mapping information will be copied to any document that imports your EDD. For more information about including mapping information with an EDD, see "Specifying whether to transfer HTML mapping tables" on page 152

Mapping FrameMaker elements to HTML elements

You specify how to map FrameMaker elements and attributes to HTML elements in the HTML Mapping table on the HTML reference page. For example, in the following table, Section elements map to HTML `<Div>`, and Head elements map to HTML `<H1>` through `<H6>`:

FrameMaker Source Item	HTML Item		Include Auto#?	Comments
	Element	New Web Page?		
E:Section	Div	1,2,3	N	
E:Head	H*		Y	
E:List	Ol		N	
E:ListItem	Li		N	
A:Description	alt			

In the above example, the following syntax is used to specify FrameMaker elements and attributes in the column titled FrameMaker Source Item:

`E:`*`elementname`*, where *`elementname`* is the name of the element

`A:`*`attributename`*, where *`attributename`* is the name of the attribute

Note the following points when mapping a structured document to HTML:

- In unstructured documents, there is a Headings reference page which maps specific paragraph formats to HTML headings, and assigns a hierarchy to them. In a structured document there is no need for the Headings reference page. You map FrameMaker elements to HTML headings in the Mapping table. The hierarchy is determined by the context of the FrameMaker element, just as context can determine the formatting of a FrameMaker element.
- The document can be divided into separate web pages at certain levels of hierarchy. In the above example, HTML export will create a new web page for every section of a level 1, 2, or 3.
- In unstructured documents, you map paragraph formats to list items, and specify the list type for that list item's parent. In a structured document, you specify the list type for FrameMaker list element, and then you map FrameMaker list items to the HTML `<Li>` element. Also, you do not specify the nesting of lists, since that can be determined by the hierarchy in the FrameMaker document. In the above example, `List` maps to HTML `<Ol>`, and `ListItem`

maps to HTML <Li>. If ListItem contains a List, on export to HTML the lists will nest correctly.

- In the above example, the Description attribute in FrameMaker maps to the alt attribute for HTML elements. Every occurrence of a Description attribute will result in an alt attribute in HTML, whether it is valid for that HTML element or not. The HTML specification says that User Agents should ignore attributes they don't understand, so this is not a major problem unless you need to validate your HTML file. For HTML elements that use the alt attribute (<IMG>, for example) your description can appear in place of that element.

Building blocks for structured documents

You can create HTML conversion macros that use information specific to elements and attributes. The following building blocks can be used when defining macros for elements and cross-reference formats:

Building block	Meaning
<$elemtext>	The text of the source element (up to the first paragraph break), excluding its autonumber, but including any prefix and suffix specified in the element definition
<$elemtextonly>	The text of the source element (up to the first paragraph break), excluding its autonumber and any prefix and suffix specified in the element definition
<$elemtag>	The tag of the source element
<$elemparanum>	The entire autonumber of the source element's first paragraph (or of the paragraph containing the source element), including any text in the autonumber format
<$elemparanumonly>	The autonumber counters of the source element's first paragraph (or of the paragraph containing the source element), including any characters between the counters
<$attribute[name]>	The value of the attribute with the specified name (or, if no value is specified, the default value)

10 Working with Special Files

This chapter discusses the special files you need to create a structure application. It tells you where to look for them in your directory tree; describes the workings of two of them, the application definition file and the log file; and points you to the appropriate chapters for information on the other files.

Location of structure files

FrameMaker provides a location for putting all of the special files associated with structure applications. Some files are provided with the software. Others are files you create and store at this location. Apart from this default location, you can copy the structure files to any other directory on your local machine or access them from a WebDAV enabled server.

FrameMaker looks for these files in a directory named `structure` in the home directory for FrameMaker. You can specify an alternate `structure` directory in the `maker.ini` file. For information about the `maker.ini` file, see the FrameMaker user's manual.

You can use the variable `$STRUCTDIR` to refer to the `structure` directory in the FrameMaker installation directory. The rest of this chapter follows this convention.

When you install FrameMaker, the `structure` directory contains the following files and subdirectories:

`default.rw`	The default read/write rules file. FrameMaker uses this file in the absence of a rules file in the current application. It also uses the file as the template when you create a new rules file with the **StructureTools > New Read/Write Rules File** command.
`structapps.fm`	The default master version of the application definition file.
`entfmts`	Default formats used for ISO public entities. For information on this file, see Developer Reference, Chapter 10, ISO Public Entities
`sgml/`	A directory containing SGML structure applications.
`sgml/docbook/`	The SGML DocBook starter kit files. For information on this directory, see the online manual *Using the DocBook Starter Kit*.
`sgml/isoents/`	Other ISO public entity files. For information on the contents of this directory, see Developer Reference, Chapter 10, ISO Public Entities
`xml/`	A directory containing XML structure applications.
`xml/DITA/`	The DITA XML application files.
`xml/xdocbook/`	The XML DocBook starter kit files. For information on this directory, see the online manual *Using the XDocBook Starter Kit*.

`xml/isoents/`	Other ISO public entity files. For information on the contents of this directory, see Developer Reference, Chapter 10, ISO Public Entities
`xml/xhtml/`	The XHTML starter kit files. For information on this directory, see the online manual Using the XHTML Starter Kit.

If you choose, you can create other directories under `$STRUCTDIR` to hold files for your applications.

Accessing structure files from a WebDAV server

Using FrameMaker, you can directly open and edit XML files stored on a WebDAV-enabled server.

1.Using a browser, specify the URL of the XML file.

You may need to provide authentication credentials at the time of login.

2.Click the edit icon on the browser's toolbar, and select **Edit with Adobe FrameMaker *<version>***.

The XML file is opened in FrameMaker.

3.Select the application name from the dialog box that appears. This dialog box does not appear if you have specified the application name in the processing instructions of the XML file.

4.Edit the document in FrameMaker and save the changes. The changes can be saved directly on the server based on the Preference settings. You can opt to update the file on the server each time you save changes or update the file only when you close it. When you close an XML file, the file is automatically checked-in and unlocked on the server.

FrameMaker uses the structured application definition file to locate referenced files, such as read-write rules, templates, and EDDs. If an EDD does not exist, FrameMaker generates it using the XML schema file. FrameMaker repeats this process every time the XML file is opened until you generate and save the EDD explicitly. FrameMaker reads the structured application definition file or the `schemaLocation` attribute in an XML file for identifying the related XML schema.

Application file paths to the schema, DTD, read-write rules, or template paths can be specified as HTTP paths in the `structapps.fm` file.

XML files can contain references to other files located on a WebDAV or non-WebDAV server. You can provide references to such XML files using an absolute path or a relative path. When you open an XML file containing references, FrameMaker downloads all the referenced files from the server to a temporary location on the local machine. These referenced files can be graphic files, text files, or other XML files.

For example, an XML file in which you want to insert an image resides on the following location:

```
<http://ndot-xp:8080/webdav/public/Auto-Template/TempGen.xml>
```

you can specify the reference to an image file using an absolute path:

```
<image file = "http://ndotxp:8080/webdav/public/Auto-Template/
image.jpg"/>
```

or a relative path:

`<image file = "image.jpg"/>`.

FrameMaker opens a referenced file if it is a FrameMaker file or a supported format, such as a text file.

Location of the application definition file

FrameMaker supports two versions of the `structapps.fm` file. The default or the master copy continues to reside in the install directory path of FrameMaker. But additionally, a user copy is created in the Application Data folder.

Documents and Settings\<user_name>\Application Data\Adobe\FrameMaker\<*version*>\

When you launch FrameMaker for the first time, the master copy of the `structapps.fm` file is copied from the install directory to the Application Data directory. Then on, if an application name exists in both the user copy and the master copy, the user copy definition is given precedence. If a structured application is not found in the user copy, then FrameMaker searches for it in the master copy.

You can access and edit the user copy of the `structapps.fm` file from FrameMaker do the following step:

- Select **StructureTools > Edit Application Definitions**. This command opens the user copy of the `structapps.fm` file.

> Note that the $STRUCTDIR variable still points to the master copy of the structapps.fm file in the FrameMaker installation directory.

There are multiple advantages of using two different copies of the `structapps.fm` file. As an administrator, you can use the master copy to propagate `structapps.fm` changes to multiple users. As a user, you can change an application definition in the user copy to override the same in the master `structapps.fm` file. If you want to make changes only to a single application while inheriting other applications' behavior from the master copy, ensure that you delete those definitions from the user copy.

> In the non-administrative mode on your machine, you may have to copy all the application files to a new location because they may not be writable. In such a case, you can either edit the user copy of the `structapps.fm` to specify the path to the new directory or edit the path mapped to $STRUCTDIR in maker.ini to point to the new directory with the structure files.

Application definition file

You need to deliver a `structapps.fm` file to your end users. In some situations, you make more than one application available to your end users at the same time. For example, your end users may sometimes need an application based on the DocBook DTD for writing technical

documentation and may at other times need an application based on a completely different DTD particular to your company. If so, you provide two applications in one `structapps.fm` file.

The `structapps.fm` file provides definitions for the available applications; it can also contain information relevant to all applications, such as a default place to look for entity files. These file paths can also be HTTP paths if the files reside on a WebDAV-enabled server.

FrameMaker can associate a particular document with a structure application in several ways:

- An EDD can explicitly associate itself with a structure application. If it does, all FrameMaker documents that use that EDD are automatically associated with the application.
- A structure application can name one or more document types (identified in the markup in the doctype declaration and in the document's root element). If it does, all markup documents that use one of those document elements are automatically associated with the application.
- If neither of the above occurs or if the associations are ambiguous, FrameMaker asks the end user to pick a structure application to use with a document.

A full description of each available element in the structured application definition document can be found in Developer Reference, Chapter 1, Structure Application Definition Reference.

Editing a structured application definition document

The `structapps.fm` file is a structured FrameMaker document. You can edit this file by choosing **StructureTools > Edit Application Definitions** and using standard FrameMaker editing techniques for structured documents. This command opens the `structapps.fm` file from the Application Directory and not from the FrameMaker installation directory.

When you open the `structapps.fm` file, you can view the elements by opening the Elements catalog and choosing All Elements from the Options dialog. Each of the elements is shown in the document as an entry with a descriptive string, and in most cases an editable text value. For example an element `Entities`, which contains an element `Entity`, which in turn contains an element `Filename`, is shown as:

Entity locations
Entity name: *ename*
Filename: *fname*

A feature that can be enabled or disabled is represented by an element that contains an `Enable` or `Disable` element, and is shown like this:

Namespace: *Enable/Disable*

You add entries to the file by inserting elements, and edit the text parts of existing entries to set or modify values.

FrameMaker reads the `structapps.fm` file on startup. While developing your application, you may need to change its contents. If you do so, have the software reread the file by using the **StructureTools > Read Application Definitions** command.

The **Read Application Definitions** command re-reads both the `structapps.fm` files and merges the application information from the two files. If an application is defined in both the user and the master copy, the application definition in the user copy is given precedence. This command replaces the current list of structure applications stored in memory with the applications defined in the current `structapps.fm` file.

Log files

FrameMaker log files give you information used to identify and correct errors in your application.

Generating log files

FrameMaker can produce a log file of errors and warnings for each file it processes. If it runs without encountering errors or other conditions requiring messages, it does not create a log file.

FrameMaker generates messages for conditions, such as:

- Markup syntax errors
- Read/write rule syntax errors
- Missing or otherwise unavailable files
- Missing entities
- Inconsistencies between read/write rules and the relevant EDD or DTD
- XSLT errors and XSLT messages.

Messages in a log file

Messages written to the log file include warnings, errors, and fatal errors.

Warnings are notifications of potential problems; a warning does not necessarily mean something is wrong with the input. For example, when the software creates an EDD from a DTD, it issues a warning if it encounters an `element` rule in a read/write rules document for a generic identifier that doesn't exist in the DTD. This situation is legal, but the software warns you in case you have misspelled the generic identifier in the rules document. For all warnings, the software writes a message and continues processing.

Errors indicate actual problems in the processing. For example, when FrameMaker updates an existing EDD, it reports an error for a FrameMaker element definition that has no counterpart in the DTD. For most errors, the software writes a message and continues processing.

The third category of message indicates *fatal errors*. These errors cause processing to stop. For example, if FrameMaker encounters a syntax error in a rules document, it stops processing.

To aid in debugging your structure application, FrameMaker provides a facility for locating certain problems. Choose **StructureTools > Check Read/Write Rules**. FrameMaker uses the current

application to check the validity of the rules document in that application. The command can find many potential problems and most fatal errors.

Using hypertext links

When FrameMaker creates a log file, where possible it includes hypertext links as an additional debugging tool. Each message in a log file has one of the forms shown in the following examples:

```
-> /usr/vpg/ch1.fm;
   Invalid property specified for element "AFrame".
```

or

```
/usr/vpg/ch1.sgm; line 25;
Required attribute "lang" is missing for element "list".
```

The first part of the message indicates the location of the problem in the source document; it always includes the source document's filename. If the source is a FrameMaker document, such as a rules document or a FrameMaker document being imported or exported, there is an arrow to the left as in the first example. The arrow indicates a hypertext link. If you activate the link, the software opens the document to the page containing the problem. If the source isn't a FrameMaker document, there is no hypertext link. In this situation, the description includes the line number in the file, as in the second example. You must open the file in an appropriate editor.

The second part of the message describes the particular problem encountered. This part of the message always contains a hypertext link to an explanation of the message.

Setting the length of a log file

Processing documents can produce extremely large log files. In practice, you are unlikely to look at all pages of an extremely large log file. For this reason and to save space and time, FrameMaker limits the number of messages it sends to the log file for a given document. For information on how to change that number, see Developer Reference, page 30: Limiting the length of a log file.

Other special files

There are several other file types you must work with in creating a structure application. These files are discussed in other chapters.

- For information on creating an EDD, see Part III, "Working with an EDD."
- For information on creating read/write rules files, see Chapter 19, "Read/Write Rules and Their Syntax."
- For information on ISO public entity set files, see Developer Reference, Chapter 10, ISO Public Entities

Part III Working with an EDD

Part II explains how to develop an element definition document (EDD) and define elements in it. If you're developing an EDD as part of a larger structure application, you should be familiar with Part II, "Developing a FrameMaker Structure Application," before using the material in this part.

The chapters in this part are:

- Chapter 12, "Developing an Element Definition Document (EDD)"

  Discusses the process of developing an EDD—from creating a new EDD through building a structured template from your element definitions. Read this chapter first for an overall understanding of the process.

  This chapter lists all elements in an EDD's Element Catalog. It provides links to sections where the elements are covered in the syntax chapters, which describe specific types of rules, and show examples of these constructs in element definitions:

  –Chapter 13, "Structure Rules for Containers, Tables, and Footnotes"

  –Chapter 14, "Attribute Definitions"

  –Chapter 15, "Text Format Rules for Containers, Tables, and Footnotes"

  –Chapter 16, "Object Format Rules"

  –Chapter 17, "Context Specification Rules"

11 Developing an Element Definition Document (EDD)

An *element definition document* (EDD) contains the structure rules, attribute definitions, and format rules for all of the elements in a group of FrameMaker documents. You write and maintain the definitions in the EDD and then convert them to an Element Catalog in a structured template. To work with the elements, end users create documents from the template or import the formats from an existing document into a new document.

You can start a new EDD in two ways. If you have a DTD that your end users' documents will follow, you can begin with the DTD and create an EDD with element definitions that correspond to constructs in the DTD. If you do not have a DTD, or if you do not plan to translate structured documents to markup, you can create an EDD and write its definitions entirely in FrameMaker.

An EDD is a regular structured FrameMaker document—it has an Element Catalog already set up with everything you need to define elements for end-users' documents. When developing the EDD, you insert elements from the catalog and in most cases provide values to fill out the definitions. For general information on working in structured documents, see the *Using FrameMaker*.

In this chapter

This chapter explains the process of developing an EDD and provides a summary of the elements in the EDD's Element Catalog. It contains these sections.

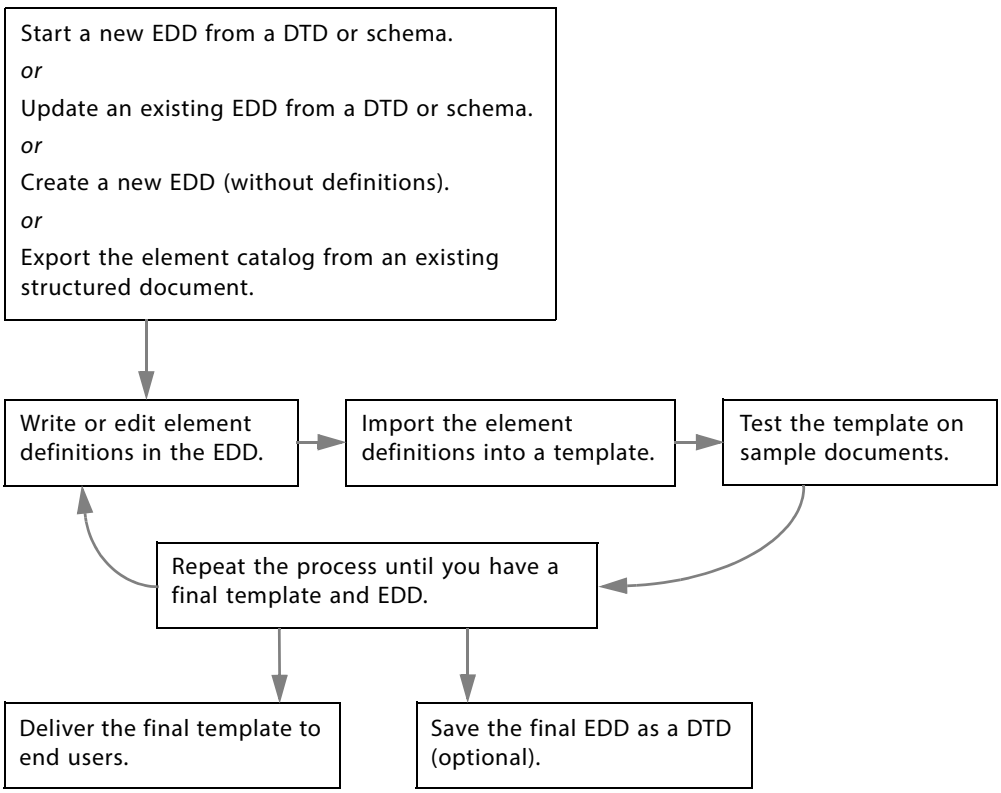
–"Creating an Element Catalog in a template" on page 166
–"Saving an EDD as a DTD for export" on page 168

- Where to find structured documents and EDDs to review:

–"Sample documents and EDDs" on page 171

Overview of the development process

These are the basic steps you go through to create an EDD and to develop it as part of a workable template for your end users:



If you start the process with a DTD, FrameMaker creates a new EDD with definitions that correspond to the declarations in the DTD. In most cases, as part of this conversion you need to analyze the EDD and DTD together, develop read/write rules, and repeat the process until the translation is complete. You'll probably at least need to add format rules to the definitions in the EDD.

If you do not have a DTD, you can either export the Element Catalog from an existing FrameMaker document to create a new EDD or create a new empty EDD by selecting **StructureTools > New**

EDD. Then you either edit the definitions of the exported elements or write new element definitions from scratch.

When you're finished writing and editing element definitions in the EDD, you import the definitions into a FrameMaker template, which also stores formats that may be referenced in your format rules. (You may need to coordinate this part of the process with a template designer responsible for formatting information.) Then you test the template on sample XML or SGML documents, revise the definitions in the EDD, reimport the definitions into the template, and repeat the process as necessary until you have a template that works the way you want it to.

Finally, you deliver the structured template to end users—in many cases, along with other pieces of a larger XML or SGML application. If you didn't start from a DTD but your users will export documents to markup data, you also need to save the EDD as a DTD.

At the end of the process, even though you are finished developing the EDD, you should normally keep it as a separate document to use for maintaining the element definitions.

Creating or updating an EDD from a DTD

If your documents need to conform to a DTD, you can use the DTD as a starting point for your EDD. FrameMaker creates a new EDD with element and attribute definitions that correspond to element declarations and attribute definition list declarations in the DTD. If the declarations in the DTD change, you can update the EDD to match.

About the DTD

A DTD is required in the prolog of an SGML document, and may be in the prolog of an XML document. The DTD provides element and attribute definition list declarations. The markup document can contain these declarations directly, or it can have an identifier that references a set of declarations stored separately in an external entity. This entity is sometimes called an *external DTD subset*.

If you start an EDD from a DTD, the DTD you use can be either a complete DTD at the beginning of a markup document or an external DTD subset stored in a separate file. In this section, the term *DTD* can mean either case.

For more information on DTDs and how they can be stored, see "XML and SGML DTDs" on page 88.

Read/write rules and the new EDD

When starting from a DTD, we recommend that you first create an initial EDD with no *read/write rules*—or with only a subset of the rules if you have some already developed. This lets you see how FrameMaker translates the DTD with little or no help from your rules.

Once you have both a DTD and an EDD, you can refine the translation in an iterative process of developing read/write rules. First analyze the DTD and new EDD together to plan how to modify

the translation with rules. Then develop at least some of your rules, update the EDD from the DTD using the rules (see "Updating an EDD from a DTD" on page 142), test the results on sample markup documents, and repeat the process as many times as necessary. You may find it easiest to write and test only a few rules during each iteration. For a more detailed discussion of this process, see "Task 3. Creating read/write rules" on page 114.

You develop read/write rules in a special rules document that is part of a structure application. When you create an EDD from a DTD, you can specify which application (and hence which set of rules) to use with the EDD. For information on developing a read/write rules document, see Chapter 19, "Read/Write Rules and Their Syntax."

An application definition file (such as `structapps.fm`) describes what files are used in each structure application you deliver to an end user. If necessary, update an application definition in this file so that the application uses the appropriate read/write rules document. To do this, insert a `ReadWriteRules` element in the definition and type the pathname of the document. For more information, see "Application definition file" on page 132.

Creating an EDD from a DTD

To create an EDD from a DTD, choose Open DTD from the File>Structure Tools submenu in any open FrameMaker document. Select the DTD in the Open DTD dialog box, and then select an application in the Use Application dialog box.

The name of the application you select is stored in an `SGMLApplication` or `XMLApplication` element in the EDD for future updates and exports. All FrameMaker documents with an Element Catalog derived from the EDD use the application by default.

If you are creating an initial EDD without read/write rules, select <No Application>. This specifies a default structure application with no rules. When you open a DTD with <No Application>, FrameMaker displays a dialog box with the choice to specify whether the EDD will be for an XML or an SGML To specify a structure application for the EDD later, update the EDD from the DTD (see "Updating an EDD from a DTD" on page 142), and this time select the application, or insert and fill in an `SGMLApplication` or `XMLApplication` element in the EDD. For information on filling in the element, see "Setting a structure application" on page 153.

What happens during translation

A DTD has element and attribute definition list declarations that correspond to element and attribute definitions in an EDD. When you translate a DTD to an EDD, FrameMaker makes assumptions about how the constructs from the DTD should be represented. FrameMaker reads through the entire DTD, processing elements and their attributes one at a time. In the absence of read/write rules, the software translates an element declaration from the DTD to a FrameMaker element definition of the same name, and it produces an attribute definition for each attribute defined for the element.

Note that DTDs contain syntactic information about the structure of a class of documents, but they do not address the semantics of elements they define. For example, DTDs do not distinguish

between an element used to define an equation and one used to define a marker. For this reason, the default translation may not convert all of the markup elements correctly. (An exception to this is *CALS* tables. If your DTD uses the CALS table model, FrameMaker does recognize those elements as table elements.) You can modify the default translation using read/write rules.

For details on the translation of each type of element, see Part IV, "Translating Between Markup Data and FrameMaker."

Updating an EDD from a DTD

These are two of the reasons you may need to update an EDD:

- If you started the EDD from a DTD, in most cases you need to modify the translation by developing and testing read/write rules in an iterative process. As part of each iteration, you update the EDD using the DTD and your current set of read/write rules.
- If any element or attribute declarations in the DTD change, you update the EDD to revise the corresponding definitions in the EDD.

To update an EDD from a DTD, choose Import DTD from the File>Structure Tools submenu in the EDD. Select the DTD in the Import DTD dialog box. If the Use Application dialog box appears, select a structure application for the EDD. (Use Application appears only if no application is specified in the EDD.)

In the updated EDD, FrameMaker adds definitions for new elements from the DTD, removes definitions for elements that are no longer defined, and revises the content rules and attribute definitions for the remaining elements to match changes in the DTD and the current read/write rules. Any format rules and comments in the EDD are not affected, except for those in definitions that have been removed. (The software records these changes in a log file.) You can save the modified EDD if you want to keep the changes.

Log files for a translated DTD

If FrameMaker encounters any problems while starting or updating an EDD from a DTD, it produces a log file of warnings and errors. A warning is a notification of a potential problem, but it does not necessarily mean something is wrong with the DTD or the resulting EDD. An error indicates an actual problem in the processing; some errors can cause the processing to stop altogether.

A log file can have warning messages for conditions such as name changes, and it can have error messages for markup syntax errors, read/write rule errors, and missing files. If you're updating an EDD from a DTD, the log file also includes a list of changes made to the EDD and may include error messages for inconsistencies between the DTD and the EDD.

This is an example of a message in a log file:

```
/usr/struct/tutorial/chapter.dtd; line 63
Parameter entity name longer than (NAMELEN-1); truncated
```

The first line in the message gives the location of the problem in the DTD. The second line describes the problem; you can click this line to see a longer explanation.

A log file is initially locked so that you can click in it to use the hypertext links. If you want to save the log file, you must first unlock it by pressing Esc Flk. (Press Esc Flk again to relock the file.) For general information on FrameMaker log files, see "Log files" on page 134.

Creating or updating an EDD from an XML Schema

If your XML documents are associated with an XML Schema, you can generate a DTD from the Schema, then use the DTD as a starting point for your EDD. FrameMaker maps the elements of the Schema into DTD, and you can continue from there as described in "Creating or updating an EDD from a DTD," (the previous section). If the declarations in the Schema change, you can update the DTD and then the EDD to match.

For details on the translation of each type of element, see Part IV, "Translating Between Markup Data and FrameMaker." For a complete listing of how Schema elements are mapped into DTD elements, see Developer Reference, Chapter 6, XML Schema to DTD Mapping

Starting an EDD without using a DTD

If you do not have a DTD, or if you do not plan to translate structured documents to markup, you can start an EDD without using a DTD. You either create a new EDD and define the elements from scratch or create an EDD that has the element definitions from an existing FrameMaker document. When you start an EDD without a DTD, you can still save the EDD as a DTD later.

If you plan to translate documents to XML or SGML but do not yet have a DTD, we recommend that you begin with an EDD and then continue the development process from there. An EDD has richer semantics than a DTD and a set of tools that facilitate defining elements, so you will probably find it easier to develop an environment for your end users in an EDD.

Creating a new EDD

You can create a new EDD and enter all of the element definitions from scratch. The new EDD has a highest-level element called `ElementCatalog`, a `Version` element with the number of the current FrameMaker release, and one empty `Element` element.

To create a new EDD, choose **New EDD** from the **StructureTools** menu.

Exporting an Element Catalog to a new EDD

You can export the Element Catalog of an existing structured document to create an EDD. The new EDD has:

- a highest-level element called `ElementCatalog`

- a `Version` element with the number of the current FrameMaker release
- a `Para` element with the name of the structured document and the current date and time
- element definitions for all of the elements from the document's catalog

Exporting an Element Catalog is helpful when you already have a structured document that you'd like to use as a basis for other documents. You will probably need to add or edit element definitions in the new EDD.

To export an Element Catalog to a new EDD, choose Export Element Catalog as EDD from the File>Developer Tools menu in the structured document with the Element Catalog.

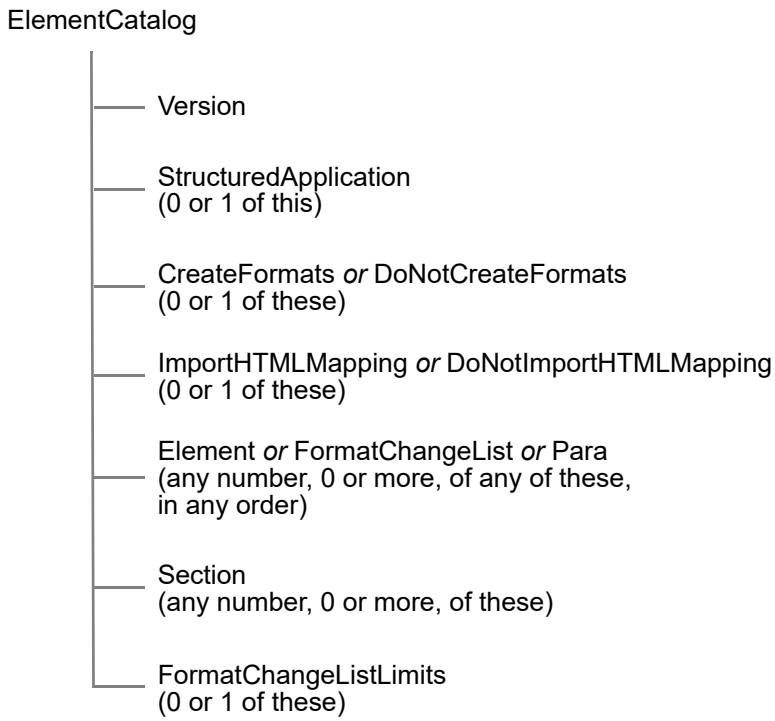
The exported EDD and the EDD from which it was created are equivalent in that they have the same element definitions. The two EDDs may differ, however, in the order and grouping of definitions. `Section`, `Head`, and `Para` elements from the original EDD are also not preserved in the exported EDD.

The Element Catalog in an EDD

An EDD is a regular structured FrameMaker document. It comes with an Element Catalog that has everything you need to define elements and to specify related information for your end users' documents. You insert elements from the catalog as you do in any other structured document, and in most cases provide values to fill out the rules.

High-level elements

The highest-level element in an EDD is `ElementCatalog`. It can have the following child elements, in the order shown. Only the `Version` element is required. `ElementCatalog` and the required `Version` are inserted automatically, along with the optional `CreateFormats` and `Element`, when you start a new EDD.



The `Version` element displays the number of the FrameMaker release used when the EDD was created; the number is not editable. The following child elements are optional:

- `StructuredApplication`: Specifies an SGML or XML application for the EDD and for documents that use the EDD. You need to type the name of the application. See "Setting a structure application" on page 153
- `CreateFormats` or `DoNotCreateFormats`: Specifies whether or not to create formats automatically when you import element definitions into a template or other document. `CreateFormats` is inserted automatically in a new EDD. See "Specifying whether to create formats automatically" on page 152.
- `ImportHTMLMapping` or `DoNotImportHTMLMapping`: Specifies whether or not to import the EDD's HTML mapping table into a document when you import the element definitions into a template or other document. See "Specifying whether to transfer HTML mapping tables" on page 152.
- `Element`: Begins the definition of an element. See "Writing element definitions" on page 155.
- `FormatChangeList`: Begins the definition of a named format change list. You can use one of these lists to describe a set of changes to format properties and then refer to the list from element definitions. This is helpful when two or more elements use the same set of changes

because you need to describe the changes only once. See "Defining a format change list" on page 234.

- `FormatChangeListLimits`: Begins a specification of minimum and maximum limits on relative values used in format change lists and text format rules in the EDD. See "Setting minimum and maximum limits on properties" on page 235.
- `Para`: Inserts a paragraph so that you can comment the EDD. This is useful for adding descriptions of groups of elements or sections of the EDD. See "Organizing and commenting an EDD" on page 153.
- `Section`: Creates a section so that you can group element definitions or format change lists in the EDD. By using sections, you can divide a large EDD into more manageable parts. A `Head` child element is inserted automatically with `Section`. See "Organizing and commenting an EDD" on page 153.

All elements in the catalog

This section lists all of the elements in an EDD's Element Catalog in alphabetical order—except for a few very low-level elements for completing a formatting specification, such as `Left` and `Right` (for `Alignment`) and `Yes` and `No` (for `ChangeBars`). Click an element tag to go to the discussion of the element in one of the EDD chapters.

Element tag	Purpose of the element
`Alignment`	Side head alignment
`AlignOn`	Character for a decimal tab stop
`AllContextsRule`	Context specification in a format rule
`AnchoredFrame`	Initial content type for a graphic element
`Angle`	Font angle
`Attribute`	Attribute definition
`AttributeList`	List of attribute definitions
`AttributeValue`	Attribute value in a prefix or suffix rule
`AutoInsertions`	Structure rule for automatically nested descendants
`AutonumberFormat`	Autonumber format
`AutonumCharFormat`	Character format for an autonumber
`Bottom`	Bottom cell margin for a table
`BottomCellMarginLimits`	Limits on all bottom cell margins
`Case`	Capitalization style
`CellMarginLimits`	Limits on all cell margins
`CellMargins`	Set of table cell margins
`ChangeBars`	Change bars in page margins
`CharacterFormatTag`	Reference to a character format in a format rule

Element tag	Purpose of the element
Choice	Attribute type
Choices	List of values for a Choice attribute
Color	Text color
Comments	Explanatory paragraph in an element definition
Container	Element type
ContextLabel	Label for a formatting variation of an element
ContextRule	Context specification in a format rule
CountAncestors	Ancestor to count in a level rule
CreateFormats	Create formats when importing an EDD
CrossReference	Element type
CrossReferenceFormat	Initial cross-reference format
Default	Default value for an attribute
DefaultPunctuation	Default punctuation for a side or run-in head
DefaultSystemVariable	Default variable for a system variable element
DoNotCreateFormats	Do not create formats when importing an EDD
Element	Element definition
ElementCatalog	Highest-level element in an EDD
ElementPgfFormatTag	Reference to a base paragraph format
Else	Clause in a context or level rule
ElseIf	Clause in a context or level rule
Equation	Element type
Exclusion	Structure rule for excluding an element from a defined element or its descendants
Family	Font family
FirstIndent	First-line paragraph indent
FirstIndentChange	First-line paragraph indent, as a change to the current indent
FirstIndentLimits	Limits on all first-line paragraph indents
FirstIndentRelative	First-line paragraph indent, relative to a left indent
FirstParagraphRules	Text format rules for first paragraph in an element
FontSizeLimits	Limits on all font sizes
Footnote	Element type
FormatChangeList	List of changes to text formatting properties
FormatChangeListLimits	List of limits on values in format change lists
FormatChangeListTag	Reference to a format change list in a format rule

Element tag	Purpose of the element
FrameAbove	Reference to a graphic frame above a paragraph
FrameBelow	Reference to a graphic frame below a paragraph
FramePosition	Position for a graphic frame
GeneralRule	Structure rule for allowed content in an element
Graphic	Element type
Head	Head for a section in an EDD
Height	Line spacing
HeightChange	Line spacing, as a change to the current height
Hyphenate	Automatic hyphenation
Hyphenation	Set of hyphenation properties
IDReference	Attribute type
IDReferences	Attribute type
If	Clause in a context or level rule
ImportedGraphicFile	Initial content type for a graphic element
Inclusion	Structure rule for allowing an element in a defined element or its descendants
Indents	Set of paragraph indents
InitialObjectFormat	Object format rule for a graphic, marker, cross-reference, or equation
InitialStructurePattern	Structure rule for automatic tagging in a table
InitialTableFormat	Object format rule for a table
InsertChild	Child element for automatic insertion
InsertNestedChild	Nested child element for automatic insertion
Integer	Attribute type
Integers	Attribute type
KeepWithNext	Keep paragraph with next paragraph
KeepWithPrevious	Keep paragraph with previous paragraph
Language	Language for hyphenation and spell-checking
Large	Initial equation size
LastParagraphRules	Text format rules for last paragraph in an element
Leader	Tab leader character
Left	Left cell margin for a table
LeftCellMarginLimits	Limits on all left cell margins
LeftIndent	Left paragraph indent

Element tag	Purpose of the element
LeftIndentChange	Left paragraph indent, as a change to the current indent
LeftIndentLimits	Limits on all left paragraph indents
LetterSpacing	Additional letter spacing to optimize word spacing
LevelRule	Level specification in a format rule
LineSpacing	Set of line spacing properties
LineSpacingLimits	Limits on all line spacing
Marker	Element type
MarkerType	Initial marker type
MaxAdjacent	Maximum adjacent hyphenated lines
Maximum	Specification for a limit on a value
Maximum	Maximum word spacing
Medium	Initial equation size
Minimum	Specification for a limit on a value
Minimum	Minimum word spacing
MoveAllTabStopsBy	Relative change position for all tab stops
Name	Attribute name
NoAdditionalFormatting	No text formatting changes
NoAutonumber	No autonumbering
OffsetHorizontal	Horizontal text range offset
OffsetVertical	Vertical text range offset
Optimum	Optimum word spacing
Optional	Optional attribute value
Outline	Outline text style
Overline	Overline text style
PairKerning	Pair kerning
Para	Explanatory paragraph in an EDD, outside an element definition
ParagraphFormatTag	Reference to a paragraph format in a format rule
ParagraphFormatting	Formatting an element as a paragraph
ParagraphSpacing	Set of paragraph spacing properties
PgfAlignment	Paragraph alignment
Placement	Paragraph placement on a page
Position	Autonumber position in a paragraph
Prefix	Text string for a prefix

Element tag	Purpose of the element
PrefixRules	Rules for a prefix
PropertiesAdvanced	Set of formatting properties for frames, hyphenation, and word spacing
PropertiesBasic	Set of formatting properties for indents, alignment, tab stops, and line and paragraph spacing
PropertiesFont	Set of formatting properties for font and text style
PropertiesNumbering	Set of formatting properties for autonumbering
PropertiesPagination	Set of formatting properties for paragraph placement
PropertiesTableCell	Set of formatting properties for table cells
Range	Range of values for a numeric attribute
ReadOnly	Read-only attribute
Real	Attribute type
Reals	Attribute type
RelativeTabStopPosition	Tab stop position, relative to a left indent
Required	Required attribute value
Right	Right cell margin for a table
RightCellMarginLimits	Limits on all right cell margins
RightIndent	Right paragraph indent
RightIndentChange	Right paragraph indent, as a change to the current value
RightIndentLimits	Limits on all right paragraph indents
Rubi	Element type
Rubi Group	Element type
Section	Section in an EDD
Shadow	Text shadowing
ShortestPrefix	Shortest prefix in a hyphenated word
ShortestSuffix	Shortest suffix in a hyphenated word
ShortestWord	Shortest hyphenated word
Size	Text size
SizeChange	Text size, as a change to the current size
Small	Initial equation size
SpaceAbove	Space above a paragraph
SpaceAboveChange	Space above a paragraph, as a change to the current spacing
SpaceAboveLimits	Limits on all space above paragraphs
SpaceBelow	Space below a paragraph

Element tag	Purpose of the element
SpaceBelowChange	Space below a paragraph, as a change to the current spacing
SpaceBelowLimits	Limits on all space below paragraphs
Specification	Child element for If, Else, or ElseIf in a context or level rule
Spread	Text spread
SpreadChange	Text spread, as a change to the current spread
StartPosition	Paragraph start position in a column
StopCountingAt	Ancestor to stop counting at in a level rule
Strikethrough	Strikethrough text style
StructuredApplication	Reference to an SGML application for an EDD
Subrule	Nested format rule
Suffix	Text string for a suffix
SuffixRules	Rules for a suffix
Superscript Subscript	Superscript or subscript text style
SystemVariable	Element type
SystemVariableFormatRule	Object format rule for a system variable
TabAlignment	Tab stop alignment
Table	Element type
TableBody	Element type
TableCell	Element type
TableFooting	Element type
TableFormat	Initial table format
TableHeading	Element type
TableRow	Element type
TableTitle	Element type
TabStop	Tab stop definition
TabStopPosition	Tab stop position
TabStopPositionLimits	Limits on all tab stop positions
TabStops	Set of tab stop definitions
Tag	Element tag
TextFormatRules	Text format rules
TextRangeFormatting	Formatting an element as a text range
Top	Top cell margin for a table
TopCellMarginLImits	Limits on all top cell margins

Element tag	Purpose of the element
Tracking	The space between characters
TrackingChange	The space added to the current tracking
Underline	Underline text style
UseSystemVariable	Variable for a system variable element
ValidHighestLevel	Validity at the highest level in a flow
Variation	Font variation
VerticalAlignment	Text alignment in a table cell
Weight	Font weight
WidowOrphanLines	Minimum lines from a paragraph that appear alone in a column
WordSpacing	Set of word spacing properties

Defining preliminary settings in an EDD

The beginning of an EDD can have three settings that define general characteristics for the EDD: Version, CreateFormats (or DoNotCreateFormats), and StructuredApplication. FrameMaker supplies the version number, which is not editable. The other settings you can define or change yourself.

Specifying whether to create formats automatically

When you import element definitions from an EDD into a template or other document, your definitions may refer to paragraph formats, character formats, table formats, or cross-reference formats that are not already in the template. You can have FrameMaker create any missing, named formats in the template when you import the definitions. (The new formats will have default properties. In many cases, you or the template designer will probably need to change the properties.)

A new EDD has a CreateFormats element, which specifies that formats will be created automatically on import. If you do not want FrameMaker to create the formats, select the CreateFormats element and change it to DoNotCreateFormats. Whichever element you use must come before any element definitions, format change lists, sections, and paragraphs.

Specifying whether to transfer HTML mapping tables

FrameMaker products can save documents as HTML. To do this, each document can have a mapping table on its HTML reference page. An EDD can also have an HTML mapping table on its EDD_HTML reference page. When importing an element definition into a document, you can have FrameMaker also import the EDD mapping table. If you are importing into a book, FrameMaker

imports the mapping table to the BookHTML reference page of the first component file in the book.

An EDD can include an `ImportHTMLMapping` element, which tells the software to import the mapping table from the EDD to the HTML reference page of any document that imports the EDD. If you do not want documents to import the EDD's mapping table, select the `ImportHTMLMapping` element and change it to the `DoNotImportHTMLMapping` element.

When exporting a document's element catalog as an EDD, if the document has a mapping table on the HTML reference page, it will export that table to the EDD, and the EDD will contain an `ImportHTMLMapping` element.

For information on HTML mapping tables, see *Using FrameMaker*.

Setting a structure application

If you are working with XML or SGML, you need to specify which structure application to associate with the EDD. A structure application defines information such as a DTD, an SGML declaration, a read/write rules document, an application definition file, entity catalogs, and a FrameMaker template (which has the elements from your EDD). FrameMaker uses the application when you translate between a DTD and an EDD and when an end user shares documents between XML or SGML and FrameMaker.

> **Important:** The DTDs for SGML and XML are significantly different. For this reason you should always use XML structure applications for XML files, and SGML structure applications for SGML files.

When you first convert a DTD to an EDD using **StructureTools > Open DTD**, you can select a structure application for the EDD. The new EDD has a `StructuredApplication` element with the name of the application. If you select `<No Application>` when you convert the DTD, the new EDD does not have this element.

To set a structure application in an EDD that does not have an application, insert a `StructuredApplication` element and type the name of the application. This element must come before any sections, paragraphs, element definitions, and format change lists. To change an application already set in an EDD, edit the name in the `StructureApplication` element.

All documents that use the EDD are also associated with the structure application. Users can change to a different application in an individual document by using the Set Structure Application command (File menu).

For information on the parts of a structure application and the process of developing one, see Chapter 10, "Creating a Structure Application."

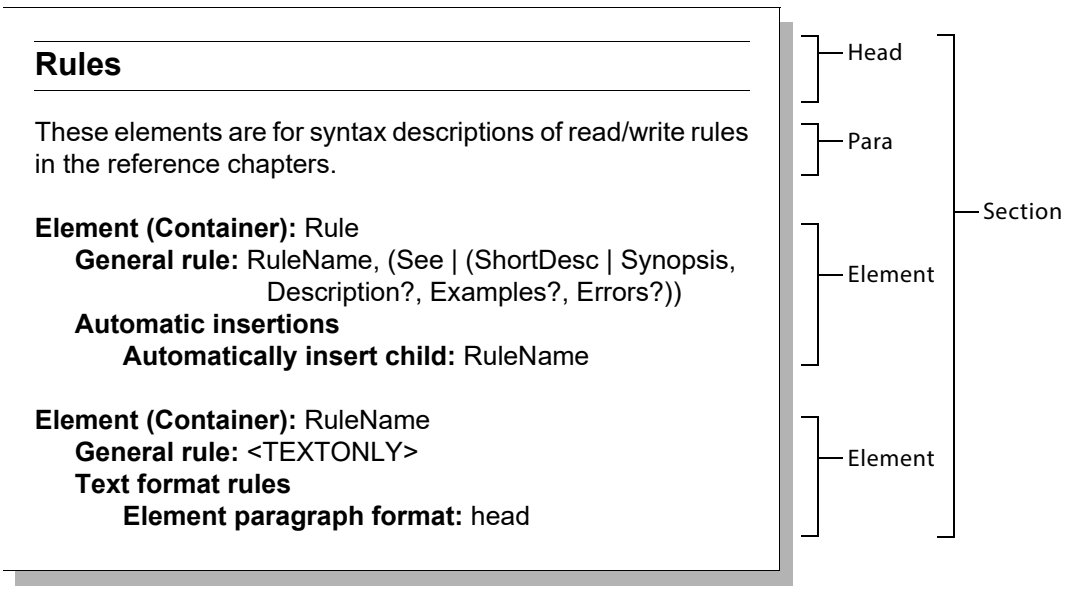
Organizing and commenting an EDD

You can add optional `Section` elements to an EDD to group element definitions and format change lists, and you can add optional `Para` elements to provide additional information about the EDD. Using a few `Section` and `Para` elements judiciously can make your EDD more readable and easier to maintain.

Sections are particularly useful when an EDD has many definitions. You may want to group and label the definitions by element type, with format change lists in another section. For example, your sections might be named `Containers`, `Tables`, `Objects`, and `Format Change Lists`.

Add explanatory paragraphs wherever needed to provide descriptive information about a group of elements. This information can be helpful to someone maintaining the EDD later. (You can also add comments to individual element definitions. See "Writing element definitions," next.)

This example shows a part of an EDD that is organized in a section and has an introductory paragraph for the section:



To add a section, insert a `Section` element in `ElementCatalog` or in another `Section`. A `Head` child element is inserted automatically. Type a label for the section in the `Head` element. You can insert `Element`, `FormatChangeList`, `Para`, and other `Section` elements as child elements of a parent `Section` element. If necessary, wrap or move existing elements into the section.

To add a paragraph, insert a `Para` element in `ElementCatalog` or in a `Section`, and type the paragraph.

Section and Para elements are ignored when you import element definitions into a template or other document. When you save an EDD as a DTD, they are written as comments.

Writing element definitions

To write an element definition, begin by inserting an element called Element in ElementCatalog or in a Section. Then use the Element Catalog or the status bar as a guide as you type text and insert child elements to create a valid definition.

Elements in FrameMaker fall into two basic groups: containers, tables, and footnotes; and object elements such as markers and equations. Containers, tables, and footnotes can hold text and other elements, whereas an object element holds one of its specified type of object and nothing more. These differences are reflected in the way you write the element definitions:

- Containers, tables, and footnotes must have content rules that define valid contents for the element. Object elements do not have content rules.
- Containers, tables, and footnotes can have text format rules that specify font and paragraph formatting for text in the element and its descendants. Object elements can have an object format rule that specifies a single property, such as a marker type or an equation size. (A table can have both text format rules and an object format rule.)

In other respects, element definitions are alike for the two groups of elements. They must all have a unique element tag and a declared type, and they can have attribute definitions and comments.

About element tags

When naming an element, give the element a tag that is self-explanatory and unique. A user will need to recognize the purpose of the element to select it in the Element Catalog and use it properly. Element tags are case-sensitive, and they can contain white space but none of these special characters:

```
( ) & | , * + ? < > % [ ] = ! ; : { } "
```

An element tag can have up to 255 characters in FrameMaker, but you should try to keep the tags concise. The default width of the Element Catalog a user sees shows the first 14 characters of a tag. (If you are using context labels, the maximum length is 255 for the tag and label together.)

The Element Catalog in a document shows the currently available elements in alphabetical order (unless the end user is displaying a customized list). In some cases, especially if the list of elements is long, you may want to name elements in a way that will group them logically in the catalog. For example, if you have two types of table elements you might name them TblSamples and TblStandard to display them together in the catalog. If you do begin any tags the same way, keep the first part of the tag as short as possible.

Don't begin tags the same way unless you need to for grouping. A user can usually find elements in the catalog if the tags are distinct, and the user may want to type in a unique beginning string to identify a tag for a quick key command (such as Control-1 for Insert Element).

> **Note: SGML:** If you plan to export documents to SGML, define element tags that conform to the naming rules and the maximum name length permitted by the concrete syntax you'll be using in SGML. If you prefer tags that do not adhere to the SGML conventions, you can provide read/write rules to convert them to SGML equivalents when you export. For more information on element names in SGML, see "Naming elements and attributes" on page 296.

Guidelines for writing element definitions

Here are a few points to keep in mind when writing element definitions:

- In most cases, you should work iteratively in the EDD. Write at least some of the definitions, import the definitions into a template, test the template on sample documents, and repeat the process as necessary.

  For information on importing and testing the definitions, see "Creating an Element Catalog in a template" on page 166.

- In many EDDs, the most complicated part of the definitions is the format rules (particularly text format rules). The first time you work in a particular EDD, consider defining just the structure rules and attribute definitions and testing only the structure and attributes at that point. Then you can go back and add the format rules and test them in a separate pass.

- Provide a highest-level element for each structured flow possible in the documents. For a book file, provide a highest-level element for the book and for each possible book component (such as Front, Chapter, and Index).

- After writing element definitions, validate the EDD before importing the definitions so that you find missing elements and content errors.

- Remember that the formats you refer to in element definitions must be stored in the template. If you are working with a template designer, you need to coordinate the tags and properties of the formats with the designer.

- Create user variables for text in the EDD that you use again and again. For example, if several elements have the same general rule, define a variable for the general rule. Then, if necessary, you can change the general rules for all the elements by redefining the variable definition.

See also "Keyboard shortcuts for working in an EDD" on page 165.

Defining a container, table or footnote element

Containers are general-purpose elements that you define for text, child elements, or a combination of the two. Paragraphs, text ranges, heads, sections, and chapters are common examples of containers. In a typical document, most elements are containers.

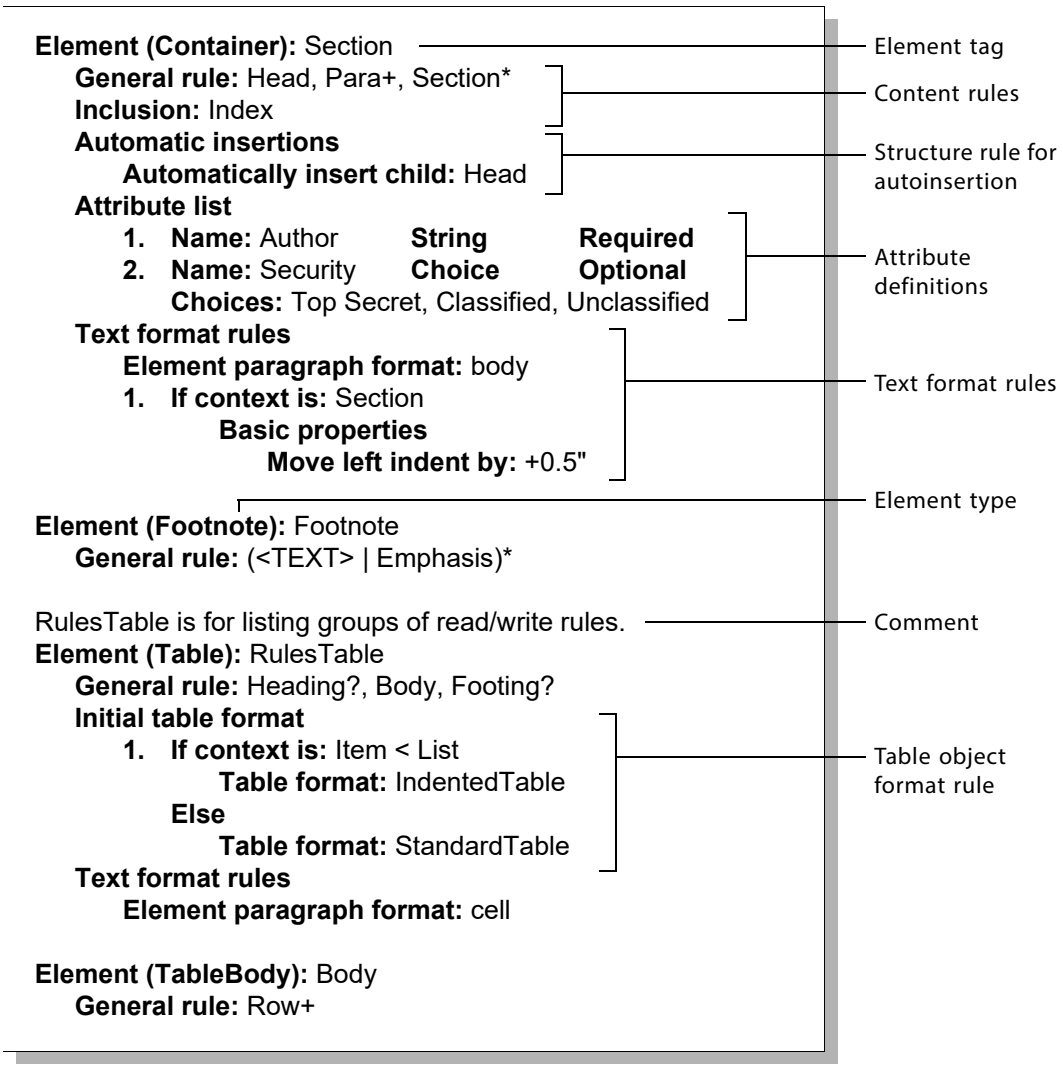
Tables, table parts (titles, headings, bodies, footings, rows, and cells), and footnotes are similar to containers in that they can hold child elements and in some cases text. But these elements are for a specific purpose—for a table or a footnote—and can be used only for that purpose in a document.

An element definition for a container, table, table part, or footnote specifies a unique element tag and element type and can also have any of these items:

- A comment that describes the element
- Content rules that describe valid contents for the element or its descendants (the general rule part of this is required). For each structured flow in the documents, at least one container needs a rule specifying that the element is valid at the highest level
- For a container, additional structure rules that provide initial contents for new instances of the element. For a table, a tagging pattern that specifies the element tags assigned to the rows and cells an end user creates with a new table
- Attribute definitions that specify attributes to store descriptive information with the element
- Text format rules that determine how to format text in the element or its descendants
- For a table, an object format rule that determines an initial table format for new instances of the element

Examples

These are definitions for containers, tables, table parts, and footnotes:



Basic steps

This section gives an overview of the steps for defining a container, table, table part, or footnote. Refer to the chapters that follow for detailed descriptions of the syntax and more examples.

Note that the steps in this section suggest an order for the rules in a definition, but in some cases you can write the rules in a different order. (For example, the element for highest-level validity can go before or after the general rule.) Use the Element Catalog as a guide for inserting elements in a valid order.

1. Insert an `Element` element in the highest-level element of the EDD (`ElementCatalog`) or in a `Section`. Then type a tag in the `Tag` element.

When you insert `Element`, the `Tag` child element is inserted automatically. For guidelines on providing a tag, see "About element tags" on page 155.

2. **(Optional) If you want to include a comment for the definition, insert a `Comments` element before `Tag` and type the comment.**

   The comment appears just above the definition's tag line. If you include a comment, place the insertion point right after `Tag` when you're finished.

3. **Insert an element to specify the element type.**

   An element type determines what other child elements will be available as you write the definition.

For this type	Insert the element
General-purpose element for holding text, child elements, or both	`Container`
Table	`Table`
Element for holding rows in a structured table	`TableHeading`, `TableBody`, or `TableFooting`
Element for holding cells in a structured table	`TableRow`
Element for holding text in a structured table	`TableTitle` or `TableCell`
Footnote	`Footnote`

   When you insert one of these elements, the name of the type appears in parentheses before the tag and a `GeneralRule` child element is inserted automatically.

4. **Type the general rule in the `GeneralRule` element to define allowed contents for the element.**

   A general rule describes the child elements the element can contain, whether the child elements are required or optional, and the order in which the child elements can occur. It also specifies whether the element can have text.

   If you do not specify a general rule, FrameMaker gives the element a default general rule that depends on the element's type. To use a default rule, leave the `GeneralRule` element empty (but do not delete `GeneralRule` or the definition will be invalid). These are the default general rules:

Element type	Default general rule
Container	`<ANY>`
Table	`TITLE?, HEADING?, BODY, FOOTING?`
Table heading, body, or footing	`ROW+`
Table row	`CELL+`
Footnote, table title, or table cell	`<TEXT>`

For information on the syntax and restrictions of general rules, see "Writing an EDD general rule" on page 174.

5.(Optional) Define other content rules as necessary.

Every structured flow in a document needs one highest-level container element. If the element you're defining is a container, you can insert a `ValidHighestLevel` child element to allow the element to be at the highest level. For more information, see "Specifying validity at the highest level in a flow" on page 179.

SGML, only: For a container, table, table part, or footnote, you can define inclusions and exclusions. An inclusion is an element that can occur anywhere inside the defined element or its descendants, and an exclusion is an element that cannot occur anywhere in the element or its descendants. For each element you want to include or exclude, insert an `Inclusion` or `Exclusion` element and type the element tag. For more information, see "Adding inclusions and exclusions" on page 179.

6.(Optional) Write additional structure rules to specify initial contents or tagging for new instances of the element.

For a container, you can define nested descendants that will appear automatically with the element in a document. Insert an `AutoInsertions` element, and for the first child insert an `InsertChild` element and type the element tag. Then for each nested descendant, insert an `InsertNestedChild` element and type the tag. For more information, see "Inserting descendants automatically in containers" on page 182.

For a table, heading, body, footing, or row, you can define element tags that will be used in row or cell elements in the table or table part. Insert an `InitialStructurePattern` element, and then type the tags of the child elements, separated by commas. For more information, see "Inserting table parts automatically in tables" on page 183.

7.(Optional) Write attribute definitions to define attributes that can store additional information about instances of the element.

In FrameMaker, attributes can be used to record information such as the current status of an element, to maintain IDs and ID references for cross-referencing between elements, and to allow an element to be formatted using the current value of its attribute.

Insert an `AttributeList` element. The first `Attribute` child element is inserted automatically. Define the first attribute, and then insert and define additional `Attribute` elements as necessary. For more information, see Chapter 14, "Attribute Definitions."

8.(Optional) Write text format rules to describe how to format text in the element or its descendants.

Text format rules can refer to a paragraph format to use as a "base" format for the element and can specify context-dependent changes to the format in use. If you write text format rules for a table, heading, body, footing, or row, the rules specify formatting only for text in descendant titles and cells.

Insert a `TextFormatRules` element. Then to specify a paragraph format, insert an `ElementPgfFormatTag` element as the first child element of `TextFormatRules` and

type the format tag. For each set of formatting changes, insert a context element (`AllContextsRule`, `ContextRule`, or `LevelRule`), specify the context, and define the changes for the context.

For containers, you can also write text format rules for the first and last paragraphs in the element, and you can define and format a prefix or suffix to appear at the beginning or end of the element. Insert `FirstParagraphRules`, `LastParagraphRules`, `PrefixRules`, or `SuffixRules`, and define the context and formatting specifications.

For information on the syntax of text format rules and how rules can be inherited from ancestors, see Chapter 15, "Text Format Rules for Containers, Tables, and Footnotes,"

9.(Optional) If the element is a table, write an object format rule to define an initial table format for new instances of the table.

A table format determines the basic appearance of the table—such as indentation and alignment, margins and shading in cells, and ruling between columns and rows.

Insert an `InitialTableFormat` element, and insert and define context elements as necessary. Type the tag of a table format for each context. For more information, see "Setting a table format" on page 242.

Defining a Rubi group element

Documents that include Japanese text most likely require Rubi to express the pronunciation of certain words. A *Rubi group* is an element that contains such text. The Rubi group includes a Rubi Group element for the base word (Oyamoji), and a Rubi element for the phoenetic spelling (Rubi) to the Oyamoji. These elements can be used only for a Rubi group.

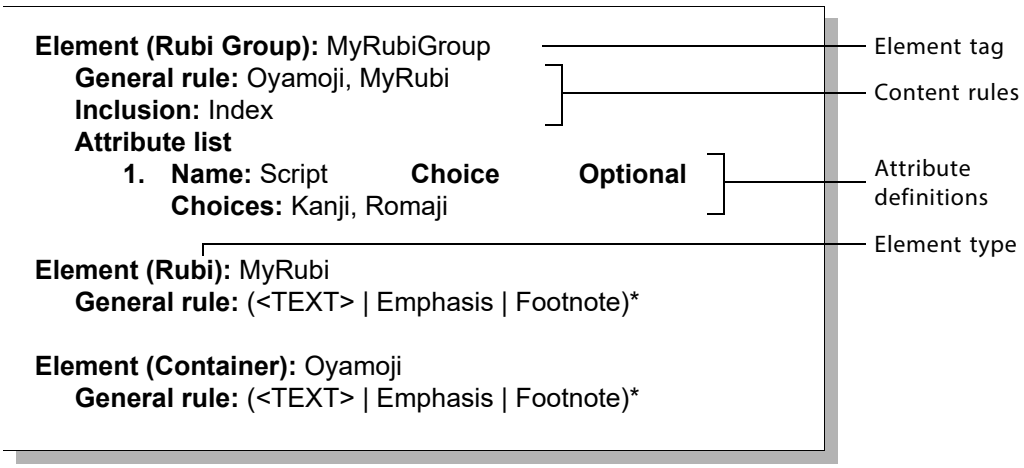
The Rubi element must be the last child element of the Rubi Group element. The Rubi Group and Rubi elements can contain other child elements. For example, the Oyamoji can be in a container of its own, or you can insert footnotes within Oyamoji or Rubi text. Note that the JIS rules for a Rubi group state that the group cannot extend across a line break. Therefore, the children of either the Rubi Group or the Rubi elements should be text range elements; FrameMaker will not insert a line break or a paragraph break within a Rubi Group.

An element definition for a Rubi group specifies a unique element tag and an element type for both the Rubi group and the Rubi text, and they can also have any of these items:

- A comment that describes the element
- Content rules that describe valid contents for the element or its descendants (the general rule part of this is required). A Rubi Group element cannot be valid at the highest level.
- For a Rubi group, an initial structure pattern that specifies the element tag assigned to the child Rubi element
- Attribute definitions that specify attributes to store descriptive information with the element
- Text format rules that determine how to format text in the element or its descendants

Examples

These are definitions for a Rubi group, a Rubi element, and a container for Oyamoji text:



Basic steps

The steps for creating a Rubi Group element and a Rubi element are very much the same as the steps for defining a container, table, table part, or footnote element. Use the Element Catalog as a guide for inserting elements in a valid order. Refer to the chapters that follow for detailed descriptions of the syntax and more examples.

Note that you cannot insert an `AutoInsertions` element to specify auto insertions for a Rubi group, but you can insert an `InitialStructurePattern` element, and then type the tag of the child Rubi element. For more information, see "Inserting Rubi elements automatically in Rubi groups" on page 186.

Defining an object element

A FrameMaker document uses special object elements for tables, graphics, markers, cross-references, equations, system variables, footnotes, Rubi groups, and Rubi text. An instance of the element holds exactly one of the specified objects.

This section explains how to define object elements except for tables. Because tables can hold other elements, they are similar to containers and are described in "Defining a container, table or footnote element" on page 156.

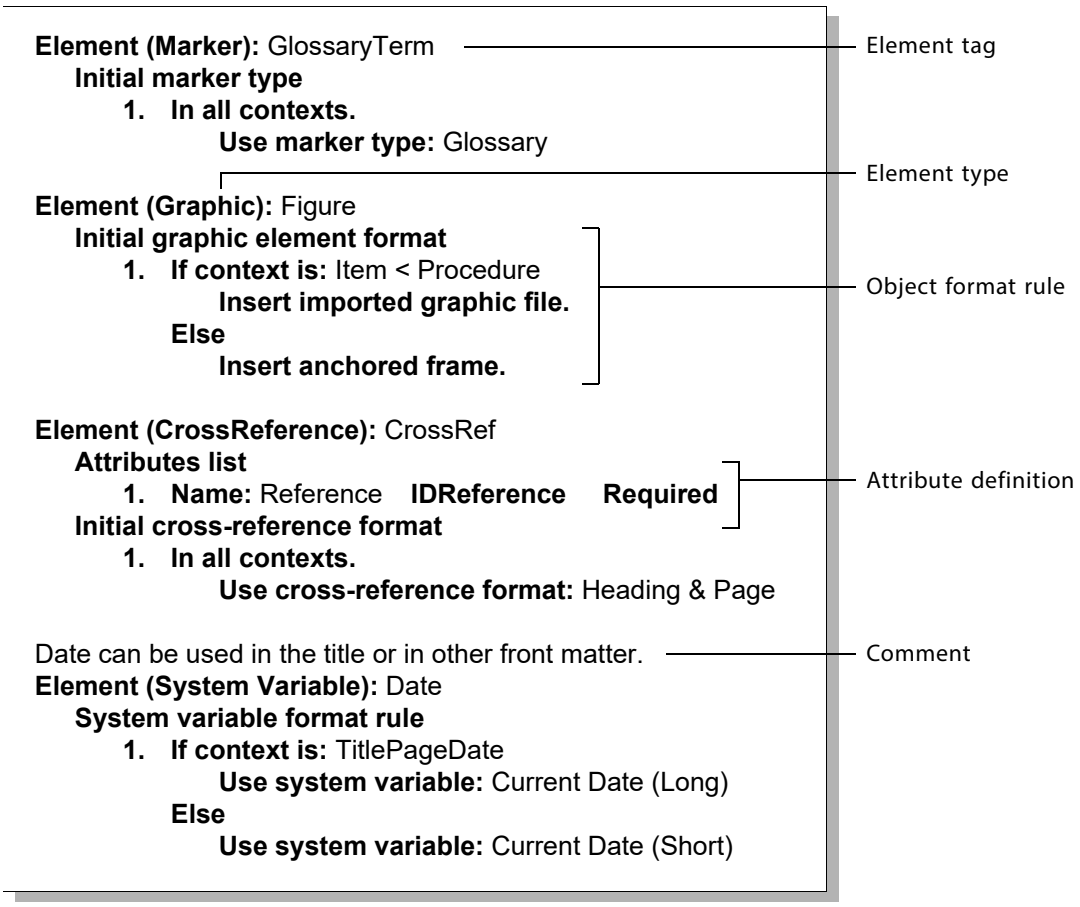
An element definition for a FrameMaker object specifies a unique element tag and element type and can also have any of these items:

- A comment that describes the element
- Attribute definitions that specify attributes to store descriptive information with the element

- An object format rule that determines a formatting property, such as a marker type or equation size, for new instances of the element

Examples

These are definitions for object elements:



Basic steps

This section gives an overview of the steps for defining an object element. Refer to the chapters that follow for detailed descriptions of the syntax and more examples.

1. Insert an `Element` element in the highest-level element of the EDD (`ElementCatalog`) or in a `Section`. Then type a tag in the `Tag` element.

When you insert `Element`, the `Tag` child element is inserted automatically. For guidelines on providing a tag, see "About element tags" on page 155.

2. (Optional) If you want to include a comment for the definition, insert a `Comments` element before `Tag` and type the comment.

The comment appears just above the definition's tag line. If you include a comment, place the insertion point right after `Tag` when you're finished.

3.Insert an element to specify the element type.

An element type determines what other child elements will be available as you write the definition.

For this object	Insert the element
Anchored frame	`Graphic`
Cross-reference	`CrossReference`
Equation	`Equation`
Imported graphic file	`Graphic`
Marker	`Marker`
System variable	`SystemVariable`

When you insert one of these elements, the name of the type appears in parentheses before the tag.

4.(Optional) Write attribute definitions to define attributes that can store additional information about instances of the element.

In FrameMaker, attributes can be used to record information such as the current status of an element, to maintain IDs and ID references for cross-referencing between elements, and to allow an element to be formatted using the current value of its attribute.

Insert an `AttributeList` element. The first `Attribute` child element is inserted automatically. Define the first attribute, and then insert and define additional Attribute elements as necessary. For more information, see Chapter 14, "Attribute Definitions."

5.(Optional) Write an object format rule to define a formatting property for new instances of the element.

These are the properties you can specify in an object format rule:

Element type	Property
Graphic	Anchored frame or imported graphic file
Cross-reference	Cross-reference format
Equation	Equation size
Marker	Marker type
System variable	System variable name

With the exception of system variable names, these properties are not binding. An end user can change a marker type, cross-reference format, or other property at any time, and the change is not considered to be a format rule override. (The user can remove all format rule overrides when he or she reimports element definitions.) A user cannot change the variable name for a system variable element.

Unlike text format rules, an object format rule defines the property only for the current element. Because object elements do not have descendants, the object rule is not passed on through a hierarchy to other elements.

Insert an `InitialObjectFormat` or `SystemVariableFormatRule` element, and insert and define context elements as necessary. Define a formatting property for each context. For more information, see Chapter 16, "Object Format Rules."

Keyboard shortcuts for working in an EDD

This section gives some of the most useful keyboard shortcuts for working in an EDD (or in any other structured document). For a complete list of the shortcuts available, see the FrameMaker quick reference.

Editing structure

Use these shortcuts for editing the structure of an EDD—for example, inserting, wrapping, changing, and rearranging elements (Note: for these shortcuts to function, an element must already be selected).

To	Press
Insert an element at the current location	Control-1 (one) or Esc Ei
Wrap an element around the current selection	Control-2 or Esc Ew
Change the current element	Control-3 or Esc Ec
Unwrap the current element	Esc Eu
Merge into the first element (when more than one element is selected)	Esc Em
Merge into the last element (when more than one element is selected)	Esc EM
Split the current element	Esc Es
Repeat the last insert, wrap, or change element command	Esc ee
Move the element to the next higher level (promote)	Esc EP
Move the element to the next lower level (demote)	Esc ED
Collapse or expand the current element in the Structure View	Esc Ex
Collapse or expand the element's siblings in the Structure View	Esc EX
Transpose the current element with the previous one	Esc ET
Transpose the current element with the next one	Esc Et

Moving around the structure

Use these shortcuts to move the insertion point around the structure of an EDD.

To move the insertion point	Press
To the next element down in the structure	Esc sD or Meta-down arrow
To the previous element up in the structure	Esc sU or Meta-up arrow
To the beginning of the next element's contents	Esc sN or Meta-right arrow
To the start of the current element	Esc sS
To the end of the current element	Esc sE
Just before the current element's parent	Esc sB

Creating an Element Catalog in a template

A FrameMaker template stores all of the catalogs and properties that end users need for their documents. This can include an Element Catalog built from your EDD—as well as a Paragraph Catalog, Character Catalog, Table Catalog, and a variety of other properties that affect the appearance of documents, such as page layouts and cross-reference formats. You create a template's Element Catalog by importing the definitions from the EDD into the template. (In many cases, the other catalogs and properties are set and maintained by a template designer.)

When end users need to work with the elements you've defined, they can create new documents from the template or import the Element Catalog from the template into their existing documents. Users do not normally need to see and interpret definitions in the EDD.

You can also make the template part of a structure application. To do this, insert a `Template` element in the application definition in the `structapps.fm` file and type the pathname of the template. When end users open a markup document that uses that application, FrameMaker applies structure and formatting from the template to the document. For more information, see "Application definition file" on page 132.

Many of the formats stored in a template may be used by format rules in your element definitions; for example, text format rules can refer to paragraph formats, and table object format rules can refer to table formats. If you are working with a template designer, you'll need to coordinate the tags and properties of these formats with the designer. If you are acting as template designer as well as EDD developer, you'll need to define the formats yourself. For advice on planning and designing the parts of a template, see the *Using FrameMaker*.

Even after importing element definitions into a template, you should keep the EDD as a separate file for maintenance purposes.

Importing element definitions

To import element definitions, choose **Element Definitions** from the **File > Import** submenu in the template. Select the EDD in the Import from Document dialog box. (The EDD must be open to appear in this dialog box.) If the EDD is invalid, an alert tells you to correct the validation errors and then import the EDD.

If you import definitions into a template that does not yet have an Element Catalog, FrameMaker creates a new catalog for the template. If you import definitions into a template that already has a catalog, FrameMaker completely replaces the old catalog with your new one.

Log files for imported element definitions

If FrameMaker encounters any problems while importing element definitions, it produces a log file of warnings and errors. A warning is a notification of a potential problem, but it does not necessarily mean something is wrong with the EDD. An error indicates an actual problem in the processing; some errors can cause the processing to stop altogether.

Some warning messages describe formats that FrameMaker needs to create because they are referenced in the EDD but not yet stored in the template. If you do not want FrameMaker to create formats automatically when you import element definitions, you can insert a `DoNotCreateFormats` element as a child element of `ElementCatalog` in the EDD. (If the EDD already has a `CreateFormats` element, select it and change it to `DoNotCreateFormats`.)

A log file can also have messages for:

- defined elements not used in any content rules
- elements referenced in content rules that are not defined
- definitions that have syntax errors

This is an example of a message in a log file:

```
Referenced element "Part" is undefined.
```

All messages in the log file have a hypertext link to the EDD. You can click a message to go to the line with the problem in the EDD.

A log file is initially locked so that you can click in it to use the hypertext links. If you want to save the log file, you must first unlock it by pressing Esc Flk. (Press Esc Flk again to relock the file.) Or use the **Toggle View Only** button on the formatting bar. For general information on FrameMaker log files, see "Log files" on page 134.

Debugging element definitions

You should test element definitions thoroughly before delivering a structured template to end users. First correct any errors in the EDD that are identified in the log file and reimport the

definitions until FrameMaker no longer finds errors. Then import the Element Catalog from the template into a sample document that contains representative text and check to see that the elements behave in the way you expect. You may need to test the elements in the document, make revisions in the EDD, reimport the elements into the EDD, reimport the catalog into the document, and repeat the process as necessary.

Here are a few tasks to go through while testing elements in a sample document:

- Insert and wrap a variety of elements in the document. If you've defined any containers to have child elements added automatically, make sure the child elements also appear when you insert the containers.
- Move elements around in the document. Check the **Structure View** to see that the elements are valid only where they should be.
- Enter attribute values in elements that allow them. Check the Structure View to make sure that the types of values you want to add are valid according to the attribute's definition.
- Once you have the format rules in the EDD, move elements around to verify that they are formatted correctly according to context. Check to see that extra formatting items such as prefixes and autonumbers appear where they should.

For help on identifying syntax and context errors in an element definition, see:

- "Debugging structure rules" on page 187
- "Debugging text format rules" on page 237
- "Debugging object format rules" on page 249

Saving an EDD as a DTD for export

If your end users will be saving FrameMaker documents as markup and you did not begin the development process with a DTD, you need to save your EDD as a DTD so that users will have a DTD for exported documents. FrameMaker creates a new DTD with element declarations and attribute definition list declarations that correspond to element and attribute definitions in the EDD.

You need to define a finished DTD as part of a structure application before your end users can save documents as markup. To do this, insert a `DTD` element in the application definition in `structapps.fm` and type the pathname of the DTD. For more information, see "Application definition file" on page 132.

Read/write rules and the new DTD

When creating a DTD from an EDD, we recommend that you first create an initial DTD with no read/write rules—or with only a subset of the rules if you have some already developed. This lets you see how FrameMaker translates the EDD with little or no help from your rules.

Once you have both an EDD and a DTD, you can refine the translation in an iterative process of developing read/write rules. First analyze the EDD and new DTD together to plan how to modify the translation with rules. Then develop at least some of your rules, update the DTD from the EDD using the rules, test the results on sample documents, and repeat the process as many times as necessary. You may find it easiest to write and test only a few rules during each iteration. For a more detailed discussion of this process, see "Task 3. Creating read/write rules" on page 114.

You develop read/write rules in a special rules document that is part of a structure application. When you create a DTD from an EDD, you can specify which application (and hence which set of rules) to use with the DTD. For information on developing a read/write rules document, see Chapter 19, "Read/Write Rules and Their Syntax."

Creating a DTD from an EDD

To create a DTD from an EDD, choose Save as DTD from the File>Developer Tools submenu in the EDD. Specify a location in the Save as DTD dialog box.

If the Use Application dialog box appears, select a structure application for the DTD. (Use Application appears only if no application is specified in the EDD.) If you're creating an initial DTD without read/write rules, select `<No Application>` for a default structure application with no rules. To specify structure application for the DTD later, you can use Save as DTD again and this time select the application.

What happens during translation

An EDD has element and attribute definitions that correspond to element and attribute definition list declarations in a DTD. When you translate an EDD to a DTD, FrameMaker makes assumptions about how the constructs from the EDD should be represented. FrameMaker reads through the entire EDD, processing elements and their attributes one at a time. In the absence of read/write rules, the software translates a FrameMaker element definition to an XML or SGML element declaration of the same name, and it produces an attribute list declaration for each attribute defined for the element.

FrameMaker writes other EDD constructs in various ways; for example, variables become entities and markers become processing instructions. Comments and `Section` and `Para` elements in the EDD become comments.

Note that EDDs include more semantic information about the usage of elements than DTDs do. For example, an EDD may have special element types corresponding to markers, system variables, and graphics. The declarations in a DTD created by FrameMaker reflect this information.

For details on the translation of each type of element, see "Translating Between Markup Data and FrameMaker" on page 264.

> **Important:** When exporting an EDD as a DTD, if the new DTD file has the same name as the old DTD file, FrameMaker can save a backup version of the old DTD. To use this feature, you must turn on **Automatic Backup on Save** in the **Preferences** dialog box.

SGML declarations

FrameMaker runs a new DTD through an XML or SGML parser. In the process, it may identify errors in the syntax of the DTD. Two of the most common errors are invalid markup names and an inappropriate SGML declaration. You can use read/write rules to translate FrameMaker element tags to valid markup names.

For SGML, if the default SGML declaration that FrameMaker provides is not appropriate for your DTD, you can modify the declaration to avoid capacity and quantity errors. The default SGML declaration for FrameMaker uses the reference *concrete syntax* and the reference quantity set. To change to a different declaration, insert an `SGMLDeclaration` element in the application's definition in `structapps.fm` and type the pathname for the new declaration. For a description of the default SGML declaration and the variations that FrameMaker supports, see Developer Reference, Chapter 9, SGML Declaration

If you need to reapply the parser to a DTD but not recreate the DTD, select **StructureTools > Parse Structured Document**. You must parse an XML or SGML instance that references the DTD

Log files for a translated EDD

If FrameMaker encounters any problems while creating a DTD from an EDD, it produces a log file of warnings and errors. A warning is a notification of a potential problem, but it does not necessarily mean something is wrong with the EDD or the resulting DTD. An error indicates an actual problem in the processing; some errors can cause the processing to stop altogether.

A log file can have warning messages for conditions such as name changes, and it can have error messages for FrameMaker syntax errors, read/write rule errors, capacity and quantity errors, and missing files.

This is an example of a message in a log file:

```
/usr/fmsgml/tutorial/chapter.edd; line 24
Invalid property specified for element "AFrame".
```

The first line in the message gives the location of the problem in the EDD; you can click this line to go to the problem in the EDD. The second line describes the problem; you can click this line to see a longer explanation.

A log file is initially locked so that you can click in it to use the hypertext links. If you want to save the log file, you must first unlock it by pressing Esc Flk. (Press Esc Flk again to relock the file.) Or use the **Toggle View Only** button on the formatting bar For general information on log files, see "Log files" on page 134.

Sample documents and EDDs

FrameMaker comes with several structured documents and EDDs that you can review as samples or use as a starting point for developing EDDs of your own. You'll find the files in the following directories.

```
$FMHOME/fminit/uilanguage/Samples/FMSGML/
$FMHOME/fminit/uilanguage/Samples/Templates/Structured/
```

The `FMSGML` directory contains a commented EDD for this developer's guide and one structured document that uses the EDD.

The `Structured` directory contains structured documents for outlines, reports, and viewgraphs.

12 Structure Rules for Containers, Tables, and Footnotes

Containers, tables, table parts, footnotes, and Rubi groups can all hold other elements; they build the structural hierarchy of a document. For each of these elements in an EDD, you need to define the allowable contents to describe a document structure that is valid.

Content rules are structure rules that describe allowable content in FrameMaker. These rules translate to content models and declared content in markup. If you convert a DTD to an EDD, or an EDD to a DTD, the allowed-content information for elements is preserved. You can modify some of the default translations with read/write rules.

For containers, tables, and some table parts, you can also specify a structure for a new instance of the element in a document. These rules do not describe the range of allowable contents (as content rules do), but provide a starting structure as a convenience for your end users.

In this chapter

This chapter explains how to write structure rules in element definitions for containers, tables, table parts, footnotes, and Rubi groups. It contains these sections.

Overview of EDD structure rules

All containers, tables, table parts, footnotes, and Rubi groups must have a general rule that specifies what the element is allowed to contain in a document. These elements can also have optional inclusions or exclusions that specify child elements that can or cannot occur in the element and its descendants. The general rule and inclusions and exclusions together make up an element's *content rules*.

> **Note: XML:** The specification for XML does not support inclusions and exclusions. If an EDD uses inclusions and exclusions, when you save the associated FrameMaker file to XML the software ignores them. The result could be a document that is valid in FrameMaker, but invalid in XML. When converting an EDD to a DTD, the resulting XML DTD could contain errors. You should only use inclusions and exclusions in an EDD that is part of an SGML structure application.

At least one container in the EDD must also have a rule stating that the container is valid at the highest level in a structured flow. All other elements in the flow are descendants of this container. For example:

Element (Container): Chapter
General rule: Para+, Section+
Valid as the highest-level element.
Inclusion: CrossRef
Automatic insertions
Automatically insert child: Para

At least one container must have a specification about highest-level validity.

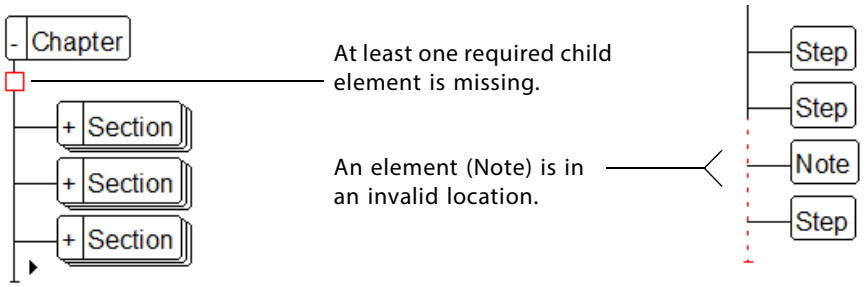
Element (Table): RuleTable
General rule: Title?, Heading, Body, Footing?
Initial structure pattern: Heading, Body

A general rule is required for every container, table, table part, and footnote.

Element (TableBody): Body
General rule: Row+

Element (Footnote): Footnote
General rule: (<TEXT> | Emphasis)+

In a document, if an instance of an element does not conform to the content rules, the Structure View identifies the problem. If an element is missing one or more child elements required in its general rule, a small hole marks the first place where elements are missing. If a child element is in an invalid location according to the general rule and any inclusions or exclusions that apply, the vertical line next to it is broken to the end of the parent. (The hole and dotted line are in red on a color monitor.)



An end user can also validate the document for details on these errors and to find additional errors such as a highest-level element not permitted at that level. For more information on the Structure View and validation, see the FrameMaker user's manual.

Other optional structure rules are available to help you develop a convenient working environment for end users. For containers, you can define descendants to insert automatically along with a new instance of the container in a document. And for tables and most table parts, you can define element tags for child table part elements (in a repeating pattern if necessary).

Writing an EDD general rule

A *general rule* describes the child elements that an element can contain, whether the child elements are required or optional, and the order in which the child elements can occur. It also specifies whether the element can have text. Every element definition for a container, table, table part, footnote, or Rubi group must have a general rule.

When you insert `Container`, `Table`, `Table Heading`, Table `Body`, `Table Footing`, `Table Title`, Table `Cell`, `Table Row`, `Footnote`, `Rubi`, or `Rubi Group` as the element type in a definition, the `GeneralRule` child element is inserted automatically. Type the text of the rule in the `GeneralRule` element. For example:

Element (Container): Section
General rule: Head, Paragraph+, Section*

The tags in the rule must be for elements defined in the current EDD. Containers, footnotes, table titles, and table cells can use any elements in the EDD except for tables and table parts. Tables and other table parts have more specific restrictions on the elements you can use; for information on this, see "Restrictions on general rules for tables" on page 177.

The typographical rules for Rubi groups prohibit line breaks. A Rubi group can use any elements in the EDD, but if you insert a paragraph container element in a Rubi group, FrameMaker will ignore the paragraph break and treat it as a text range. For best results, the general rule for a Rubi group should not include any paragraph container elements.

Syntax of a general rule for EDD elements

A general rule can list child element tags and content symbols, and it can use occurrence indicators, connectors, and parentheses to further describe the contents allowed.

Occurrence indicators and connectors

An occurrence indicator after an element tag specifies whether the child element is required or optional and whether it can be repeated. If you do not use an occurrence indicator, the element is required and can occur once. These are the indicators available:

Symbol	Meaning
Plus sign (+)	Child element is required and can occur more than once.
Question mark (?)	Child element is optional and can occur once.
Asterisk (*)	Child element is optional and can occur more than once.

Multiple element tags in a general rule are separated with connectors that specify the order in which the child elements can occur. These are the connectors available:

Symbol	Meaning
Comma (,)	Child elements must occur in the order given.
Ampersand (&)	Child elements can occur in any order.
Vertical bar (|)	Any one of the child elements in the group can occur.

For example, this general rule specifies that the element must begin with a `Head`, then it must have one or more `Paragraph` elements, and then it can have one or more optional `Section` elements:

```
Head, Paragraph+, Section*
```

This rule specifies that the element can have either one or more `Paragraph` elements or one `List` element:

```
Paragraph+ | List
```

Be careful to write general rules that are not ambiguous. When an end user validates a document, FrameMaker must be able to match child elements to the tags in the general rule without looking ahead to other child elements. For example, this rule is ambiguous because FrameMaker cannot tell whether an `Item` in the document matches the first or second `Item` in the rule without looking ahead for a second `Item`:

```
Item?, Item
```

If you want to specify that an element must have one or two Items, write this rule instead:

```
Item, Item?
```

The connectors in a group of element tags must all be the same type. For example, this rule is erroneous because it uses two different connectors:

```
Caption, Graphic | Table
```

If you need to mix connectors in an element rule, use parentheses to define groups of element tags. In the rule above, if you want a `Caption` followed by either a `Graphic` or a `Table` put parentheses around `Graphic | Table`. For more information, see "Parentheses" on page 177.

Content symbols

A general rule can also use symbols that specify content other than child elements. These are the content symbols available:

Symbol	Meaning
`<TEXT>`	Element can contain text.
`<TEXTONLY>`	Element can contain only text. It cannot contain child elements, even inclusions defined in the content rules of its ancestors.
`<ANY>`	Element can contain any combination of text and elements defined in the EDD.
`<EMPTY>`	Element cannot contain any text or elements.

You can use the `<TEXTONLY>`, `<ANY>`, or `<EMPTY>` symbol only in a rule by itself, but you can combine the `<TEXT>` symbol with element tags for more complex expressions. For example, this rule specifies that the element can begin with text and can end with a table:

```
<TEXT>, Table?
```

Text is always optional and repeatable. An occurrence indicator after a token does not change the meaning of the general rule: `<TEXT>`, `<TEXT>+`, and `<TEXT>*` are all equivalent.

Use the `<TEXTONLY>` token for elements that directly correspond to markup elements with declared content `CDATA` or (for SGML) `RCDATA`.

Use the `<EMPTY>` symbol for elements you want to remain empty. These are some possible examples of empty container elements:

- A paragraph element that has an autonumber but no content, such as a section number on a line by itself
- A text range element that has a `UniqueID` attribute but no content, to describe a source location within a paragraph for cross-references
- A table cell element that remains empty in tables, such as cells in an empty row to provide space between groups of cells in a table

For information on translation to XML and SGML, see "How content rules translate to markup data" on page 181.

Parentheses

You can use parentheses to group element tags and content symbols in a general rule. The items within a pair of parentheses act as a single tag in the rule's syntax. You can use occurrence indicators and connectors with a group as you do with an individual element tag.

For example, this rule specifies that the element must begin with a `Head`, then it must have at least one `Paragraph` or one `List` element, and then it can have one or more optional `Section` elements:

Head, (Paragraph | List)+, Section*

Note that because of the plus sign after the parenthesized group, the `Paragraph` and `List` elements can be repeated any number of times.

A group can also be nested within another group. For example, this rule specifies that the element must begin with a `Front` element and then must have either one or more `Part` elements or one or more `Chapter` or `ErrorSection` elements followed by one or more optional `Appendix` elements:

Front, (Part+ | ((Chapter | ErrorSection)+, Appendix*))

The connectors within a single parenthesized group must be the same, although a group nested within another group can use a different connector.

Make sure that a parenthesized group does not introduce any ambiguities to the general rule. For example, this rule is ambiguous because either alternative can begin with a `Preface` element:

Preface | (Copyright?, Preface, Foreward)

Outer parentheses around a general rule are optional. If you save an EDD as a DTD, FrameMaker inserts outer parentheses when markup requires it.

Restrictions on general rules for tables

The structural parts of a table each use a general rule to describe how an instance of the table can be built. For example, this general rule specifies that a `Table` element begins with a `Title` and then has a `Body` and either a `Heading` or a `Footing` (but not both):

Element (Table): Table
General rule: Title, ((Heading, Body) | (Body, Footing))

The general rule for a table or table part uses the same syntax as the general rule for other elements, but a few additional restrictions apply:

Element type	Restrictions
Table	Limited to one each of the following types of child elements, in this order: title, heading, body, footing. (The rule can specify these elements more than once in Or expressions, but a table element can have only one of each child.)

Element type	Restrictions
	The plus sign (+), asterisk (*), and ampersand (&) are not allowed.
	The tokens <TEXT>, <TEXTONLY>, <ANY>, and <EMPTY> are not allowed.
Heading, body, or footing	Limited to one or more row child elements.
	The tokens <TEXT>, <TEXTONLY>, <ANY>, and <EMPTY> are not allowed.
Row	Limited to one or more table cell child elements.
	The tokens <TEXT>, <TEXTONLY>, <ANY>, and <EMPTY> are not allowed.
Title or cell	All child elements are allowed, except for tables and table parts.
	The tokens <TEXT>, <TEXTONLY>, <ANY>, and <EMPTY> are allowed.

Note that the table format associated with a table element determines whether a new table has a title. When using an object format rule to assign a format to a table, you need to make sure that the format is consistent with the table's general rule. For more information on these formats, see "Setting a table format" on page 242.

In a document, an end user cannot insert table or table part elements outside their restricted context, even if he or she has all the elements available in the Element Catalog. For example, the user cannot insert a table title after an existing title or a table heading in a container. If the user tries to do this, FrameMaker displays an alert.

Default general rules for EDD elements

If you do not specify a general rule, FrameMaker gives the element a default general rule that depends on the element's type. When you import the EDD into a template, FrameMaker inserts a default general rule wherever the EDD has an empty `GeneralRule` element. You can then save the EDD with these corrections.

These are the default general rules:

Element type	Default general rule
Container	<ANY>
Table	TITLE?, HEADING?, BODY, FOOTING?
Table heading, body, or footing	ROW+
Table row	CELL+
Footnote, table title, or table cell	<TEXT>
Rubi Group	<TEXT>, RUBI
Rubi	<TEXT>

If you try to save an EDD as a DTD when there are empty general rules, the resulting DTD will have syntax errors. You need to import the EDD into a template first so that FrameMaker can insert default general rules for you.

Keep in mind that for an EDD to be valid you need to insert a `GeneralRule` element in the definition for every container, table, table part, footnote, or Rubi group—even when you are not filling in the general rule.

Specifying validity at the highest level in a flow

Every structured flow in a document needs one highest-level element. This element is a container and holds all other elements in the flow. You need to define at least one highest-level element for each type of structured flow that can appear in documents.

To specify validity at the highest level in a flow, insert a `ValidHighestLevel` element right before or after the element's general rule. For example:

Element (Container): Chapter
General rule: Paragraph+, Section, Section+
Valid as highest-level element.

When defining a highest-level element, you may want to give it a name that identifies the type of document or flow. For example, in a chapter document the element might be called `Chapter`.

Provide a highest-level element for book files as well as for flows in document files. For example:

Element (Container): Book
General rule: Front, Chapter+, Index?
Valid as highest-level element.

Adding inclusions and exclusions

Note: XML: The specification for XML does not support inclusions and exclusions. If an EDD uses inclusions and exclusions, when you save the associated FrameMaker file to XML the software ignores them. The result could be a document that is valid in FrameMaker, but invalid in XML. When converting an EDD to a DTD, the resulting XML DTD could contain errors. You should only use inclusions and exclusions in an EDD that is part of an SGML structure application.

An *inclusion* is an element that can occur anywhere inside the defined element or its descendants. Inclusions are often used for elements that might be necessary throughout a hierarchy, such as cross-references or terms with special formatting. An *exclusion* is an element that cannot occur anywhere in the defined element or its descendants.

You can define inclusions and exclusions as part of the content rules for containers, tables, table parts, footnotes, and Rubi groups. Defining inclusions and exclusions in a few high-level elements saves you the effort of allowing or prohibiting child elements for individual lower-level elements.

Because inclusions and exclusions apply to an element and its descendants, at some point in the hierarchy an element may be both included and excluded. When this happens, FrameMaker does not allow the element to occur. For example, an `Index` element might be specified as an inclusion in a `Report` element. If a `Head` element excludes `Index` elements, the `Index` is excluded even though `Head` is a descendant of `Report`.

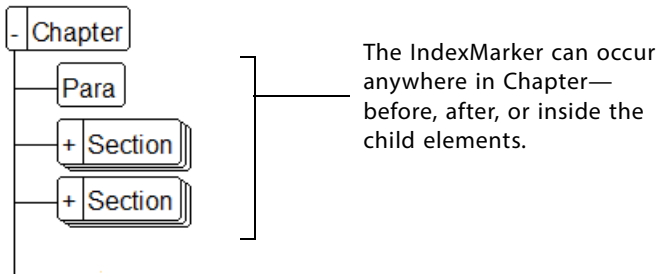
Inclusions

To add an inclusion to an element definition, insert an `Inclusion` element anywhere after the general rule (or optional validity specification) and before the format rules. Then type the tag of the element you want to include.

For example, instead of adding `IndexMarker` to the general rules for `Para` and `Section` (and all the elements they contain), you might specify `IndexMarker` as an inclusion for `Chapter`:

Element (Container): Chapter
General rule: Para+, Section*
Valid as highest-level element.
Inclusions: IndexMarker

In the example, an `IndexMarker` can occur before or after a `Para` or `Section` as well as anywhere within a `Para` or `Section` (unless one of the elements specifies `IndexMarker` as an exclusion):



When defining an inclusion, look for descendants that should not use the inclusion. Add an exclusion in the descendants' definitions to prohibit the inclusion in those contexts.

If you want more than one inclusion, for each additional inclusion, insert an `Inclusion` element and type the tag, or put multiple element names in the same element, separated by commas.

An inclusion can use any element tag defined in the current EDD. An end user will be able to insert the included element only if it is allowed in the context, even though it may be a valid inclusion. For example, the user cannot insert a table footnote between table rows even though the footnote may be a valid inclusion in the table because table footnotes are allowed only in titles and cells.

Exclusions

To add an exclusion to an element definition, insert an `Exclusion` element anywhere after the general rule (or optional validity specification) and before the format rules. Then type the tag of the element you want to exclude.

For example, you might use an exclusion to prevent end users from creating nested `Procedure` elements:

Element (Container): Procedure
General rule: Step+
Exclusions: Procedure

The most common uses of exclusions are to prevent nesting and to counter an inclusion for a particular context.

If you want more than one exclusion, for each additional exclusion, insert an `Exclusion` element and type the tag, or put multiple element names in the same element, separated by commas.

How content rules translate to markup data

In FrameMaker, the general rule and the inclusions and exclusions use a syntax that is based on SGML model groups and declared content. (The occurrence indicators, connectors, and parentheses are the same in both environments.) On import or export between an EDD and a DTD, the content information about child elements is preserved. Note that you do not need to put parentheses around the entire general rule in FrameMaker.

When you convert an EDD to a DTD, FrameMaker also translates content symbols in general rules:

- The FrameMaker content symbol `<TEXT>` translates to a content token of `#PCDATA`. (`#PCDATA` can be combined with element tags, as `<TEXT>` can be in FrameMaker.)
- The FrameMaker general rule `<TEXTONLY>` translates to declared content of `RCDATA`.
- The FrameMaker general rule `<ANY>` translates to the reserved name `ANY` in a markup content model.
- The FrameMaker general rule `<EMPTY>` translates to a declared content of `EMPTY`.

When you convert a DTD to an EDD, FrameMaker performs the translations in reverse. In addition, a declared content of `CDATA` translates to `<TEXTONLY>` in FrameMaker.

For more detailed information on how content rules translate to XML or SGML, see Chapter 21, "Translating Elements and Their Attributes." For information on how structured tables translate to XML or SGML, see Chapter 23, "Translating Tables."

Inserting descendants automatically in containers

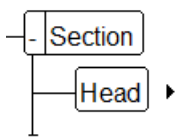
In the definition of a container, you can specify nested descendants to insert automatically. Whenever an end user inserts the container in a document, the descendants are inserted automatically along with it. This makes it convenient for users to work with containers that always begin with the same structure.

To insert descendants automatically, insert an `AutoInsertions` element anywhere after the general rule (or optional validity specification) and before the format rules. For the first descendant, insert an `InsertChild` element and type the element tag. Then for each additional descendant, insert an `InsertNestedChild` element and type the tag. You can have any number of `InsertChild` elements as well as any number of `InsertNestedChild` elements.

For example, this autoinsertion rule specifies that a new `Section` begins with a nested `Head`:

Element (Container): Section
General rule: Head, Para+
Automatic insertions
Automatically insert child: Head

If the last descendant in the autoinsertion sequence is a container that allows text, the insertion point is placed automatically inside the container, ready for the end user to add text:



FrameMaker can also insert sibling elements automatically. In the `Section` and `Head` example above, since a `Section` usually begins with a `Head` and then a `Para`, you can have both the `Head` and `Para` inserted automatically.

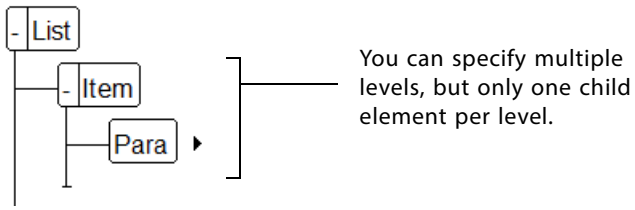
For example, this autoinsertion rule specifies that a new `Section` begin with a nested `Head` and a sibling `Para` element:

Element (Container): Section
General rule: Head, Para+
Automatic insertions
Automatically insert child: Head
Automatically insert child: Para

You can also have FrameMaker insert a sequence of descendants, in which each nested descendant has one child element inserted along with it. For example, this autoinsertion rule specifies that a new `List` has a nested `Item`, and the `Item` has a nested `Para`:

Element (Container): List
General rule: Item+
Automatic insertions
Automatically insert child: Item
and nested child: Para

This is the structure of the new `List`:



Keep in mind that the autoinsertion rules from one element definition do not carry over to another. In the `List` and `Item` example above, even though the `Item` definition in the same EDD may specify `Para` as a nested descendant, to insert `Para` automatically with `List` you still need to explicitly specify `Para` (and `Item`) in the `List` definition.

The descendants in autoinsertion rules do not need to be containers. If you use a table, graphic, cross-reference, variable, marker, footnote, or equation as a nested descendant, that element should be the last one in the sequence of descendants. The automatic insertion stops after the non-container element is inserted.

When FrameMaker inserts descendants automatically in a document, it opens dialog boxes as necessary.

- If a descendant is a table, graphic, cross-reference, variable, marker, or equation and FrameMaker requires information about the element (such as a table format or a graphic filename), the appropriate dialog box opens as the element is inserted.

The end user can cancel a dialog box, and the autoinsertion stops at that point. (This does not affect any descendants that were already inserted automatically.) Be aware of the dialog boxes that may open during autoinsertion so that you can design a behavior that is reasonable for the user.

Inserting table parts automatically in tables

All tables have certain characteristics in common: A table has at least a body; a table heading, body, or footing has at least one row; and a table row has the same number of cells as there are

columns in the table. Whenever an end user inserts a structured table, or part of one, FrameMaker automatically inserts the necessary child elements to build a basic structure.

You can define an initial structure pattern for a table, heading, body, footing, or row, and FrameMaker will use that structure when an end user inserts the table or table part in a document. If you do not define a structure pattern, FrameMaker gives new instances of the element a default initial structure it derives from the general rule.

When an end user inserts a new table element, he or she uses the Insert Table dialog box to specify the number of heading, body, and footing rows and the number of columns. The initial structure pattern (or the general rule) determines which row elements to use in the heading, body, and footing, and which cell elements to use in the columns.

Initial structure pattern

To define an initial structure pattern for a table, heading, body, footing, or row, add an `InitialStructurePattern` element anywhere after the general rule and before the format rules. Then list the tags of child elements in the initial structure, separated by commas.

For a table element, the initial structure pattern must include at least a body and can have at most a title, a heading, a body, and a footing, in that order. When FrameMaker gives a table its structure, it uses each type of table part in the pattern. For example, this pattern specifies a table with an initial structure of `Heading` and `StandardBody`:

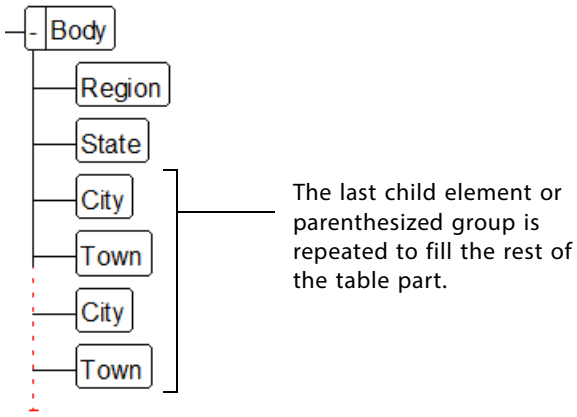
Element (Table): Table
General rule: Title?, Heading, (RulesBody | StandardBody)
Initial structure pattern: Heading, StandardBody

A table format specifies whether or not a new table has a title. Although you can include a title in the initial structure pattern, this information is overridden by the table format. For information on giving a table element a format, see "Setting a table format" on page 242.

For a heading, body, footing, or row, if the table part needs more elements than appear in the structure pattern, FrameMaker repeats the last item in the syntax to fill the number of rows and cells required. (In these cases, the defined structure is truly a pattern.) The last item may be a single child element or a parenthesized group. For example, this pattern specifies a body using the row elements `Region`, `State`, `City`, and `Town`:

Element (TableBody): Table
General rule: Region, State, City, Town, Zip
Initial structure pattern: Region, State, (City, Town)

If an end user inserts a table with this body and specifies six body rows in the Insert Table dialog box, the new table has the following body row structure:



It is also possible that not all of a structure pattern will be used. For example, if an end user inserts a table with the body defined above and specifies two body rows, the body in the new table has only a `Region` row and a `State` row.

If an end user adds a row to an existing table by inserting a row element, the structure of the new row is based on the initial structure pattern of the parent heading, body, or footing. (But if the user adds a row by pressing Control-Return or by using the Add Rows or Columns command, the new row takes the structure of the row with the insertion point.)

The syntax of an initial structure pattern is restricted as follows:

- The only arguments allowed are element tags. The content symbols `<TEXT>`, `<TEXTONLY>`, `<EMPTY>`, and `<ANY>` are not allowed.
- The comma (`,`) is the only connector allowed. No occurrence indicators are allowed.
- Parentheses are allowed to group element tags.

For more information on the symbols allowed, see "Restrictions on general rules for tables" on page 177.

Be sure that the child elements in an initial structure pattern are valid according to the general rule of the table or table part. Otherwise, a new table may have invalid structure (such as in the example above with `City` and `Town`).

Default initial structure

If you do not specify an initial structure pattern in an element definition, FrameMaker uses the general rule to give a new table or table part its initial structure.

For a table element, FrameMaker builds a default initial structure by taking the first of each type of table part in the table's general rule. For example, suppose a table element does not have an

initial structure pattern but has the following general rule, where `Normal` and `Special` are two types of table bodies:

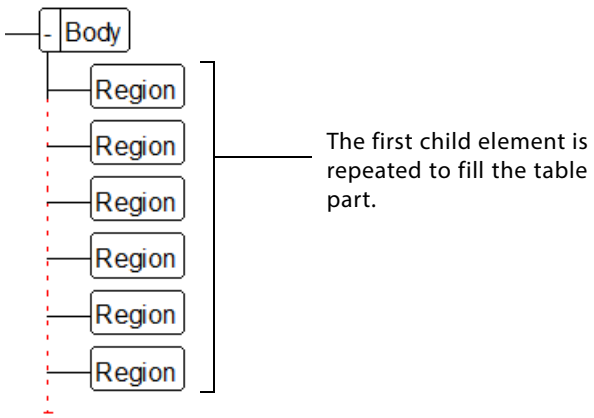
Element (Table): Table
General rule: Title?, Heading, (Normal | Special)

If an end user inserts a table, uses a table format that includes a title, and specifies some heading and body rows, the new table has an initial structure of `Title`, `Heading`, and `Normal`. (If the user also gives the table some footing rows, those rows have the default tag `Footing`.)

For a table heading, body, footing, or row, the default initial structure is simply the first element tag to appear in the general rule. If the table part needs more child elements, the first child element is repeated to fill the number of rows and cells required. For example, suppose a body element does not have an initial structure pattern but has the following general rule, where `Region`, `State`, and `City` are rows:

Element (TableBody): Body
General rule: Region, State, City

If an end user inserts a table with this body and specifies six body rows in the Insert Table dialog box, the new table has the following row structure:



Inserting Rubi elements automatically in Rubi groups

A Rubi group always includes a Rubi element as its last child. Whenever an end user inserts a Rubi group, FrameMaker automatically inserts the necessary child elements to build a basic structure.

You can define an initial structure pattern for a Rubi group and FrameMaker will use that structure when an end user inserts the Rubi group in a document. If you do not define an initial structure pattern, FrameMaker gives new instances of the Rubi group a first Rubi element found in the Rubi group's general rule. If no Rubi element is specified in the general rule, FrameMaker inserts a default Rubi element named `RUBI`.

Initial structure pattern

For a Rubi Group element, the initial structure pattern can specify one child Rubi element. To define an initial structure pattern for a Rubi group, add an `InitialStructurePattern` element anywhere after the general rule and before the format rules. Then specify the tag of the child Rubi element. For example, this pattern specifies a Rubi group with an initial child Rubi element named `MyRubiElement`:

Element (Rubi Group): MyRubiGroup
General rule: Oyamoji, (MyRubiElement | SpecialRubiElement)
Initial structure pattern: MyRubiElement

Debugging structure rules

After writing structure rules, you should try them out by importing the EDD into a sample document and inserting, wrapping, and moving a variety of elements. If any elements that you expect to be valid are displayed as invalid (or vice versa) in the Structure View, check the EDD for these errors:

- Typing errors in element tags, content symbols, or symbols in any structure rule
- A general rule that has ambiguous element tags or mixed connectors (For advice on avoiding these problems, see page 176.)
- Inclusions or exclusions specified for an element that is not a container or for a container with the general rule `<TEXTONLY>`
- An element used at the highest level in its flow that does not have a specification for highest-level validity

If FrameMaker identifies any problems when you import an EDD, it produces a log file of warnings and errors. For information on how to work with this file, see "Log files for imported element definitions" on page 167.

To help you locate ambiguous element tags or mixed connectors, the log file shows a caret symbol (^) in front of a tag or a connector that may be incorrect. For example, this ambiguous general rule is displayed as follows in the log file:

^Item?, ^Item

13 Attribute Definitions

Attributes provide a way to store descriptive information with an instance of an element that is not part of the element's content. In FrameMaker, you can define attributes for many purposes—such as to record the current status of an element's content, to allow cross-referencing between elements with ID attributes, or to specify how an element is to be formatted. You can define attributes for any element in FrameMaker.

Attribute definitions in FrameMaker translate to attribute declarations in markup. If you move documents between FrameMaker and XML or SGML and conform to the markup requirements, your attributes and their values are usually preserved. FrameMaker provides a default translation for all of its attribute types, and you can modify the translation by using read/write rules.

In this chapter

This chapter explains how to define attributes in FrameMaker. It contains these sections:

Some uses for attributes

One of the most common uses for attributes is to record information about an instance of an element that an end user may want to inspect or update while editing the document. This information describes the element's content in some way. For example:

- A security attribute in a memo element can tell the level of classification for the memo's contents (`Security=Unclassified`).
- A status attribute in a section element can describe the current review stage of the section's contents (`Status=Alpha`).

- An author attribute in a chapter element can identify the author of the document (`Author=itp`).
- A size attribute in a graphic element can specify the width of a frame (`ArtWidth=8.5`). For XML you provide a CDATA value, and for SGML you can provide a number value.

Attributes can store source and destination information for elements. These are often used for cross-referencing between elements. For example:

- An identifier attribute in a head element can uniquely identify the element as a source for cross-references (`ID=Intro`).
- A reference attribute in a cross-reference element can store the ID of the source element that is referred to (`Reference=Intro`).

You can also use attributes to determine the appearance of an element in a FrameMaker document. For example:

- A type attribute in a list element can specify whether the list should be numbered or bulleted (`Type=Bulleted`).
- A prefix attribute in a note element can provide a text string to display before the element's content (`Prefix=Important`).

How an end user works with attributes

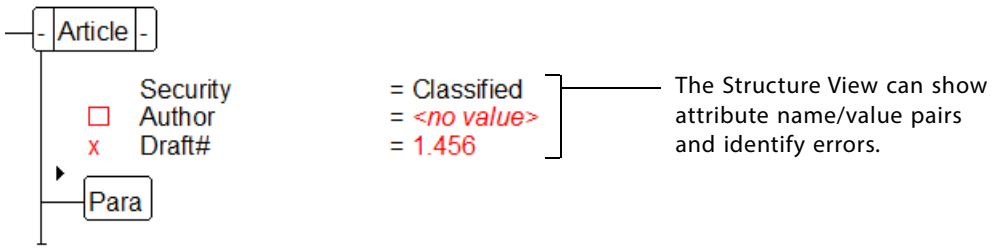
In a document, an end user provides values for particular instances of attributes. The user can also validate the document to be sure that the attribute values meet the criteria of your definitions. To develop an environment that is reasonable for the user, it may help to understand a few things about how the user edits and validates attributes.

An end user supplies values for an attribute in the Attributes dialog box. The dialog box shows information from the attribute definition—name, type, range or choices, and default value—to guide the user to enter values that are appropriate. If the user tries to enter a value that is not valid according to the definition, FrameMaker does not accept the value and displays an alert.

You can also define read-only attributes with values that are set by a structure API client or by a FrameMaker cross-reference. An end user cannot modify read-only attributes.

Even though an end user cannot enter invalid attribute values in the Attributes dialog box, it is still possible to end up with inappropriate values or to have attributes in an element that is not defined to contain them. This usually happens because the user has pasted values or attributes or imported a new Element Catalog with different attribute definitions, or because of changes made to the document by a structure API client.

The end user can see attribute name/value pairs for each element in the Structure View. The view displays all attributes, no attributes, or only attributes with required and specified values, whichever the user requests. Invalid values are identified by an *x* next to the attribute name. If an attribute is missing a required value, the view shows a hole next to the name. (The error information is in red on a color monitor.) For example:



FrameMaker also identifies errors involving attributes when the end user validates the document. For more information on the Structure View and how a user works with attributes, see the FrameMaker user's manual.

Writing attribute definitions for an element

To make attributes available for an element, you need to define the attributes as part of the element's definition. You can define attributes for any element in FrameMaker.

An element definition can have a list of attribute definitions. Within the list, each definition must have a name, a type, and specification of whether an attribute value is required or optional. If the attribute is of type `Choice`, the definition must also include a list of possible values. For example:

Element (Container): Article
General rule: Para+, Section*
Attribute list
1. **Name:** Author **String** **Required**
2. **Name:** Security **Choice** **Optional**
   Choices: Top Secret, Classified, Unclassified

Each attribute must have a name, a type, and a required or optional specification.

A Choice attribute also needs a list of values.

Element (Graphic): ImportedArt
Attribute list
1. **Name:** Width **Real** **Optional**
   Range: From 6 **to** 11
   Default: 8.5

You can optionally define a range of possible values (for the numeric attribute types) and a default value (if a value is optional). You can also make any attribute read-only.

To write attribute definitions for an element, insert an `AttributeList` element. (If the element is a container, table, table part, or footnote, the attribute list goes after the structure rules.) When you insert the `AttributeList` element, the first `Attribute` child element is inserted automatically. Define the first attribute, and then insert and define additional `Attribute` elements as necessary. The definitions are numbered automatically.

Attribute name

When you insert an `Attribute` element, the `Name` child element is inserted automatically along with it. Type the attribute name in the `Name` element. For example:

1. **Name:** Author

Give an attribute a name that is self-explanatory. Although different elements can have attributes with the same name, it's good practice to use the same name only for the same semantics. Attribute names are case-sensitive, and they cannot contain white-space characters or any of these special characters:

```
( ) & | , * + ? < > % [ ] = ! ; : { } "
```

An attribute name can have up to 255 characters in FrameMaker, but you should try to keep the names concise. End users often display attribute names with their elements in the Structure View.

> **Note: SGML:** If you plan to export documents to SGML, you may want to define attribute names that conform to the naming rules and the maximum name length permitted by the concrete syntax you'll be using in SGML. If you prefer names that do not adhere to the SGML conventions, you can provide read/write rules to convert them to SGML equivalents when you export. For more information, see "Naming elements and attributes" on page 296.

Attribute type

The attribute type determines what kind of values are allowed in the attribute. You can specify attribute types for string values, numeric values, and IDs and their references. Insert one of the type elements after the name in the attribute definition. For example:

1. **Name:** Author **String**

These are the attribute types available:

Type	Values allowed in the attribute
`String`	An arbitrary text string
`Strings`	One or more arbitrary text strings
`Integer`	A whole number, which may be signed (optionally restricted to a range of values); valid integers: `224`, `+795`, `-1024`
`Integers`	One or more integers (optionally restricted to a range of values)
`Real`	A real number (optionally restricted to a range of values), can use `e` notation to express 10 to the power of *x*; valid real numbers: `23`, `.23`, `0.23`, `2.3e-1`, `.023e+01`
`Reals`	One or more real numbers (optionally restricted to a range of values)
`Choice`	A value from a list of predefined choices
`UniqueID`	A string that uniquely identifies the element

Type	Values allowed in the attribute
IDReference	A string used elsewhere as the value of a UniqueID attribute
IDReferences	One or more references to UniqueID attributes

Note the following about these attribute types:

- With the Choice attribute type, you also need to provide a list of possible values in the attribute definition. With the numeric types, you have the option of constraining the values to a range. For details, see "List of values for Choice attributes" on page 193 or "Range of values for numeric attributes" on page 194.
- The numeric types allow only integer or real values and no additional strings. This allows for precise validation in FrameMaker, because the software can check that the values fall within a numeric range. If you want end users to have the flexibility to enter values such as 3 in or 60 mm and do not need to limit them to a range, use the String attribute type instead of a numeric type.
- The UniqueID, IDReference, and IDReferences types are typically used for element-based cross-referencing. For information on working with these types, see "Using UniqueID and IDReference attributes" on page 195.
- The multiple-token types Strings, Integers, and Reals are for attributes that need to store a set of information. For example, you can use a Reals attribute to store dimensions for an imported graphic object—an end user might enter two tokens that together define the height and width of the object.

In a document, if an end user tries to enter a value that is not allowed by the attribute type, FrameMaker does not accept the value and displays an alert describing the problem. If the user enters an invalid value by other means (such as pasting), FrameMaker identifies the attribute as invalid. (A duplicate UniqueID value is discarded.)

> **Note: SGML:** All attributes in FrameMaker translate to attributes in SGML. FrameMaker and SGML have different attribute types available however, and the types generally do not translate one to one. Multiple SGML attribute types can translate to the same FrameMaker type, and vice versa. For more information, see Chapter 21, "Translating Elements and Their Attributes."

Specification for a required or optional value

You need to specify whether or not an attribute requires a value for each instance of the element in a document. Insert a Required or Optional element after the type element in the attribute definition. For example:

1. **Name:** Author **String** **Required**

In a document, if an attribute requires a value but does not have one, FrameMaker identifies the attribute as invalid.

If a value is optional, you can assign a default value to the attribute. A default value can be used to control the element's formatting or to work with a structure API client. For more information, see "Default value" on page 195.

Hidden and Read-only attributes

You may sometimes want to restrict end users from editing the value of an attribute, and let FrameMaker or a structure API client provide the value instead. This is especially helpful for an attribute you want to use for precise tracking of particular element instances or changes in a document.

For example, FrameMaker can set an attribute value for elements used in cross-referencing. You specify read-only for a `UniqueID` attribute that stores an ID as source information for its element. Then when an end user inserts an element-based cross-reference to an instance of the element, FrameMaker sets the attribute's ID for that instance. For information on using an attribute in this manner, see "Using UniqueID and IDReference attributes" on page 195.

Your structure application might store information in attributes that is of no interest to the end user. In that case, you can make the attribute hidden. Not only is the end user kept from editing the attribute value, he or she will never see the attribute or its value.

These are possible uses for read-only or hidden attributes with values set by an API client:

- The attribute stores key information in a structured document that is a representation of a database. For example, a `UniqueID` attribute in FrameMaker might store an object ID with data extracted from a database. You may want to preserve the ID so that you can export the data back to its source location in the database.
- The attribute stores information about the version of a document. For example, some version-control schemes have an `IDReference` attribute pointing to each element in a document, and the attribute is updated for each revision.

To make an attribute read-only, insert a `ReadOnly` element after the `Required` or `Optional` element in the attribute definition. For example:

1. Name: Source **UniqueID** **Optional** **Read-only**

To make an attribute hidden, insert a `Hidden` element after the `Required` or `Optional` element in the attribute definition. For example:

1. Name: Source **UniqueID** **Optional** **Hidden**

> **Note:** If you make an attribute read-only or hidden, we suggest that you specify the value to be optional. If a value is required and an instance of the attribute does not have a value, the document will show a validation error that the end user cannot correct.

Although a read-only or hidden attribute is not editable in the FrameMaker user interface, you can still edit its value with a structure API client. For information on API clients, see the *Structure Import/Export API Programmer's Guide*.

List of values for Choice attributes

For an attribute of type `Choice`, you need to provide a list of possible values for the end user. Insert a `Choices` element after the `Required` or `Optional` element (or after the `ReadOnly` element if the definition has one). Then type the possible values, separating them with a comma and a space. For example:

1. **Name:** Security **Choice** **Required**
   Choices: Top Security, Classified, Unclassified

The tokens can be strings of up to 255 characters. They can have white-space characters but cannot have any of these special characters:

```
( ) & | , * + ? < > % [ ] = ! ; : { } "
```

If you're using attribute values to format elements, the order of the values in the list may matter. When specifying attribute values in format rules, you can use greater-than or less-than operators to test the location of the current value in the `Choices` list. For more information, see "Attribute indicators" on page 255.

> **Note: SGML:** If you plan to export documents to SGML, you may want to define values that conform to the naming rules and the maximum name length permitted by the concrete syntax you'll be using in SGML. If you prefer values that do not adhere to the SGML conventions, you can provide read/write rules to convert them to SGML equivalents when you export. For more information, see "Renaming attribute values" on page 301.

In a document, the values appear in the Attribute Value drop-down list in the Attributes dialog box when this attribute is selected. An end user chooses from the list to enter one of the values in the attribute. If the user enters an invalid value through other means (such as pasting), FrameMaker identifies the attribute as invalid.

Range of values for numeric attributes

If an attribute's type is `Integer`, `Integers`, `Real`, or `Reals`, you can optionally provide a range of possible values. Insert a `Range` element after the `Required` or `Optional` element (or after the `ReadOnly` element if the definition has one). A `From` child element is inserted automatically. Type the beginning value, insert a `To` element, and then type the ending value. For example:

1. **Name:** ArtWidth **Real** **Optional**
   Range: From 6 **to** 11

The values you enter must be appropriate for the type of attribute—that is, either all integers or all real numbers. The range is inclusive.

In a document, the Attributes dialog box shows the range of values when this attribute is selected. If the end user tries to enter a value outside the range, FrameMaker does not accept the value and

displays an alert describing the problem. If the user enters an invalid value by other means (such as pasting), FrameMaker identifies the attribute value as invalid.

Default value

If a value is optional in an attribute, you can specify a default value. FrameMaker can use the default value for formatting by attributes if an instance of the attribute does not have a value. Insert a `Default` element as the last child element in the attribute definition, and then type the value. For example:

1. **Name:** ArtWidth **Real** **Optional**
   Range: From 6 **to** 11
   Default: 8.5

If the attribute is one of the multiple-token types (`Strings`, `Integers`, `Reals`, or `IDReferences`), you can specify more than one value token for the default value. For each additional token, insert a `Default` element and type the value. When you add a second token, the `Default` label in the EDD changes to plural. For example:

1. **Name:** ArtWidth **Real** **Optional**
   Range: From 6 **to** 11
   Default: 8.5
   11

For information on how FrameMaker can use attribute values in formatting, see "Using attributes to format elements" on page 199. For information on structure API clients, see the *Structure Import/Export API Programmer's Guide*.

Using UniqueID and IDReference attributes

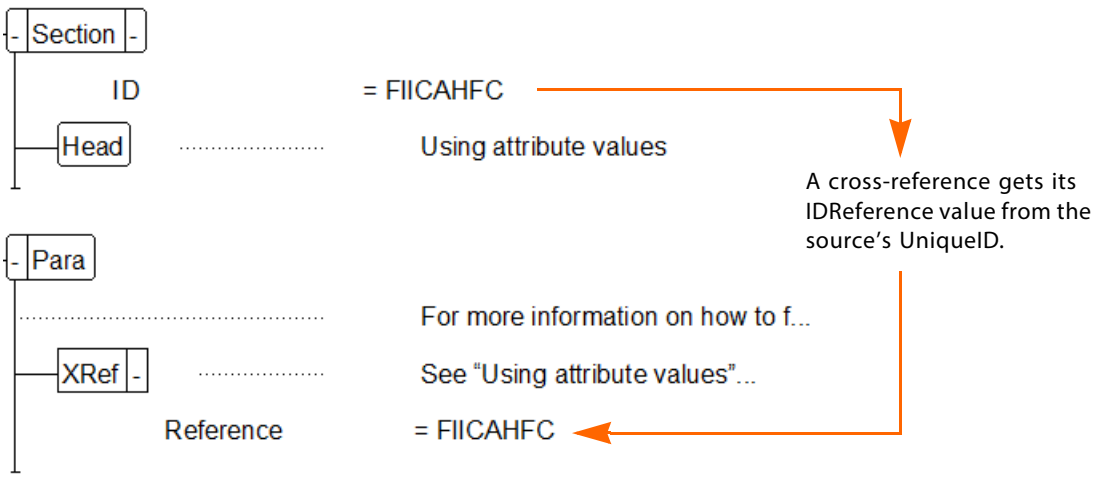
A `UniqueID` attribute stores a string that uniquely identifies an instance of an element in a document or book. In FrameMaker, this ID value is often used as a source address for cross-references to the element. A structure API client or a structure application might also use the ID for other purposes.

When a `UniqueID` attribute is used for element-based cross-referencing, the elements that reference the ID have an `IDReference` attribute. The `IDReference` attributes store the address from the source's `UniqueID` attribute.

FrameMaker provides a mechanism for automated cross-referencing using the `UniqueID` and `IDReference` attributes. You need to assign a `UniqueID` attribute to elements that will likely be the sources of references (such as chapters, sections, and tables) and an `IDReference` attribute to the cross-reference object elements that refer to the sources. When an end user inserts a cross-reference to a source, FrameMaker automatically fills in the cross-reference's `IDReference` attribute with the value from the source's `UniqueID` attribute.

With FrameMaker cross-referencing, information from the source appears in the cross-reference in the document. For example, a cross-reference might show the heading and the beginning page number of a section it refers to. To determine what information from a source appears in a cross-reference, you define a *cross-reference format*. For more on these formats, see "Setting a cross-reference format" on page 246.

This example shows how a FrameMaker element-based cross-reference appears in the Structure View:



If an end user inserts a cross-reference to a `UniqueID` that does not yet have a value, FrameMaker generates an ID value for the `UniqueID` and the `IDReference`.

Attributes for cross-referencing can help end users keep track of their references and what they point to. Because the `UniqueID` and `IDReference` attributes on both ends of a cross-reference store the same value, a user can look at a cross-reference element in the Structure View and see information about the reference's source. In addition, if the user knows the `UniqueID` value of a source, he or she can find all cross-references to the source by searching for other elements with that value in their `IDReference` attribute.

You may also want to make it possible for end users to work with `IDReference` attributes as informal pointers to elements with a `UniqueID`. In this case, the pointers are not FrameMaker cross-references, so information from the source (such as heading and page number) does not appear in the document—but the user can still look at the Structure View to see pointers to related information. For example, suppose you define an `IDReference` for a `Para` container element. To keep track of information from another section in an instance of the `Para`, a user can

manually enter the section's ID in the IDReference for Para (using the Attributes dialog box). He or she can go to the source from the Para later by searching for the element with the ID.

Note: SGML: When importing and exporting between FrameMaker and SGML, the UniqueID and IDReference attributes are preserved. UniqueID attributes in FrameMaker translate to ID attributes in SGML, and IDReference attributes translate to IDREF attributes. For more information, see Chapter 25, "Translating Cross-References."

UniqueID attributes

You can assign a UniqueID attribute to any element in FrameMaker. If you plan to use the element as a source for cross-references, the element will likely be a chapter, section, table, or figure. An element can have only one UniqueID attribute.

In the attribute definition, insert a UniqueID element after the name. For example:

Element (Container): Section
General rule: <TEXT>
Attribute list
1. Name: ID **UniqueID** **Optional** **Read-only**

The value in an instance of a UniqueID attribute must be unique for this attribute type in a document or book. Even if a document has two different elements (such as Section and Chapter) that have a UniqueID attribute, you cannot have any duplication of ID values.

It is possible for a document to end up with IDs that are not unique—for example, if the end user shows hidden text that contains an element with a conflicting ID, or if you change an attribute type to UniqueID and instances of the attribute already have duplicate values. FrameMaker identifies duplicate IDs as invalid.

An end user can provide ID values, or FrameMaker can generate the values. Each method has its advantages:

- If a user provides the IDs, he or she can use values that are meaningful (whereas an ID that FrameMaker generates is a random string). This can make it much easier to remember IDs and to recognize them in the Cross-Reference dialog box and the Structure View.
- If FrameMaker generates the IDs, the IDs are virtually guaranteed to be unique within a document or book, and they will remain unique because a user cannot edit them. The IDs that FrameMaker generates also conform to the SGML reference concrete syntax.

Although UniqueID attributes are often used in sources for cross-references, an element with this attribute is not required to have a reference to it.

When an end user provides an ID

An end user can provide an ID value of up to 255 characters in the Attributes dialog box or in some cases by pasting (unless the attribute is read-only). FrameMaker tries to ensure the uniqueness of IDs as the user edits a document. For example, if the user enters an ID that is not

unique, FrameMaker does not accept the value and displays an alert. If the user pastes an element with an ID that is not unique, the pasted element loses its attribute value.

FrameMaker cannot test for whether an entered ID is used in a different document in a book, especially since one document can be in more than one book. When the user validates a book, however, FrameMaker reports conflicts between IDs across documents in the book.

An end user can edit an existing ID in the Attributes dialog box (unless the attribute is read-only). Any cross-references already pointing to that element may become unresolved. FrameMaker accepts the value if it is unique, but warns the user about possible unresolved references. The user can check for unresolved cross-references by searching for an element with an `IDReference` attribute value equal to that of the replaced ID.

> **Note: SGML:** If you plan to export documents to SGML, the SGML naming rules will likely allow fewer characters—and different characters—for attribute values. Refer to the concrete syntax you'll be using in SGML for the maximum name length and the characters allowed. To ensure that your end users conform to the concrete syntax, you may want to prepare recommendations on entering values. Remind users to begin IDs with a name-start character and to use only name characters thereafter.

When FrameMaker generates an ID

If an end user inserts a cross-reference element and the source's `UniqueID` attribute does not yet have a value, FrameMaker provides a unique value for it. The value is entered in both the `UniqueID` attribute and the `IDReference` attribute pointing to it.

An ID that FrameMaker generates is an eight-character string that begins with a capital letter and then has capital letters and digits. If you are exporting documents to markup, these IDs conform to the SGML reference concrete syntax.

A generated ID is unique in its document. The ID has the form *pppxxxxx*, where *ppp* is derived from the name of the document and *xxxxx* is a random string. Because two documents in a book must have different filenames, generated IDs in the documents will not conflict. If an end user renames a document, the *ppp* is recalculated for values that FrameMaker generates thereafter, but any existing values are not replaced.

FrameMaker automatically generates an ID value if the user inserts a cross-reference to the element, but does not insert an ID value. If you only want FrameMaker to have the ability to enter ID values, make the attribute read-only, thereby preventing the user from setting the attribute. For more information, see "Hidden and Read-only attributes" on page 193.

IDReference attributes

You can assign an `IDReference` or `IDReferences` attribute to any element in FrameMaker. If you plan to use the element as a FrameMaker cross-reference, the element should be a cross-reference object element.

In the attribute definition, insert an `IDReference` or `IDReferences` element after the `Name` element. For example:

Element (CrossReference): XRef
Attribute list
1. Name: Reference **IDReference** **Required**

In a document, when an end user inserts a cross-reference element with an `IDReference` and points the reference to an element with a `UniqueID`, FrameMaker automatically fills in the `IDReference` with the value from the source's `UniqueID`.

FrameMaker also allows an end user to point a cross-reference element to a paragraph source rather than to an element source. In this case, the value of the `IDReference` remains unspecified.

If you want to require end users to base all cross-references on elements rather than on paragraphs, provide an `IDReference` attribute for every cross-reference element, and make the attribute required. If a cross-reference points to a paragraph, its `IDReference` attribute does not have a value, and FrameMaker identifies the attribute as invalid.

> **Note: SGML:** For element-based cross-referencing in FrameMaker, you use the singular `IDReference` attribute, and one value is stored to describe the location of the source. Some SGML applications may require the `IDReferences` list attribute to store multiple values that build composite source information. Use `IDReferences` if you plan to share documents with one of these applications.
>
> If you export an `IDReferences` list attribute to SGML, multiple-value source data for the attribute is preserved. If you import a multiple-value source from SGML, FrameMaker preserves the source data but uses only the first value.

You can define more than one `IDReference` attribute for an element. For example, you may want to do this to allow end users to work with the attributes as informal pointers to several sources. For a discussion of pointers that are not FrameMaker cross-references, see page 196.

It is possible for a document to end up with an `IDReference` value that does not match a `UniqueID` value in the document or book (usually because the end user has pasted the `IDReference` value or deleted the `UniqueID` value). FrameMaker identifies the `IDReference` as invalid.

Using attributes to format elements

You can refer to an attribute name/value pair in a format rule to identify instances of an element in a document. If an instance of the element has the attribute name and value specified, FrameMaker applies the format rule to it.

For example, suppose you want to be able to format the items in a `List` element two different ways—with bullets or with numbers. By giving the element an attribute that describes the type of `List`, you can allow an end user to specify the bulleted or numbered type for each instance

of the element and have the instance formatted according to that information. (If you did not use an attribute, you would need to define two separate `List` elements.)

In the `List` element definition give the element a `Type` attribute, and in the `Item` definition refer to the possible values for the attribute in the text format rules. The `Item` definition in this example specifies that each `Item` begins with a bullet if it appears in a `List` that has a `Type` attribute with the value `Bulleted`, or the `Item` begins with an incrementing number if it appears in a `List` that has a `Type` attribute with the value `Numbered`:

Element (Container): List
General rule: Item+
Attribute list
1. **Name:** Type **Choice** **Required**
   Choices: Bulleted, Numbered

Element (Container): Item
General rule: <TEXT>
Text format rules
1. **If context is:** List [Type = "Bulleted"]
   Numbering properties
   Autonumber format: \b\t
   Character format: bulletsymbol
   Else, if context is: List [Type = "Numbered"]
   1.1 **If context is:** {first}
   Numbering properties
   Autonumber format: <n=1>\t
   Else
   Numbering properties
   Autonumber format: <n+>\t

The value of the attribute (from the List element) determines a context for text formatting.

You can refer to attribute values in object format rules as well as in text format rules. In this example, a `Table` uses `Format A` if its `Type` attribute has the value `Summary` or it uses `Format B` if its `Type` attribute has the value `Examples`:

Element (Table): Table
General rule: Title, Heading, Body
Attribute list
1. **Name:** Type **Choice** **Optional**
   Choices: Summary, Examples
   Default: Summary
Initial table format
1. **If context is:** [Type = "Summary"]
   Table format: Format A
   Else, if context is: [Type = "Examples"]
   Table format: Format B

The value of the attribute (from the current element) determines a context for object formatting.

FrameMaker can use the default value of an attribute to determine context. In the `Table` example above, if `Type` is not specified for a table in a document, the table's initial format will be `Format A` (the format for the default value `Summary`).

Note that in both the `Item` and the `Table` examples, you could use `Else` rather than `ElseIf` to describe the second context. In the `Item` definition `Numbered` is the only other possible value for the `Type` attribute from `List`, and in the `Table` definition `Examples` is the only other possible value. When you use the `Else` specification, the label in the definition is simply `Else` and you do not specify a context such as `Type="Numbered"`.

You can test the values of multiple attributes by joining the specifications with an ampersand (`&`). For example, this specification is true if the element has a `Type` attribute with the value `Numbered` *and* a `Content` attribute with the value `Procedure`:

```
List [Type = "Numbered" & Content = "Procedure"]
```

In addition to testing for equality with attribute values, you can also use `!=` to test for inequality (with all attribute types) or a greater-than sign (>) or less-than sign (<) to test for comparison (with the `Choice` and numeric types).

If you use a greater-than sign or a less-than sign with a `Choice` attribute in a format rule, FrameMaker evaluates the name/value pair using the order in the list of values in the attribute's definition, with the "lowest value" being the one on the left. For example, this pair specifies any `Security` value that is to the left of `Classified` in the defined `Choices` list for the `Security` attribute:

```
Report [Security < "Classified"]
```

For more detailed information on attribute name/value pairs in context specifications, and the operators you can use with them, see "Attribute indicators" on page 255.

Using attributes to provide a prefix or suffix

A *prefix* is a text range that appears at the beginning of an element (before the element's content); a *suffix* is a text range that appears at the end of an element (after the content). An attribute value can provide the text for the prefix or suffix of a container. In this example, the `Note` definition specifies that the prefix is the current value of the element's `Label` attribute:

Element (Container): Note
General rule: <TEXT>
Attribute list
1. **Name:** Label **Choice** **Required**
   Choices: Important, Note, Tip

Prefix rule
1. **In all contexts.**
   Prefix: <attribute[Label]> ——— The value of the attribute provides a text string for the prefix.
   Font properties
   Weight: Bold

A reference to an attribute value can also include element tags. In this case, FrameMaker uses the value for the named attribute in the closest containing element with the specified tag. For example, this definition displays the value of the `Label` attribute in the closest containing `List` or `Section` that has the attribute:

Prefix: <attribute[Label: List, Section]>

If you list element tags and want to search in the current element, you need to include that element in the list of tags.

For more detailed information on attribute names in prefix and suffix definitions, see "Attributes in a prefix or suffix rule" on page 232.

14 Text Format Rules for Containers, Tables, and Footnotes

You can define text formatting properties for containers, tables, table parts, and footnotes in FrameMaker. When an end user inserts one of these elements in a document, text in the element or in a descendant is formatted automatically according to the format rules in the EDD.

Text formatting in FrameMaker is hierarchical—an element can inherit its properties from ancestors and pass on its properties to descendants.

XML uses CSS and XSL to express formatting of a document apart from the markup data. When reading XML, FrameMaker does not use the CSS data or any of the formatting expressed in XSL. Instead, FrameMaker uses the template associated with the XML structure application to format the data. When writing XML, FrameMaker can generate a CSS file based on the formatting in the document template. You can also use the **StructureTools > Generate CSS2...** command to generate a CSS.

The formatting information in the template includes format definitions as well as format rules specified in the EDD. You specify whether or not to generate the CSS in the `structapps.fm` file. For more information, see and Chapter 20, "Saving EDD Formatting Information as a CSS Stylesheet."

Markup does not provide a mechanism for formatting text, so the text format rules you write in FrameMaker do not have counterparts in markup. If you import a DTD, you can add format rules to the resulting EDD for use in FrameMaker. If you import a markup document, the text in the document is formatted for elements that have format rules in the EDD you use. If you export a document or EDD to markup, the text formatting information is not preserved.

In this chapter

This chapter explains how to write text format rules for containers, tables and their parts, and footnotes in FrameMaker. It contains these sections.

- Other format rules for special purposes:
  - "Writing first and last format rules" on page 226
  - "Defining prefixes and suffixes" on page 228
  - "When to use an autonumber, prefix or suffix, or first or last rule" on page 233
- Format change lists you can refer to and limits on values in change lists:
  - "Defining a format change list" on page 234
  - "Setting minimum and maximum limits on properties" on page 235
- Information to help you correct errors in format rules:
  - "Debugging text format rules" on page 237

Overview of text format rules

The text format rules in an element definition can have any combination of the following:

- A reference to a "base" paragraph format stored in the document. The paragraph format defines all aspects of text and paragraph formatting—including font properties, indentation, line spacing, alignment, autonumbering, and hyphenation properties. If a definition does not have a reference to a format, the defined element inherits its format from an ancestor.
- One or more format rules that describe changes to the paragraph format in use. A change can apply to the element in all contexts where it appears or only in a particular context, as specified in each rule. Several rules can apply in one context.

  Format rules can list changes to specific formatting properties, or they can refer to a different paragraph format, to a character format (if you're formatting the element as a text range), or to a list of changes stored elsewhere in the EDD.

When FrameMaker formats text in an element, it applies a paragraph format to the element along with any appropriate changes described in format rules. For example, a `Head` element might have a different point size at different section levels, but in other respects it would be formatted the same everywhere. If you anticipated having two levels for the element, you could define its formatting in this way:

Element (Container): Head
General rule: <TEXT>
Text format rules
Element paragraph format: head
1. **Count ancestors named:** Section
If level is: 1
No additional formatting.
Else, if level is: 2
Default font properties
Size: 14pt

The base paragraph format for all Head elements

If a Head appears in a second-level Section, the format rule changes the point size to 14.

By using context-dependent format rules, you don't need to define and maintain a separate paragraph format for each place in which an element can occur.

Any part of an element's formatting information can be inherited from an ancestor's definition. For example, you might want to indent a `Section` element and its descendants when it is nested within another `Section`. You could specify the change in indentation once, in a format rule for `Section`, and the descendants of `Section` would inherit this information:

Element (Container): Section
General rule: Head, (Para | List)+, Section*
Text format rules
1. **If context is:** Section
Basic properties
Move left indent by: +0.5"

If a Section is nested, the Section and its descendants are indented .5 inch for each level of nesting.

To write text format rules that are easy to maintain, you should normally define as little formatting information as possible in each element definition and let the elements inherit whatever properties they share with their ancestors. Using inheritance judiciously can greatly simplify your format rules.

The only table-part elements that can contain text are table titles and cells. Although you can write format rules for table, heading, body, footing, and row elements, in these cases the rules specify text formatting only for their descendant titles and cells.

In a document, if an end user applies a different paragraph format to an element or applies formatting changes to an element, the changes are considered format rule overrides. When the user re-imports element definitions, he or she can either leave the overrides as they are or remove the overrides so that the formatting in elements with text conforms to the text format rules.

How elements inherit formatting information

In a typical document, many elements can not only use the same paragraph format but also share changes to the format. Text formatting in FrameMaker is *hierarchical*, which means that elements can inherit all or part of their formatting information from ancestors. This lets you control common formatting information in parent elements.

It is even possible to have only one paragraph format for an entire document. The format is associated with the document's highest-level element, and all other elements inherit the format and specify changes to it when necessary.

The general case

When FrameMaker formats text in an element, it first determines which paragraph format to apply:

- If the element's definition specifies a base paragraph format, that format is used.

- If the element's definition does not specify a paragraph format, FrameMaker searches up through the element's ancestors until it finds an element with a format and then uses that format.

  In general, if FrameMaker reaches the top of the element's hierarchy and still has not found a format, it uses the default `Body` paragraph format for the document. (The behavior is somewhat different for an element in a table or footnote, or if the document is part of a book. See "Inheritance in a table or footnote" on page 209 or "Inheritance in a document within a book" on page 210.)

After FrameMaker takes a paragraph format from an ancestor or from the top of the hierarchy, it starts at that point and goes back down through the hierarchy to the current element, picking up formatting changes in format rules along the way. The changes modify the paragraph format cumulatively at each point. A format can have an *absolute value* (a fixed value, such as an indent expressed as distance from the left margin) or a *relative value* (a change to a current setting, such as an amount to move an indent). An absolute value replaces the same value in the paragraph format, and a relative value is added to the format's value to create a new value.

For example, in this set of definitions only the `Section` element has a base paragraph format (`body`). The descendants of `Section` all use the `body` paragraph format, and most also specify changes to it:

Element (Container): Section
General rule: Head, (Para | List)+, Section*
Text format rules
Element paragraph format: body
1. If context is: Section
Basic properties
Move first indent by: +0.5"
Move left indent by: +0.5"

This paragraph format applies to Section and its descendants.

Element (Container): Head
General rule: <TEXT>
Text formal rules
1. In all contexts
Default font properties
Weight: Bold
Size: 14pt

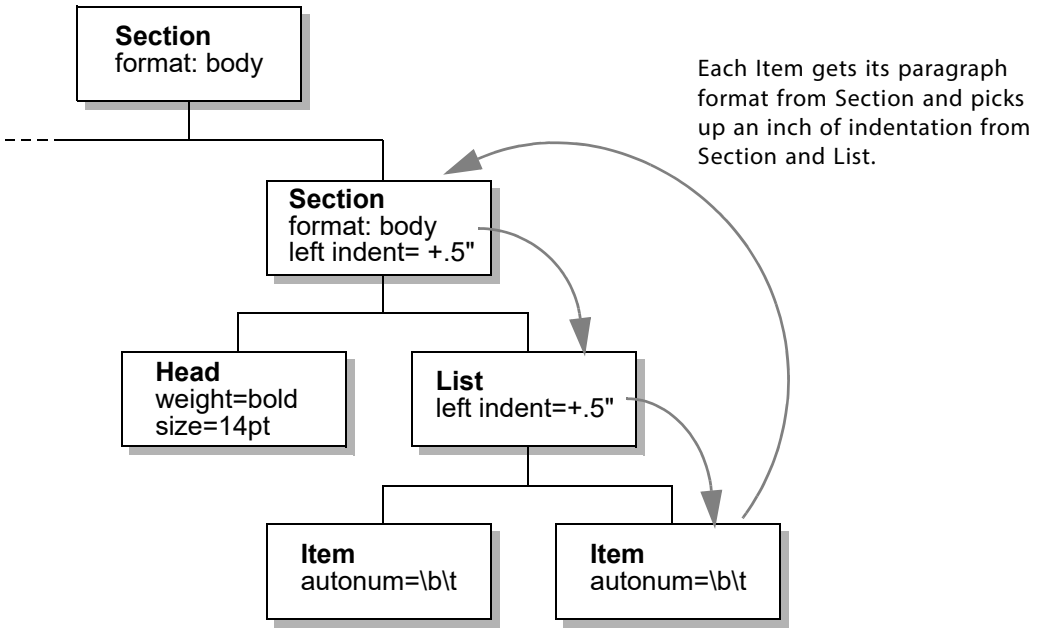
Element (Container): Para
General rule: <TEXT>

Element (Container): List
General rule: Item+
Text formal rules
1. In all contexts
Basic properties
Move left indent by: +0.5"

Element (Container): Item
General rule: <TEXT>
Text formal rules
1. In all contexts
Numbering properties
Autonumber format: \b\t

When no paragraph format is specified, the default format `Body` is used. Note that because case is significant in format tags, `body` in the example is not the same as the tag `Body`.

The following document structure uses these definitions. FrameMaker formats text in the `Item` elements by searching up to the second-level `Section` for a paragraph format. The `Item` elements are left-indented 0.5 inch more by the format rule for the second-level `Section` and another 0.5 inch by the `List` element's format rule. They also have a bullet provided by an autonumber in their own definition.



Note that when FrameMaker picks up changes from ancestors, it includes changes from the element with the paragraph format.

Inheritance in a table or footnote

If the current element is in a table or a footnote, FrameMaker searches through the element's ancestors for a paragraph format in the same way that it does for other elements, with these differences:

- For text in a table title or table cell, FrameMaker does not search beyond the ancestor table element. If it reaches the table element and still has not found a paragraph format, it uses the paragraph format stored in the table format being used for the appropriate type of table part.
- For text in an element in a footnote, FrameMaker does not search beyond the ancestor footnote element. If it reaches the footnote element and still has not found a paragraph format, it uses the document's current footnote paragraph format (if the footnote is in the main flow) or current table-footnote paragraph format (if the footnote is in a table).

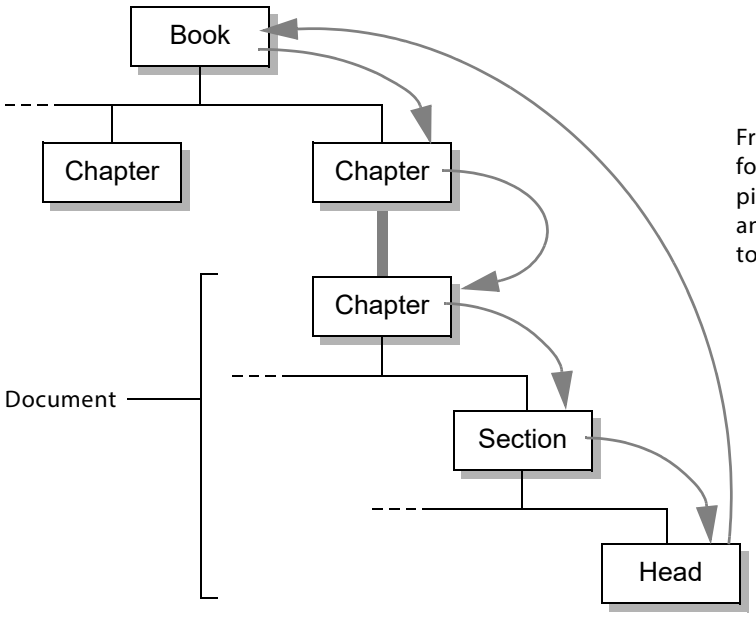
  The Paragraph Catalog in a document stores default `Footnote` and `TableFootnote` paragraph formats, and an end user can also define custom footnote paragraph formats. FrameMaker applies whichever format is specified in the document's Footnote Properties or Footnote Table Properties dialog box. For information on footnote paragraph formats, see the FrameMaker user's manual.

Inheritance in a document within a book

In a document that is part of a book, FrameMaker searches through an element's ancestors for a paragraph format in the same way that it does in other documents. If FrameMaker does not find a format in the document (when searching outside a table or footnote), it continues looking for one in the ancestor elements of the book file.

If FrameMaker finds a paragraph format in the book file, it uses that format. If it reaches the top of the element's hierarchy in the book and still has not found a format, it uses the default `Body` paragraph format stored in the document.

When FrameMaker uses a format from an ancestor in a book or from the top of the hierarchy, it starts at that point and then goes back down through the hierarchy to the current element, picking up formatting changes along the way.



FrameMaker can search for a format in a book file. Then it picks up changes in the book and brings them back down to the current element.

Only a document's main flow is considered to be part of a book's hierarchy. FrameMaker continues the search up into the book file if the current element is part of the main flow.

When an end user updates a book, the element hierarchy from the book file is stored in the book's documents. To apply a paragraph format, FrameMaker looks at the book's hierarchy in the document and then uses the appropriate format from the document's format rules. Closing or even deleting the book file does not affect the book information in a document; a user can regenerate the book without the document to remove the information.

If the end user adds a document to the book, the book's text format rules are not automatically applied to elements in the document. The user needs to update the book again.

Specifying an element paragraph format

An element definition can include a reference to a base paragraph format. If an instance of the element contains text, the text's format is the paragraph format plus any changes specified for the current context in the element's format rules. A paragraph format is also passed on to the element's descendants—until a descendant provides a different format.

To specify an element paragraph format, insert an `ElementPgfFormatTag` element as the first child element of `TextFormatRules`, and type the tag of the format. The tag must refer to a paragraph format stored in the Paragraph Catalog of the documents. For example, this element uses the `item` paragraph format, and a format rule specifies an autonumber format that is either a bullet or an incrementing number (in either case, followed by a tab):

Element (Container): Item
General rule: <TEXT>
Text format rules
Element paragraph format: item ——— The base paragraph format
1. **If context is:** BulletList
Numbering properties
Autonumber format: \b\t
Else, if context is: NumberList
Numbering properties
Autonumber format: <n+>\t

Usually, paragraph formats do not vary greatly from one element to another. Your definitions will be simpler and the catalogs easier to maintain if you specify paragraph formats for as few elements as necessary and let most elements inherit their format. You can use format rules to handle changes to the format for particular elements. For information on inheritance, see "How elements inherit formatting information" on page 206.

Table, heading, body, footing, and row elements cannot contain text themselves, but can hold other elements that do. If you specify a paragraph format for one of these elements, the format applies only to the element's descendants that can contain text.

A paragraph format can also be part of a context-dependent format rule so that it is used only in some cases. This is described in "Writing context-dependent format rules," next.

Writing context-dependent format rules

You can define changes to the element paragraph format in use with one or more format rules. The rules describe possible contexts in which an element can occur and give formatting changes for each context. When FrameMaker formats text in the element, it uses the current paragraph format (which may have been inherited and modified by ancestors' definitions), plus any format rules that apply to the current context.

A format rule provides context and formatting information:

- The rule can apply to all contexts in which the element occurs, or it can define particular contexts or the number of levels deep the element is nested in an ancestor. If the rule defines contexts or levels, it can have separate if, else/if, and else clauses for different possibilities.
- Each "in all contexts" rule or if, else/if, and else clause specifies formatting changes. The formatting changes can be a list of specific properties; or a reference to a different paragraph format, a character format (if you are formatting the element as a text range), or a list of properties stored elsewhere in the EDD.

For example:

Element (Container): Head
General rule: <TEXT>
Text format rules
1. **In all contexts.**
   Default font properties
   Weight: Bold
2. **If context is:** Section < Section
   Numbering properties
   Autonumber format: <n>.<n+>\t
   Else, if context is: Section
   Numbering properties
   Autonumber format: <n+>\t

A format rule clause has a context specification …

… and formatting changes for that context.

A format rule or clause can also include a context label to help end users select elements when inserting cross-references or preparing a table of contents or other generated list. An element can have more than one format rule, and format rules can be nested inside one another. The format rules are numbered automatically.

Chapter 17, "Context Specification Rules." describes the context specifications and context labels in format rules. For a summary of the formatting changes available, see "Defining the formatting changes in a rule" on page 212.

An element's format rules or paragraph format can also be inherited from an ancestor. For information on this, see "How elements inherit formatting information" on page 206.

Defining the formatting changes in a rule

Each clause in a text format rule includes formatting changes that apply to the element in the specified context. The changes can modify paragraph properties or text-range properties, or they can specify no additional formatting for the context.

Paragraph formatting

You can define paragraph-formatting changes for any container, footnote, table, or table part. Insert a `ParagraphFormatting` element after the context specification, and then specify one of these changes:

- Refer to a paragraph format stored in the document. A paragraph format is a fully specified set of properties—including font settings, indentation, line spacing, alignment, and autonumbering. The format replaces the paragraph format for this context only. (It does not become the base format for the element and is not inherited by descendants.)

  Insert the element `ParagraphFormatTag`, and type the tag of the paragraph format.

- Make changes to particular paragraph format properties. The changes modify the specified properties in the paragraph format in use.

  Insert the element corresponding to the group of properties (such as `PropertiesFont`), and define the changes. For a summary of these properties, see "Specifications for individual format properties" on page 215.

- Refer to a list of changes to properties stored elsewhere in the EDD. The changes in the list modify the specified properties in the paragraph format in use.

  Insert the element `FormatChangeListTag`, and type the tag of the change list. For information on defining a list, see "Defining a format change list" on page 234.

For example:

Element (Container): Head
General rule: <TEXT>
Text format rules
1. **If context is:** {first} < (Chapter | Appendix)
   Use paragraph format: chaptitle
   Else
   Use paragraph format: head
2. **Count ancestors named:** Section
   If level is: 1
   No additional formatting.
   Else, if level is: 2
   Default font properties
   Size: 14pt

Text range formatting

A *text range* in FrameMaker is a string of text within a paragraph; it often has different font properties from the paragraph text around it. Common examples of text ranges are emphasized phrases, book titles, and code fragments. You can format a container as a text range to apply font changes and to override or replace paragraph properties.

Insert a `TextRangeFormatting` element after the context specification, and then specify one of these changes:

- Refer to a character format stored in the document. The character format replaces the font properties in the paragraph format for the element in this context.

  Insert the element `CharacterFormatTag`, and type the tag of the format.

- Make changes to particular font properties. The changes modify the specified font properties in the paragraph format in use.

  Insert the `PropertiesFont` element, and define the changes. For information on the properties, see "Font properties" on page 219.

- Refer to a list of changes to properties stored elsewhere in the EDD. The changes in the list modify the specified font properties in the paragraph format in use.

  Insert the element `FormatChangeListTag`, and type the tag of the change list. For information on defining a list, see "Defining a format change list" on page 234.

For example:

Element (Container): Emphasis
General rule: <TEXT>
Text format rules
1. **In all contexts.**
Text range.
Font properties
Angle: Oblique

A text range label appears when you format a container as a text range.

Element (Container): CodeFragment
General rule: <TEXT>
Text format rules
1. **In all contexts.**
Text range.
Use format change list: Code

If you use a format change list with a text range, only the font properties from the list are applied.

No additional formatting

You can also specify no formatting changes, and the element in that context will be formatted according to the current paragraph format (which may have been inherited and modified by ancestors' format rules). Insert a `NoAdditionalFormatting` element after the context specification.

The no-formatting option provides a way to express "if not" in a multiple-clause format rule. For example, this rule specifies that if a `Head` does *not* appear in a `LabeledPar` or `Sidebar` element, the formatting changes are made as described:

Element (Container): Head
General rule: <TEXT>
Text format rules
1. **If context is:** * < (LabeledPara | Sidebar)
No additional formatting.
Else
1.1.Count ancestors named: Section
If level is: 1
No additional formatting.
Else, if level is: 2
Default font properties
Size: 12pt
Weight: Bold

The formatting changes apply if a Head does not appear in this context.

This clause is not necessary but makes the nested rule (1.1) more readable.

In some cases, you may also want to use the no-formatting option to improve the readability of a format rule. In the nested rule (1.1) in the example above, you could leave out the `If level is: 1` clause and begin with `If level is: 2`. But the level-1 information makes the rule more comprehensive and will remind anyone reading the rule later that the paragraph format in use is correct for `Head` elements in a first-level `Section`.

Specifications for individual format properties

A format rule clause can describe changes to any property available in the Paragraph Designer (except for Next Paragraph Tag, which is not used in structured documents). These properties include indentation, alignment, line spacing, font and style settings, autonumbering, paragraph straddle formats, and hyphenation limits.

The design properties in a rule clause are organized in the groups Basic, Font, Numbering, Pagination, Advanced, and Table Cell—analogous to the layout of these properties on pages in the Paragraph Designer. To specify a set of properties, insert the element that corresponds to the group as a child element in the `ParagraphFormatting` or `TextRangeFormatting` element, and then proceed with the individual properties. For example:

Element (Container): Head
General rule: <TEXT>
Text format rules
1. **If context is:** Section < Section
Basic properties
Paragraph spacing
Change space above by: +2pt
Default font properties
Weight: Bold
PairKerning: Yes
Numbering properties
Autonumber format: <n>.<n+>\t

Properties are organized in groups analogous to pages in the Paragraph Designer.

With some properties you type the value, and with others you use a keyword child element (such as `Bold` or `Yes`) to express the value. Most of the numeric values you can enter are absolute, but a few values in the Basic, Font, and Table Cell groups are relative. A relative value can be positive or negative. (The plus sign is not required with a positive value.) You cannot enter both an absolute value and a relative value for a property.

An absolute value overrides the current value for the property in the paragraph format in use. A relative value is added to the current value to set a new value. For example, suppose an element inherits a paragraph format with a left indent of .25 inch:

- With a `LeftIndent` value of 1 inch, the new left indent is 1 inch.
- With a `LeftIndentChange` value of 1 inch, the new left indent is 1.25 inch.

You can specify a unit of measure, such as `pt` or `in`, for indentation and tab stops in the Basic properties and the frame position in the Advanced properties; if you do not specify a unit, FrameMaker uses the one set in View Options in the document. Offsets and spreads in Font properties and word spacing in Advanced properties are always in percentages. For any other numeric value, if you specify a unit it is converted to points.

These are the possible units of measure:

Unit	Notation in the EDD
Centimeters	`cm`
Millimeters	`mm`
Inches	`"` or `in`
Picas	`pc`, `pi`, or `pica`
Points	`pt` or `point`
Didots	`dd`
Ciceros	`cc` or `cicero`
Percentage	`%`

Properties that can use relative values have minimum and maximum limits. Font sizes, for example, must fall within the inclusive range 2 to 400 points. If you set a value that is outside an allowed range (either by specifying an absolute value or by calculating a new relative value), FrameMaker changes the value to the minimum or maximum. You can also set your own minimum and maximum limits. For a summary of the minimum and maximum limits and information on changing them, see "Setting minimum and maximum limits on properties" on page 235.

For more details on the formatting properties and guidelines on how to use them, see the FrameMaker user's manual.

Basic properties

The Basic properties set indentation, line spacing, paragraph alignment, paragraph spacing, and tab stops. To begin changing these properties, insert a `PropertiesBasic` element in the `ParagraphFormatting` element.

The numeric settings for Basic properties all allow an absolute value or a relative value. Use the elements with `Change` in the tag for relative values.

Indentation, spacing, and alignment

The following elements and values define the properties for indentation, spacing, and alignment:

`Indents` — Distances from the left and right edges of the text column to the text.

- `FirstIndent` *`dimension`* Sets the left indent for the first line in a paragraph. Type the distance from the left edge of the text column.
- `FirstIndentRelative` *`dimension`* Sets the left indent for the first line in a paragraph (relative to the left indent in use). Type a relative value.
- `FirstIndentChange` *`dimension`* Sets the left indent for the first line in a paragraph (added to the current first indent). Type a relative value.
- `LeftIndent` *`dimension`* Sets the left indent for all lines in a paragraph after the first line. Type the distance from the left edge of the text column.
- `LeftIndentChange` *`dimension`* Sets the left indent for all lines in a paragraph after the first line (added to the current left indent). Type a relative value.
- `RightIndent` *`dimension`* Sets the right indent for all lines in a paragraph. Type the distance from the right edge of the text column.
- `RightIndentChange` *`dimension`* Sets the right indent for all lines in a paragraph (added to the current right indent). Type a relative value.

`LineSpacing` — Vertical space between lines in a paragraph, measured from baseline to baseline. (See the note on font size and line spacing after this table.)

- `Height` *`dimension`* Sets the space between lines in a paragraph. Type the distance from one baseline to the next.
- `HeightChange` *`dimension`* Sets the space between lines in a paragraph (added to the current line spacing). Type a relative value.
- `Fixed` Keeps line spacing the same everywhere in a paragraph.

`NotFixed` Allows line spacing in a paragraph to change to accommodate the largest font on each line.

`PgfAlignment` *`keyword`*
Sets left, center, right, or justified for the horizontal position of a paragraph within the left and right indents. Insert a keyword child element from the catalog to specify the type of alignment.

`ParagraphSpacing`
Vertical space above and below a paragraph, measured from the baseline of the first and last lines.

`SpaceAbove` *`dimension`* Sets the space above a paragraph. Type the distance above the baseline of the first line.

`SpaceAboveChange` *`dimension`* Sets the space above a paragraph (added to the current space above). Type a relative value.

`SpaceBelow` *`dimension`* Sets the space below a paragraph. Type the distance below the baseline of the last line.

`SpaceBelowChange` *`dimension`* Sets the space below a paragraph (added to the current space below). Type a relative value.

When you change a font size in a document or in an EDD, FrameMaker automatically recalculates the paragraph's line spacing for you. (In a single-spaced paragraph, the default spacing is 120 percent of the font size.) You can change a font size and specify your own line spacing for it rather than using the calculated spacing. When doing this in an EDD, note the following behavior:

- If you change the font size and the line spacing in a single rule clause, the spacing is changed to the value you set, overriding any value that FrameMaker calculates for the new font size.
- If you change the font size and the line spacing in two different rule clauses, the line spacing clause must come *after* the font size clause for your spacing to override the calculated value. (If the font size comes second, FrameMaker calculates the spacing at that point and overrides the value you set earlier.)

For determining the space between two adjacent paragraphs, FrameMaker uses the space below the first paragraph or the space above the second paragraph, whichever is larger. If the paragraph is at the top of a column, the `SpaceAbove` value is ignored; at the bottom of a column, the `SpaceBelow` value is ignored.

Tab stops

To set tab stops for an element, insert a `TabStops` element in `PropertiesBasic` and then continue with the following elements and values. A single format rule clause can have one or more `TabStop` elements *or* one `MoveAllTabStopsBy` element *or* one `ClearAllTabStops` element.

`TabStop`
Definition of one tab stop.

`TabStopPosition` *`dimension`* Sets the location that the insertion point jumps to when an end user presses Tab. Type the distance from the left edge of the text column to the tab stop.

	`RelativeTabStopPosition` *dimension* Sets the location that the insertion point jumps to when an end user presses Tab. Type a relative value from the left indent of the paragraph to the tab stop. A positive value moves the tab stop right; a negative value moves it left. `TabAlignment` *keyword* Sets left, center, right, or decimal alignment for tabbed text on a tab stop. Insert a keyword child element from the catalog to specify the type of alignment. `AlignOn` *character* For decimal tabs, specifies a character on which to align the tabs. Type one character (usually a period or a comma). `Leader` *string* Specifies characters to repeat in a line to create a leader between a tab and the character following it. Type one or more characters (can be any characters including spaces).
`MoveAllTabStopsBy` *real-number*	Changes the positions for all inherited tab stops (added to the current tab stop positions). Type a relative value. A positive value moves the tab stops right; a negative value moves them left.
`ClearAllTabStops`	Removes all inherited tab stops.

Font properties

The Font properties set the font, size, and style of text in an element. To begin changing these properties, insert a `PropertiesFont` element in the `ParagraphFormatting` or `TextRangeFormatting` element. In a `ParagraphFormatting` element, the label you see in the EDD is `Default font properties`; in `TextRangeFormatting`, the label is `Font properties`.

Most of the Font properties use a child element to set the value, and in many cases FrameMaker inserts a default child element for you. You can select the child element and change it to a different one if you need to.

The following elements and values define the Font properties:

`Angle` *keyword*	Sets a font angle (such as italic or cursive). Insert a keyword child element to specify the angle.
`Case` *keyword*	Sets a capitalization style (such as lowercase or small caps). Insert a keyword child element to specify the case.
`ChangeBars` *boolean*	If Yes, displays a vertical line in the page margin where an end user has made changes to text.
`Color` *name*	Specifies a color for text. Type the name of a color defined in Color Definitions in the document.
`CombinedFont` *name*	Specifies a combined font. Type the name of the combined font that is available to your users. Note that the combined font must be defined in the FrameMaker document that will use this EDD.

`Family` *`name`*	Specifies a typeface family (such as Times or Helvetica). Type the name of a font that is available to your end users.
`OffsetHorizontal` *`real-number`*	Moves a text range element. Type the percentage of an em space you want the element to move. (FrameMaker interprets a value of 20 as 20% of an em space.) A positive value moves the element right; a negative value moves it left.
`OffsetVertical` *`real-number`*	Moves a text range element. Type the percentage of an em space you want the element to move. (FrameMaker interrupts a value of 20 as 20% of an em space.) A positive value moves the element up; a negative value moves it down.
`Overline` *`boolean`*	If `Yes`, places a line over text.
`PairKerning` *`boolean`*	If `Yes`, turns on ligatures and moves character pairs closer together as necessary to improve appearance. The ligatures, character pairs, and amount of kerning depend on the font.
`Size` *`dimension`*	Sets a point size for text. Type a number of points from 2 to 400. (The notation `pt` or `point` is optional.) (See the note on font size and line on page 218.)
`SizeChange` *`real-number`*	Sets a point size for text (added to the current point size). Type a relative value in points. (The notation `pt` or `point` is optional.) (See the note on font size and line on page 218.)
`Spread` *`real-number`*	Renamed to `Tracking` (see below). `Spread` will still work, for backward compatibility.
`SpreadChange` *`real-number`*	Renamed to `TrackingChange` (see below). `SpreadChange` will still work, for backward compatibility.
`Stretch` *`real-number`*	Sets the amount to stretch or compress the characters. Type a percentage of the font's em space. (The symbol `%` is optional.) A positive value stretches the characters; a negative value compresses them. Normal stretch is 0 percent.
`StretchChange` *`real-number`*	Sets the amount to stretch or compress the characters (added to the current stretch). Type a relative value as a percentage of the font's em space. (The symbol `%` is optional.)
`Strikethrough` *`boolean`*	If `Yes`, places a line through text.
`Superscript` `Subscript` *`keyword`*	Changes characters to a script above or below the baseline. Insert a keyword child element to specify superscript or subscript. The text size and the amount of offset are determined by Text Options in the document.
`Tracking` *`real-number`*	Sets the space between characters. Type a percentage of the font's em space. (The symbol `%` is optional.) A positive value increases the spread; a negative value decreases the spread. Normal spread is 0 percent.

`TrackingChange` *`real-number`*	Sets the space between characters (added to the current tracking). Type a relative value as a percentage of the font's em space. (The symbol `%` is optional.)
`Underline` *`keyword`*	Sets an underline style (such as double underline or numeric underline). Insert a keyword child element to specify the style.
`Variation` *`keyword`*	Sets a font variation (such as compressed or expanded). Insert a keyword child element to specify the variation.
`Weight` *`keyword`*	Sets a font weight (such as bold or black). Insert a keyword child element to specify the weight.
`Background-`*`color`*	Sets the background color for the paragraph text. Type the background color for the paragraph text.

Pagination properties

The Pagination properties affect the placement of a paragraph on a page and determine how to break the paragraph across columns and pages. To begin changing these properties, insert a `PropertiesPagination` element in the `ParagraphFormatting` element.

The following elements and values define the Pagination properties:

`KeepWithNext` *`boolean`*	If `Yes`, keeps a paragraph in the same text column as the next paragraph.
`KeepWithPrevious` *`boolean`*	If `Yes`, keeps a paragraph in the same text column as the previous paragraph.
`Placement`	Placement of a paragraph on a page. `AcrossAllCols` Makes a paragraph straddle the width of all columns in its text frame. `AcrossColsSideHeads` Makes a paragraph straddle the width of the columns and the side-head area. `InColumn` Keeps a paragraph in its text column so that it does not straddle other columns. `RunInHead` Displays a paragraph as a head that runs into the next paragraph. A run-in head can have default punctuation. `SideHead` Displays a paragraph as a side head across from the next paragraph. A side head can have default punctuation. `DefaultPunctuation` *`string`* For a run-in head or a side head, specifies punctuation to appear after the head. Type one or more characters (can be any characters including spaces). `Alignment` *`keyword`* For a side head, aligns the baseline of the head with the first baseline, top baseline, or top edge of the paragraph across from it. Insert a keyword child element to specify the type of alignment.

`StartPosition`	Location in a text column or page where a paragraph always begins. `Anywhere` Starts a paragraph right below the preceding one. The paragraph's widow/orphan setting determines where it breaks across text columns. `TopOfColumn` Starts a paragraph at the top of the next text column. `TopOfLeftPage`, `TopOfPage`, or `TopOfRightPage` Starts a paragraph at the top of the next specified page.
`WidowOrphanLines` *`integer`*	Sets the minimum number of lines in a paragraph that can appear alone at the top or bottom of a text column. Type a value from 0 to 100.

Numbering properties

The Numbering properties specify the syntax and format for an automatically generated string, such as a number that appears at the beginning of a procedure step. To begin changing these properties, insert a `PropertiesNumbering` element in the `ParagraphFormatting` element. (An autonumber can also be a fixed text string, such as the word *Note* at the beginning of a paragraph.)

If a paragraph element with an autonumber also has a prefix or suffix, the prefix appears just after the autonumber at the beginning of the paragraph, and the suffix appears just before the autonumber at the end of the paragraph.

To read about the syntax of autonumbers and the building blocks available, see the FrameMaker user's manual.

The following elements and values define the Numbering properties:

`AutonumberFormat` *`syntax`*	Specifies the style and incrementing of automatically generated numbers or bullets. Type text and building blocks to define the syntax, or insert building block child elements.
`AutonumCharFormat` *`tag`*	Applies a character format to an autonumber. Type the tag of a character format stored in the document.
`NoAutonumber` *`boolean`*	If `Yes`, turns off autonumbering for the element.
`Position` *`keyword`*	Displays an autonumber at either the beginning of its paragraph or the end of its paragraph. Insert a keyword child element to specify the position.

Advanced properties

The Advanced properties set hyphenation and word spacing options and determine whether to display a graphic with a paragraph. To begin changing these properties, insert a `PropertiesAdvanced` element in the `ParagraphFormatting` element.

The following elements and values define the Advanced properties:

`FrameAbove` *`name`*
: Displays a graphic from a reference frame above a paragraph. Type the name of a frame stored on a reference page in the document. You can also set a position for the frame.

`FrameBelow` *`name`*
: Displays a graphic from a reference frame below a paragraph. Type the name of a frame stored on a reference page in the document. You can also set a position for the frame.

`FramePosition` *`boolean`*
: Specifies whether the left alignment of the frame above or the frame below matches the text column or the paragraph indent. `Yes` specifies that it matches the paragraph indent.

`Hyphenation`
: Hyphenation properties for all text in an element.

  `Hyphenate` *`boolean`* if `Yes`, turns on automatic hyphenation for the element.

  `Language` *`string`* specifies the language to use for hyphenation rules.

  `MaxAdjacent` *`integer`* Sets the maximum number of consecutive lines that can end with a hyphen. Type a value.

  `ShortestPrefix` *`integer`* Sets the minimum number of letters in a word that can precede a hyphen. Type a value.

  `ShortestSuffix` *`integer`* Sets the minimum number of letters in a word that can follow a hyphen. Type a value.

  `ShortestWord` *`integer`* Sets the minimum length of a hyphenated word. Type a value.

`WordSpacing`
: The amount or word spacing that FrameMaker can decrease or increase in an element. These settings are percentages of the standard word spacing for the font. Normal spacing is 100 percent. Values below 100 allow tighter spacing; values above 100 allow looser spacing.

  `LetterSpacing` *`boolean`* If `Yes`, allows additional space between characters in justified text to keep the space between words from going over the maximum.

  `Maximum` *`integer`* Sets the largest space allowed between words before FrameMaker hyphenates words or adds letter spacing in justified paragraphs. Type a percentage of the font's em space. (The symbol `%` is optional.)

	`Minimum` *`integer`* Sets the smallest space allowed between words. Type a percentage of the font's em space. (The symbol `%` is optional.) `Optimum` *`integer`* Sets the optimum amount of space between words. Type a percentage of the font's em space. (The symbol `%` is optional.)
`PgfBoxColor`	Sets the background color for the area around the bounding box of the paragraph.

Table Cell properties

The Table Cell properties customize the margins of cells and the vertical alignment of text in them. To begin changing these properties, insert a `PropertiesTableCell` element in the `ParagraphFormatting` element.

A margin or alignment you set with these properties overrides the default properties for a table. For information about how custom margins and alignments work with table formats, see the FrameMaker user's manual.

The following elements and values define the Table Cell properties:

`CellMargins`	Margins between the borders of a cell and the text in the cell. `Bottom`, `Left`, `Right`, or `Top` Specifies the margin to change. For each margin, you can set a custom value or a value that is relative to the table format's margin. `Custom` *`dimension`* Type the distance in points from the border of the cell to the text. (The notation `pt` or `point` is optional.) `FromTblFormatPlus` *`dimension`* Type a relative value in points that is added to the default margin set in the table format. (The notation `pt` or `point` is optional.)
`VerticalAlignment` *`keyword`*	Sets top, middle, or bottom alignment for text in a table cell. Insert a keyword child element from the catalog to specify the type of alignment.

Asian Text Spacing properties

Typographic rules for variable width Asian text require specifications for spacing between Asian characters, Western and Asian characters, and for punctuation characters. To begin changing these properties, insert a `PropertiesAsianSpacing` element in the `ParagraphFormatting` element.

These properties only take effect when displaying a document on a system running Asian system software.

The following elements and values define the Asian text spacing properties:

`WesternAsianSpacing`	The spacing between pairs of Western and Asian characters.
	`Minimum`, `Maximum`, and `Optimum` Specifies the range of spacing percentages to use, and the optimum, or preferred, percentage of spacing to use.
`AsianAsianSpacing`	The spacing between pairs of Asian characters.
	`Minimum`, `Maximum`, and `Optimum` Specifies the range of spacing percentages to use, and the optimum, or preferred, percentage of spacing to use.
`Punctuation`	`Float`, `Fixed`, `Monospace` Specifies the type of character squeezing to use for punctuation characters.

Direction properties

The elements in a structured document are directional. This implies that the content in an element can authored and read left-to-right (for languages such as English, German, or French) or right-to-left (for Arabic or Hebrew). To enable directional support in structured applications, the `dir` EDD construct is included in the ParagraphDirection to specify the direction of the element.

You can use the dir property to define the following directions:

`ltr`	Sets the direction of the element to left-to-right.
`rtl`	Sets the direction of the element to right-to-left.
`inherit`	Sets the direction of the element to inherit from the parent element.

In this example, the `Paragraph` definition specifies that the *dir* property provides three choices: ltr, rtl, inherit.

Element (Container): Paragraph
General rule: <TEXT>
Attribute list
1. **Name:** dir **String** **Optional**
   Choices: ltr,rtl,inherit

Text format rules
1. **In all contexts.**
   PropertiesDirection
   ParagraphDirection:
   inherit

FrameMaker layout engine (Asian Composer)

FrameMaker includes an Asian Composer construct in the EDD for paragraph properties. The property defines the layout engine the element will use.

Whenever you select a font that has double byte encoding or is an CJK language font in paragraph designer, you need to check the Use Asian Composer option.

The following example contains two paragraph definitions:

Element (Container): Para_pgffmt
General rule: <Text>
Text format rules
1. **In all contexts.**
Asian spacing properties
Asian Composer: Yes

Element (Container): Para_pgfppt1
General rule: <Text>
Text format rules
1. **In all contexts.**
Asian spacing properties
Asian Composer: No

Writing first and last format rules

You can apply a special set of format rules to the first and last paragraphs in an element. This is particularly useful for parent elements that are only containers for child elements that are formatted as paragraphs (often all of the same type), such as a `List` that contains `Item` elements or a `Chapter` that contains `Section` elements.

To write first or last format rules, insert a `FirstParagraphRules` or `LastParagraphRules` element at the same level as `TextFormatRules`, and define the context and formatting specifications as you do for other format rules. For information on the internal specifications of rule clauses, see "Writing context-dependent format rules" on page 211 and "Defining the formatting changes in a rule" on page 212.

For example, these first and last format rules display a line from a reference page above and below a `List` element:

Element (Container): List
General rule: Head?, Item+
Text format rules
1. **In all contexts.**
Basic properties
Indents
Move left indent by: +12pt
Format rules for first paragraph in element
1. **In all contexts**
Advanced properties
Frame above: SingleLine
Format rules for last paragraph in element
1. **In all contexts**
Advanced properties
Frame below: SingleLine

The first and last rules display a line above and below a List.

Note that if the first and last paragraphs are child elements, you can get the same first and last formatting using `{first}` and `{last}` sibling indicators in format rules for the child elements. The first and last format rules are more natural with the hierarchical model, however. Because the properties are associated with the parent, they are inherited by any paragraph that happens to be the first or last child—so you need to define the special formatting only once. (In the `List` example above, with `{first}` and `{last}` indicators, you would need to define the lines for both the `Head` and the `Item` child elements.)

There are some similarities between the uses of first and last format rules, autonumber strings, and prefixes and suffixes. For guidelines on using the different constructs, see "When to use an autonumber, prefix or suffix, or first or last rule" on page 233.

How first and last rules are applied

The first or last format rules in an element apply to the first or last paragraph in the element. The paragraphs may be child elements, or they may just be text formatted as paragraphs. The rules are ignored in contexts in which a first or last child element is formatted as a text range.

When FrameMaker formats an element in a document, it applies format rules in this order:

- The element's text format rules
- The element's first or last format rules
- The element's prefix or suffix format rules
- The text format rules in any child elements

If an element has a prefix formatted in a separate paragraph, the prefix is the first paragraph. Similarly, a suffix formatted in a separate paragraph is the last paragraph.

If the element has no text content, no child elements, and no prefix or suffix as a separate paragraph, the first or last rules apply to the element itself.

A first or last rule with an autonumber

A first or last rule can be used with an autonumber to display the number or string with only the first or last paragraph in the element. For example, this rule displays the string *Important:* or *Note:* at the beginning of the first `Para` in `Note`:

Element (Container): Note
General rule: Para+
Format rules for first paragraph in element
1. **If context is:** [Important = Yes]
Numbering properties
Autonumber format: Important:
Character format: Bold
Else
Numbering properties
Autonumber format: Note:
Character format: Bold

If an element contains a single paragraph, and the first and last rules both specify an autonumber, only the first rule is used.

A first or last rule can also apply to a prefix or suffix that is formatted in a paragraph of its own. Because the formatting part of prefix and suffix rules allows only specification of font changes, you can use a first or last rule to give the prefix or suffix special paragraph formatting properties. For an example of this, see "A prefix or suffix for a sequence of paragraphs" on page 230.

Defining prefixes and suffixes

A *prefix* is a text range defined in the EDD that appears at the beginning of an element (before the element's content); a *suffix* is a text range that appears at the end of an element (after the content). In many cases, a prefix or suffix is formatted differently than other text in the element.

You can define a prefix or suffix for any container element in FrameMaker, using a set of rules similar to other format rules. Prefix and suffix rules describe both the text string and any special font properties for it.

To define a prefix or suffix, insert a `PrefixRules` or `SuffixRules` element at the same level as `TextFormatRules`, and write one or more rules for the prefix or suffix. In one of the rules, you need to specify a text string, using the `Prefix` or `Suffix` child element. The string can include any characters (including tabs and spaces) and one or more attribute building blocks.

If you want to include a left angle bracket (<) in the string, escape the bracket with a backslash, like this: \<. (A left angle bracket by itself begins an attribute building block.)

The format rules for a prefix or suffix can use the same context specifications available for other format rules and any of the font changes for text ranges. For information on the internal specifications of rule clauses, see "Writing context-dependent format rules" on page 211 and "Text range formatting" on page 213.

You can display a prefix or suffix with element text in a cross-reference or in a running header or footer in a document. Use the `$elemtext` (rather than the `$elemtextonly`) building block in the cross-reference format or in the header or footer definition. For more information, see the FrameMaker user's manual.

There are some similarities between the uses of first and last format rules, autonumber strings, and prefixes and suffixes. For guidelines on when to use the different constructs, see "When to use an autonumber, prefix or suffix, or first or last rule" on page 233.

How prefix and suffix format rules are applied

The format rules for a prefix or suffix describe font changes only for the prefix or suffix. The changes do not apply to descendants of the element.

> **Important:** Font changes for a prefix/suffix do not take effect if the prefix/suffix begins with a forced return.

When FrameMaker formats an element with a prefix or suffix, it applies format rules in this order:

- The element's text format rules
- The element's first or last format rules
- The element's prefix or suffix format rules

If the element has first or last rules and the prefix or suffix is formatted in a paragraph of its own, the prefix or suffix is the first or last paragraph for the purposes of formatting. Because of the order that rules are applied, a prefix or suffix format rule can override font changes in a first or last format rule.

A prefix or suffix for a text range

You can use a prefix or suffix to provide a string for a text range element inside a paragraph. For example, suppose you want double quotation marks to appear around the text of a quotation every time, without the end user having to type in the marks. You can set up a pair of quotation marks for the `Quotation` text range element as a prefix and a suffix:

Element (Container): Quotation
General rule: <TEXT>
Text format rules
1. **In all contexts.**
Text range.
No additional formatting.

Prefix rules
1. **In all contexts.**
Prefix: “

Suffix rules
1. **In all contexts.**
Suffix: ”

In this example, the prefix and suffix do not have any font changes, so they are formatted the same as other text in the `Quotation` element.

If you want fixed text to appear at the beginning or end of a paragraph rather than inside the paragraph, define the string as a prefix, suffix, or autonumber for the paragraph element. For information on autonumbers, see “Numbering properties” on page 222.

A prefix or suffix for a paragraph

If you define a prefix for a paragraph element that begins with text, the prefix appears at the beginning of the paragraph; if you define a suffix for an element that ends with text, the suffix appears at the end of the paragraph. (In these cases, a prefix for suffix is similar to an autonumber for the paragraph.) For example, you might use a prefix to display *Important:* at the beginning of a paragraph:

Element (Container): Note
General rule: <TEXT>
Prefix rules
1. **In all contexts.**
Prefix: Important:

If a paragraph element also has an autonumber, the prefix appears after an autonumber at the beginning of the paragraph, and a suffix appears before an autonumber at the end of the paragraph.

A prefix or suffix for a sequence of paragraphs

You can define a prefix or suffix for a paragraph element that contains other paragraphs. If the element begins with a paragraph child element, the prefix appears in a paragraph of its own; if the element ends with a paragraph child, the suffix appears in a paragraph of its own. This is especially useful for displaying a string with an element that has no text of its own but is only a parent for other paragraphs.

For example, suppose you want FrameMaker to display heads automatically for syntax descriptions that can have several paragraphs. If you have a `Syntax` element that is a parent for the paragraphs in a description, you can set up a prefix that provides a head appropriate for the context of `Syntax`:

Element (Container): Syntax
General rule: Para+
Prefix rule
1. **If context is:** Synopsis
   Prefix: Synopsis and contents
   Else, if context is: Args
   Prefix: Arguments
   Else, if context is: Examples
   Prefix: Examples
2. **In all contexts.**
   Text range.
   Font properties
   Weight: Bold

A syntax description begins with a bold head in a separate paragraph.

Format rules for first paragraph in element
1. **In all contexts**
   Basic properties
   Paragraph spacing
   Space below: 4pt

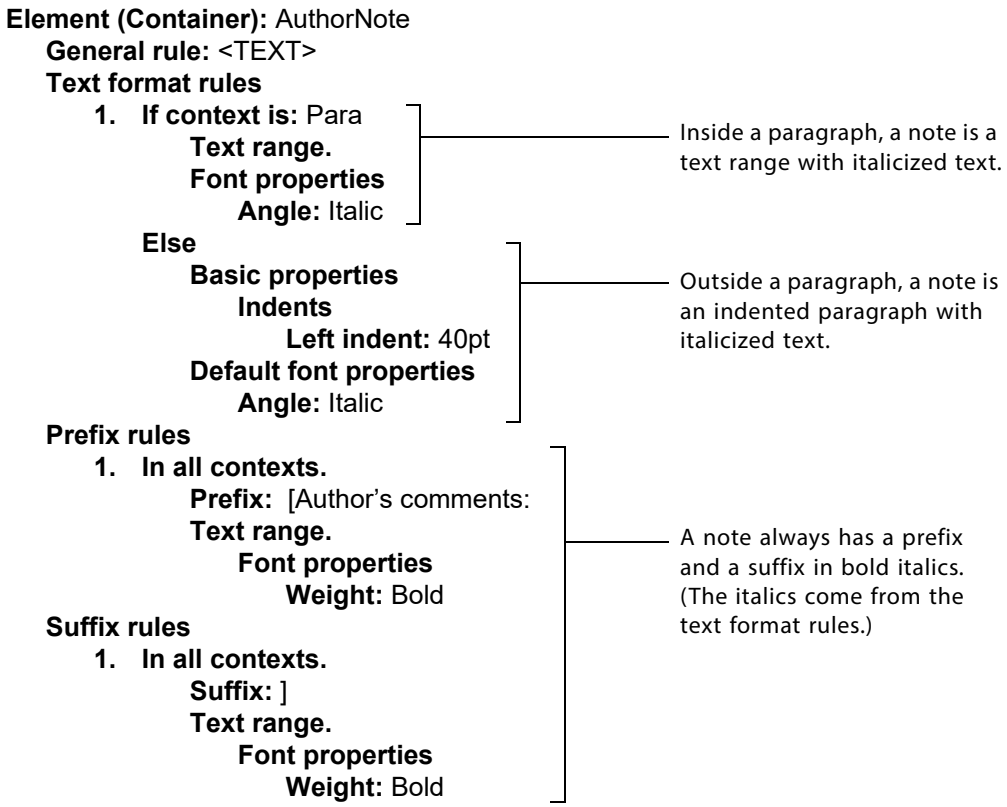
The head paragraph has extra space below it.

In each description, the boldfaced string *Synopsis and contents*, *Arguments*, or *Examples* appears in a paragraph by itself above the first child `Para`.

Note that in a prefix or suffix rule you can specify formatting changes only to font properties. If you are displaying a prefix or suffix in a paragraph by itself and want to apply paragraph changes to it such as space above or below, give the parent element a first format rule (for a prefix) or a last format rule (for a suffix). The prefix is the first paragraph in the parent, or the suffix is the last paragraph in the parent. In the `Syntax` example above, the first format rule puts 4 points of space below the prefix head.

A prefix or suffix for a text range or a paragraph

A single prefix or suffix can work with both a text range and a paragraph. For example, suppose you want to allow author annotations that could be run in with regular text or set off in a paragraph of their own. You can define the element to be formatted as a text range or as a paragraph, and the prefix and suffix will apply in either case:



The prefix string in the example has a space at the beginning and the suffix string has a space at the end, so that within a paragraph the annotation will have extra space on each side.

Note that FrameMaker first applies the text format rules to the entire element and then applies the prefix and suffix rules. In the example above, the prefix and suffix strings are in bold italics because they get their angle property (italics) from the text format rules.

Attributes in a prefix or suffix rule

You can refer to an attribute in a prefix or suffix rule, and the value from the attribute is written as the text string. If the attribute has only a default value, that value is used. This allows you to define one prefix or suffix for different places in a document.

To use an attribute value, insert an `AttributeValue` element in `Prefix` or `Suffix`. The text `<$attribute[]>` appears in the prefix or suffix definition. Include one or more of the following building blocks in the definition:

<$attribute[*attrname*(:*elemtag1*, *elemtag2*, …)]>

where *attrname* is the name of an attribute and each optional *elemtag* is the tag of an element that has the named attribute.

If you do not include an element tag, FrameMaker uses the value for the named attribute in the current instance of the element. For example, this definition displays the value of the `Security` attribute in the current element:

Prefix: <$attribute[Security]>

If you do include element tags, FrameMaker uses the value for the named attribute in the closest containing element with a tag that matches *elemtag* and an attribute that matches *attrname*. For example, this definition displays the value of the `Security` attribute in the closest containing `Item` or `LabelPara` that has the attribute:

Prefix: <$attribute[Security: Item, LabelPara]>

The element specification can include a context label in parentheses. FrameMaker finds the closest containing element with a tag that matches *elemtag* and the specified context label. For example, this definition displays the value of the `Security` attribute from the closest containing `Head` element that has a `Chapter Level` context label and a `Security` attribute:

Prefix: <$attribute[Security: Head(Chapter Level)]>

If you use empty parentheses or the token `<nolabel>` in parentheses, FrameMaker finds only elements without a context label. For example, the specification `Head()` or `Head(<nolabel>)` finds `Head` elements with no label.

When to use an autonumber, prefix or suffix, or first or last rule

You can display a fixed string of text with an element by using an autonumber or a prefix or suffix. An autonumber or prefix can be at the beginning of a paragraph; an autonumber or suffix can be at the end of a paragraph. But a prefix or suffix provides additional flexibility because you can also display the string inside a paragraph (with a text range element) or in a paragraph of its own (with a parent element). In addition, a paragraph can have both a prefix and a suffix but can have an autonumber only at one end.

A first or last format rule applies special paragraph formatting to the first or last paragraph in an element.

Here are a few cases in which you may want to use an autonumber, a prefix or suffix, or a first or last rule:

- To display a text string at the beginning or end of an element formatted as a paragraph, define the text as an autonumber or a prefix or suffix for the element. (For example, you might put *Important:* at the beginning of a paragraph.)

  If you want the string to appear with the first or last paragraph in a parent, define an autonumber in a first or last format rule for the parent. (For example, you might put *Important:* at the beginning of the first paragraph in a set-off note with several paragraphs.)

- To display a text string in the middle of an element formatted as a paragraph without breaking the paragraph, define the text as a prefix or suffix for a text range child element. (For example, you might put quotation marks around a quotation text range, or separate fields in a bibliography entry with appropriate punctuation and spaces.)
- To display a text string in a paragraph of its own at the beginning or end of a parent element that is only a container for paragraph elements, define the text as a prefix or suffix for the parent. (For example, you might put a head at the top of a section.)

  You can specify font changes for the string in the prefix or suffix rules. If you want to apply paragraph-formatting changes to the string's paragraph, define the changes in a first or last format rule for the parent.

For more information, see:

- "Numbering properties" on page 222 (for autonumber strings)
- "Writing first and last format rules" on page 226
- "Defining prefixes and suffixes" on page 228

Defining a format change list

You can describe a set of changes to paragraph format properties in a *format change list* and then refer to the list from an element definition. Since you need to describe the changes only once, this is helpful when two or more format rules use the same set of changes. A format change list can describe all the same properties that you can write out in a format rule clause.

To define a format change list, insert a `FormatChangeList` element at the same level as `Element` elements. For each group of properties you want to change, insert the element corresponding to the group (such as `PropertiesBasic`) and describe the changes. For a summary of the properties, see "Specifications for individual format properties" on page 215.

You may want to organize all the format change lists together in a separate section in the EDD. For information on using sections, see "Organizing and commenting an EDD" on page 153.

A format change list typically contains a general-purpose set of changes used by different kinds of elements. For example, this list changes the amount of spacing and indentation and can be applied to any kind of display text, such as examples and long quotations:

Format change list: DisplayText
  Basic properties
    Paragraph spacing
      Space above: 12pt
      Space below: 12pt
    Indents
      First indent position relative to left indent: 0pt
      Move left indent by: +12pt

You can refer to the format change list by name in text format rules, in first or last format rules, or in prefix or suffix rules for an element that is formatted as a paragraph or text range. For information on referring to a list, see "Defining the formatting changes in a rule" on page 212.

When you refer to a format change list in an element definition, only the changes that are appropriate for the current element apply. For example, the following list defines changes for code segments that can be formatted as either paragraphs or text ranges. If you refer to the list from a rule clause that formats the code as a text range, the font properties from the list apply but the basic properties do not:

Format change list: Code
- **Basic properties**
  - **Tab stops**
    - **Relative tab stop position:** +12pt
      - **Alignment:** Left
    - **Relative tab stop position:** +24pt
      - **Alignment:** Left
- **Default font properties**
  - **Family:** Courier
  - **Pair kerning:** No

If you apply this change list to a text range element, only the font properties are used.

Only the properties that are used from a change list are passed on to descendants.

When an end user imports element definitions, FrameMaker generates a Format Change List Catalog in the document from named change lists in the EDD. Each time the user re-imports definitions, a new catalog overwrites the existing one in the document. Importing formats does not affect the catalog. If the user removes the structure from a document, the Format Change List Catalog is also removed.

Some properties in format change lists have minimum and maximum limits. Font sizes, for example, must fall within the inclusive range 2 to 400 points. If you set a value that is outside an allowed range (either by specifying an absolute value or by calculating a new relative value), FrameMaker changes the value to the minimum or maximum. With some values, you can also set your own limits. See "Setting minimum and maximum limits on properties," next.

Setting minimum and maximum limits on properties

When defining changes to individual properties in a format rule or in a format change list, you can use relative values for some of the properties in the Basic, Font, and Table Cell groups. A relative value is added to the property's current value to determine a new value. For example, if a `Head` element inherits a paragraph format with a font size of 12 points and you specify a `SizeChange` value of 2 points for the element, the new font size is 14 points.

If a relative value is applied to a property again and again, it is possible for the property's value to become unreasonably large or small. In the example of the `SizeChange` value of 2, the font size for heads becomes 20 points after four applications of the relative value—which may be larger than you have in mind for any section head.

You can set minimum and maximum limits on any properties that can use relative values in text format rules and change lists. A limit you set also applies if you specify an absolute value for the property.

To set minimum and maximum limits on properties, insert a `FormatChangeListLimits` element as the last child element in `ElementCatalog`. For each limit you want to set, insert an element for a set of limits (such as `LeftIndentLimits`) and then insert at least a `Minimum` or `Maximum` child element and type the limiting value. For example:

Limit values for format change list properties
Left indent
Minimum: 1"
Maximum: 3"
First indent
Maximum: 1.5"
Font size
Minimum: 6 pt
Maximum: 20 pt
Line spacing
Minimum: 12 pt

If a relative value in the EDD causes a property's value to be calculated beyond the limit you set, or if you enter an absolute value that is beyond the limit, the value is set to the specified minimum or maximum. For example, if you set a minimum font size of 6 points and an element's font is recalculated to 4 points, the font changes only to 6 points in the element.

Properties that can use relative values have a minimum and maximum limit already built in. You can further limit a built-in range, but you cannot extend it. These are the limits you can define and their built-in minimum and maximum values:

Set of limits	Minimum / maximum
`FirstIndentLimits`, `LeftIndentLimits`, or `RightIndentLimits`	0" / 39"
`LineSpacingLimits`, `SpaceAboveLimits`, or `SpaceBelowLimits`	-32,767" / 32,767"
`TabStopPositionLimits`	0" / 39"
`FontSizeLimits`	2 pts / 400 pts
`SpreadLimits` (Renamed `TrackingLimits`; still works for backward compatibility)	-1000% / 1000%
`TrackingLimits`	-1000% / 1000%
`SpreadLimits`	-1000% / 1000%
`CellMarginLimits`, `BottomCellMarginLimits`, `LeftCellMarginLimits`, `RightCellMarginLimits`, or `TopCellMarginLImits`	0 pt / 32,767 pts

The minimum and maximum limits are global and apply to a value wherever it occurs in the EDD. You cannot set limits for a value in a particular format rule or change list.

For a summary of the properties available in format rules and format change lists, see "Specifications for individual format properties" on page 215.

Debugging text format rules

After writing text format rules, you should try them out by importing the EDD into a sample document with text. If any text is not formatted the way you expect, check the EDD for these errors:

- Typing errors in the tags of paragraph formats, character formats, or format change lists in formatting specifications, or in the element tags or sibling indicators in context specifications
- Typing errors or incorrect child elements in specifications for individual format properties
- Duplicated context specifications
- Format rule clauses that are not in specific-to-general order

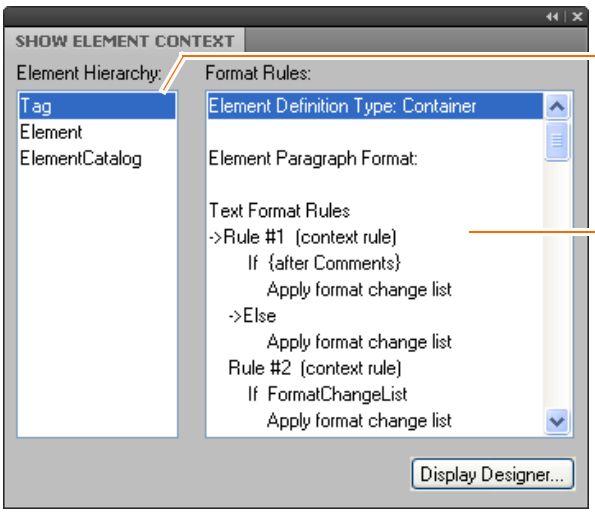
You might also need to look at formats in the document or the template to see if they are defined the way you expect. Check the definitions in these places:

- Paragraph formats in the Paragraph Designer (Format menu)
- Character formats in the Character Designer (Format menu)

If FrameMaker identifies any problems when you import an EDD, it produces a log file of warnings and errors. For information on how to work with this file, see "Log files for imported element definitions" on page 167.

As you examine the formatted contents of the document, you can use the Show Element Context dialog box to find out what format rules are being applied to text in a particular instance of an element. To open the dialog box, choose Show Element Context from the File>Developer Tools submenu.

Show Element Context lists the hierarchy of the current element on the left, beginning with the current element on top. If you select an element in the list, the right side of the dialog box shows that element's format rules. The tag of the element paragraph format in use appears on top, and below it are the format rules that modify the paragraph format. The rule clauses that apply to the particular element instance have an arrow pointing to them.



The scroll box on the right can show text format rules, first and last format rules, and prefix and suffix rules. If the selected element is an object, such as a table or cross-reference, the list shows object format rules instead. For an example of this dialog box with object rules, see "Debugging object format rules" on page 249.

To display the formatting properties for a paragraph format or character format, select the format in the list on the right and click Display Designer.

15 Object Format Rules

A FrameMaker document uses special elements for tables, graphics, markers, cross-references, equations, and system variables. Each of these objects can have a formatting property in a document, such as a table format or an equation size. You can define this property in an object format rule for the element, and the format is applied automatically when an end user inserts an instance of the element in a document.

FrameMaker object format rules have no direct counterparts in markup. If you import a DTD, you can add format rules to the resulting EDD for the objects you plan to use in FrameMaker. If you import a markup document with a CALS `table` element that has a single `tgroup` element, and if the EDD in use has a definition for an element named `Table`, any table format specified in the EDD's `Table` definition is applied to the imported table.

If you export a document or EDD to XML or SGML, the object formatting information is not preserved. However, if you export to XML the software can generate a CSS file that captures the overall formatting of your FrameMaker document. This includes the paragraph formats used in table cells, for example. You specify whether or not to generate the CSS in the `structapps.fm` file. For more information, see Developer Reference, page 24: Managing CSS import/export and XSL transformation.

In this chapter

This chapter explains how to write object format rules for tables, graphics, markers, cross-references, equations, and system variables in FrameMaker. It contains these sections.

–"Debugging object format rules" on page 249

Overview of object format rules

An object element can have one format rule that specifies a formatting property for the element in a document. With the exception of system variables, these properties are not binding—they are initial suggestions for the end user. The rules apply only to new objects in a document and do not affect existing objects.

Unlike text format rules, an object format rule defines the property only for the current instance of an element and is not passed on through a hierarchy to other elements. Although an object can have only one format rule, the rule can have separate clauses that allow for context-specific variations. For example:

Element (Marker): GlossaryTerm
Initial marker type
1. **In all contexts.**
Use marker type: Glossary

Element (CrossReference): CrossRef
Initial cross-reference format
1. **If context is:** * < Table
Use cross-reference format: Page
Else
Use cross-reference format: Heading & Page

These are the formatting properties you can define for FrameMaker objects:

- A table format for a new table element. The format determines properties such as indentation and alignment, margins and shading, and ruling between columns and rows.
- A content type for a new graphic element. The content type specifies that the element is either an anchored frame or an imported graphic object. When an end user inserts the element, the dialog box that appears is either Anchored Frame or Import File.
- A marker type for a new marker element. The marker type specifies the purpose of the marker; some possible marker types are index, glossary, and hypertext.
- A cross-reference format for a new cross-reference element. The format determines the wording and punctuation for a reference, such as *See page 17*.
- An equation size for a new equation element. The equation size (small, medium, or large) controls the font size and horizontal spread of characters.
- A variable for a new system variable element. The variable can be one of the built-in system variables for dates and times, filenames, page and table information, and text for running headers and footers.

The end user can change a marker type, cross-reference format, or other property (except for a system variable) at any time, and the change is not considered to be a format rule override. If the user re-imports element definitions and turns on "Remove Format Rule Overrides," the properties remain as the user has set them. An end user cannot change the variable in a system variable element.

Context specifications for object format rules

An object format rule can apply to all contexts in which the element occurs, or it can define particular contexts or the number of levels deep the element is nested in an ancestor. If the rule defines contexts or levels, it can have separate if, else/if, and else clauses for different possibilities. Each "in all contexts" rule and each if, else/if, and else clause specifies a formatting property.

For example:

Element (Table): Table
General rule: Title, Heading?, Body
Initial table format
1. **If context is:** * < Chapter
   Table format: StandardTable
   Else, if context is: * < Appendix
   Table format: SyntaxTable

A format rule clause has a context specification…
…and a formatting property for that context.

A format rule or clause can have another format rule nested inside it, and can also include a context label to help end users select elements when inserting cross-references or preparing a generated list.

In most respects, the context specifications for object format rules are the same as they are for text format rules. Chapter 17, "Context Specification Rules." describes the context specifications and context labels in all format rules.

Setting a table format

A FrameMaker table uses a table format to determine the basic appearance of the table—such as indentation and alignment, margins and shading in cells, and ruling between columns and rows. Like paragraph formats and character formats, table formats are defined and stored in catalogs in the documents.

You can set an initial table format for new instances of a table element. When an end user inserts the table, the format is preselected in the Insert Table dialog box.

To set a table format, insert an `InitialTableFormat` element after the table element's structure rules (and after any attribute definitions). Then insert and define context elements as necessary, and for each context, insert a `TableFormat` child element and type the tag of a format. The tag must refer to a format stored in the Table Catalog of the documents. For example:

Element (Table): Table
General rule: Heading?, Body, Footing?
Initial table format
1. **If context is:** * < Examples
Table format: TableWithLines
Else, if context is: Item < List
Table format: IndentedTable
Else
Table format: StandardTable

If you do not set a table format, the element uses `Format A`.

Make sure that a table format you use is consistent with the general rule for the table. In the example above, the table formats should not have a table title because the general rule does not allow one.

The initial table format is only a suggestion for the end user. The user can change a table to another format stored in the document (using either Insert Table or the Table Designer), and the change is not regarded as a format rule override. If the user re-imports element definitions with "Remove Format Rule Overrides" on, the table does not return to the format suggested by the EDD.

When importing markup documents, the table format may be specified through a mapping of an attribute to the fm property *table format*. See Developer Reference, page 116: is fm property for more information on this property. If this has been done, then the Initial table format rule in the EDD will be ignored. If no attribute in markup has been mapped to the fm property *table format*, then the Initial table format rule in the EDD will be applied.

Tables can also have text format rules that define formatting properties for text in descendant title and cell elements. For information on these format rules, see Chapter 15, "Text Format Rules for Containers, Tables, and Footnotes."

Specifying a graphic content type

A graphic element can be an anchored frame (which an end user can fill with any graphic object) or an imported graphic file. When a user inserts the element, the Anchored Frame dialog box or the Import File dialog box opens so that he or she can provide information about the object. You can specify a content type for a graphic element, to determine which dialog box to open for new instances of the element.

The Anchored Frame dialog box sets the size, position, and alignment of the frame and other options such as cropping and floating. The Import File dialog box sets the name and location of the graphic file and specifies whether to import by reference or by copying.

To specify a graphic content type, insert an `InitialObjectFormat` element in the graphic element's definition (after any attribute definitions). Then insert and define context elements as necessary; and for each context, insert an `AnchoredFrame` or `ImportedGraphicFile` child element. For example:

Element (Graphic): Figure
Initial graphic element format
1. **If context is:** Item < Procedure
Insert imported graphic file.
Else
Insert anchored frame.

You may want to define separate elements for imported graphics and anchored frames, and let the end user select the one that suits his or her purpose. In this case, use descriptive element tags as a guide for the user:

Element (Graphic): ImportGraphic
Initial graphic element format
1. **In all contexts.**
Insert imported graphic file.

Element (Graphic): AnchFrame
Initial graphic element format
1. **In all contexts.**
Insert anchored frame.

If you do not specify a graphic content type, the element uses the anchored frame type.

The initial content type is only a suggestion for the end user. The user can change an anchored frame to an imported file or vice versa (using either the Anchored Frame or the Import FIle dialog box), and the change is not regarded as a format rule override. If the user re-imports element definitions with "Remove Format Rule Overrides" on, the graphic does not return to the content type suggested by the EDD.

Specifying an object style

You can also choose to specify an object style for your graphic element. The graphic element could either be an anchored frame or an imported graphic file. To specify an object style, insert an `InitialObjectFormat` element in the graphic element's definition and insert the style as required:

Element (Graphic): ImportGraphic
Initial graphic element format
1. **In all contexts.**
InsetStyle.

Element (Graphic): AnchFrame
Initial graphic element format
1. **In all contexts.**
AnchoredFrameStyle.

AnchoredFrameStyle and InsetStyle are names of the object style defined in the template where the EDD is to be imported.

Setting a marker type

FrameMaker markers identify specific locations in a document—for example, to note sources for cross-references, indexes, and other generated lists or to identify active areas for hypertext commands. A marker's type signifies the purpose of the marker, such as cross-reference, index entry, or hypertext location.

You can set an initial marker type for a new instance of a marker element. When an end user inserts the element in a document, the specified marker type is preselected in the Insert Marker dialog box.

To set a marker type, insert an `InitialObjectFormat` element in the marker element's definition (after any attribute definitions). Then insert and define context elements as necessary, and for each context insert a `MarkerType` child element and enter the name of a marker type. For example:

Element (Marker): IndexEntry
Initial marker type
1. **In all contexts.**
Use marker type: Index

These are the marker types available:

Type	Purpose
`Author`, `Equation`, `Glossary`, or `Subject`	Identifies the source of an entry for a generated list or special type of index.
`Comment`	Attaches a nonprinting comment to a location.
`ConditionalText`	Identifies content that is conditional.
`CrossRef`	Marks a source location for a spot cross-reference. (Used only for a cross-reference source that is not an instance of an element or paragraph format.)
`HeaderFooter$1` or `HeaderFooter$2`	Marks a point in the flow where an end user wants the text of a header or footer to change. In a Running H/F system variable, the `$marker1` or `$marker2` building block reads this marker.
`Hypertext`	Attaches a hypertext command to a location.
`Index`	Identifies the source of an entry for a standard index.
`Type 11`	Identifies markup entities and processing instructions.
`Type 12`, … `Type 25`	Identifies the source of an entry for an unspecified type of index or generated list, or identifies a source used by a structure API client.

A marker's text provides the content of the marker, such as the text for an entry in a generated list or the command syntax for a hypertext command. When the end user inserts an instance of the marker element, he or she enters the marker text.

If you do not set a marker type, the element is preset to whatever marker type the end user last selected in Insert Marker or the Marker window.

The initial marker type is only a suggestion for the end user. The user can change a marker to another type (using either Insert Marker or the Marker window), and the change is not regarded as a format rule override. If the user re-imports element definitions with "Remove Format Rule Overrides" on, the marker does not return to the type suggested by the EDD.

Setting a cross-reference format

A cross-reference format determines the wording and punctuation for a cross-reference. For example, a format called `Heading & Page` might display a reference as *See "Defining a prefix or suffix" on page 17*. A format called `Page` might display the same reference simply as *See page 17*. Cross-reference formats are defined and stored in documents.

You can set an initial cross-reference format for a new instance of a cross-reference element. When an end user inserts the element in a document, the specified format is preselected in the Insert Cross-Reference dialog box.

To set a cross-reference format, insert an `InitialObjectFormat` element in the cross-reference element's definition (after any attribute definitions). Then insert and define context elements as necessary, and for each context insert a `CrossReferenceFormat` child element and type the name of a format. The name must refer to a cross-reference format stored in the documents. For example:

Element (CrossReference): CrossRef
Attribute list
1. **Name:** Reference **IDReference** **Required**

Initial cross-reference format
1. **In all contexts.**
Use cross-reference format: Heading & Page

If you do not set a cross-reference format, the element is preset to whatever format the end user last selected in the Cross-Reference dialog box.

The initial cross-reference format is only a suggestion for the end user. The user can change a cross-reference to another format (using the Cross-Reference dialog box), and the change is not regarded as a format rule override. If the user re-imports element definitions with "Remove Format Rule Overrides" on, the cross-reference does not return to the format suggested by the EDD.

When importing markup documents, the cross-reference format may be specified through a mapping of an attribute to the `fm property` *cross-reference format*. See Developer Reference, page 116: is fm property for more information on this property. If this has been done, then the initial cross-reference format rule in the EDD will be ignored. If no attribute in the markup has been mapped to the `fm property` *cross-reference format*, then the Initial cross-reference format rule in the EDD will be applied.

Setting an equation size

FrameMaker provides three sizes for equations: small, medium, and large. The equation size controls the font size of expressions in the equation, the size of the integral and sigma symbols, and the horizontal spread between characters. The properties for the sizes are set in the Equation Sizes dialog box in a document.

You can set an initial equation size for a new instance of an equation element. When an end user inserts the element in a document using the Element Catalog, the equation automatically has the specified size. (If a user inserts the element using the Small, Medium, or Large Equation command in the Equations palette, the command determines the equation size and can override the format rule.)

To set an equation size, insert an `InitialObjectFormat` element in the equation element's definition (after any attribute definitions). Then insert and define context elements as necessary, and for each context insert a `Small`, `Medium`, or `Large` child element. For example:

Element (Equation): DisplayEquation
Initial equation size
1. **In all contexts.**
Use equation size: Large

If you do not set an equation size, the element uses the size `Medium`.

The initial equation size is only a suggestion for the end user. The user can change an equation to another size (using the Object Properties dialog box), and the change is not regarded as a format rule override. If the user re-imports element definitions with "Remove Format Rule Overrides" on, the equation does not return to the size suggested by the EDD.

Setting a MathML equation style

FrameMaker allows you to define the object style for a MathML equation element. You can choose to define the style of the enclosing anchor frame that contains the MathML equation. Or, you can choose to define the style of the MathML equation.

To set the object style for a MathML equation size, insert an `InitialObjectFormat` element. Inside this element, insert an `AllContextsRule` element. Inside the `AllContextsRule` element, insert a `MathMLEquationWithStyle` element.

To define an object style for the enclosing anchor frame, insert the `AnchorFrameStyle` element.

To define an object style for the MathML equation, insert the `MathMLEquationStyle` element.

To specify the object style to be used, enter the name of the style in the above elements. The style name that you provide, is the name that you have defined in the structured application template.

You can choose to define the object styles for both the anchor frame and the MathML equation. However, you need to create one `InitialObjectFormat` element for each.

Specifying a system variable

System variables derive information such as the name of the file or the current date or time from the computer system, and they display this information in a document. FrameMaker provides a variety of predefined system variables. The end user or template designer cannot delete these variables or add new ones in a document, but can edit the definitions of the variables.

You can specify which variable to use in an instance of a system variable element. When an end user inserts the element in a document, the element automatically uses the specified system variable and displays the information from the variable. The format rule for a system variable is binding. The user cannot change a system variable element to another variable.

To specify a system variable, insert a `SystemVariableFormatRule` element in the system variable element's definition (after any attribute definitions). Then insert and define context elements as necessary. For each context, insert a `UseSystemVariable` child element and then a nested child element to specify a variable, or insert a `DefaultSystemVariable` child element to use the `FilenameLong` variable. For example:

Element (System Variable): Date
System variable format rule
1. **If context is:** TitlePageDate
Use system variable: Current Date (Long)
Else
Use system variable: Current Date (Short)

These are the system variables available:

Variable	Default definition	Example
`CreationDateLong`	`<$monthname> <$dayname>, <$year>`	May 15, 2009
`CreationDateShort`	`<$monthnum>/<$daynum>/ <$shortyear>`	5/15/09
`CurrentDateLong`	`<$monthname> <$dayname>, <$year>`	July 6, 2009
`CurrentDateShort`	`<$monthnum>/<$daynum>/ <$shortyear>`	7/6/09
`CurrentPageNum`	`<$curpagenum>`	3
`FilenameLong`	`<$fullfilename>`	/usr/devguide/ objrules
`FilenameShort`	`<$filename>`	objrules
`ModificationDateLong`	`<$monthname> <$dayname>, <$year>, <$hour>:<$minute00> <$ampm>`	August 1, 2009, 2:30 pm
`ModificationDateShort`	`<$monthnum>/<$daynum>/ <$shortyear>`	8/1/09

Variable	Default definition	Example
PageCount	<$lastpagenum>	18
RunningHF1	<$paratext[Title]>	The Turbulent Oceans
RunningHF2	<$paratext[Heading1]>	Threat of Extinction
RunningHF3	<$marker1>	Inspection Checklist
RunningHF4	<$marker2>	Drawing Objects
TableContinuation	(Continued)	(Continued)
TableSheet	(<Sheet <$tblsheetnum> of <$tblsheetcount>)	(Sheet 1 of 2)

Some of the system variables, such as those for running headers and page numbers, are used most often on master pages. If your end users apply structure only to body pages, you will not use these variables in an EDD.

If you do not specify a variable for a system variable element, the element uses the variable FilenameShort.

FrameMaker does not have a variable element for user variables. An end user can insert a user variable as an object in a container element.

Debugging object format rules

After writing object format rules, you should try them out by importing the EDD into a sample document and inserting new instances of the elements. If any objects are not formatted the way you expect, check the EDD for these errors:

- Typing errors in the names of table formats or cross-reference formats in formatting specifications, or in the element tags or sibling indicators in context specifications
- Incorrect child elements in the formatting specifications for graphics formats, marker types, equation sizes, or system variables
- Duplicated context specifications
- Format rule clauses that are not in specific-to-general order

You might also need to look at formats in the document or the template to see if they are defined the way you expect. Check the definitions in these places:

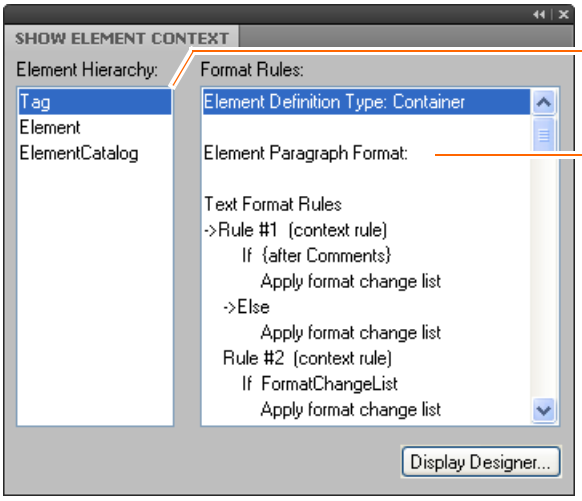
- Table formats in the Table Designer (Table menu)
- Cross-references formats in the Cross-Reference dialog box (Special menu)
- Equation sizes in the Equation Sizes dialog box (Equations pop-up menu in the Equations palette)

• System variables in the Variable dialog box (Special menu)

If FrameMaker identifies any problems when you import an EDD, it produces a log file of warnings and errors. For information on how to work with this file, see "Log files for imported element definitions" on page 167.

As you examine the formatted contents of the document, you can use the Show Element Context dialog box to find out what object format rule is being applied to a particular instance of an element. To open the dialog box, choose Show Element Context from the File>Developer Tools submenu.

Show Element Context lists the hierarchy of the current element on the left, beginning with the current element on top. If you select an element in the list, the right side of the dialog box shows that element's format rules. The rule clause that applies to the particular element instance has an arrow pointing to it.



If the selected element is a container, table title, table cell, or footnote, the scroll box on the right shows text format rules instead of object format rules. For an example of this dialog box with text rules, see "Debugging text format rules" on page 237.

To display the formatting properties for a table format, select the format in the list on the right and click Display Designer.

16 Context Specification Rules

This chapter describes the context specifications and context labels in format rules. For a summary of the formatting changes available, see "Defining the formatting changes in a rule" on page 212.

An element's format rules or paragraph format can also be inherited from an ancestor. For information on this, see "How elements inherit formatting information" on page 206.

In this chapter

This chapter explains how to write EDD context specification rules for all element types. It contains these sections.

All-contexts rules

A format rule can specify a formatting change that applies to an element in all contexts in which it can occur. To write an all-context format rule, insert an `AllContextsRule` element, and then define the formatting changes for the rule.

In this example, an `InLineEquation` element uses the `Small` equation size no matter where the element occurs in a document:

Element (Equation): InLineEquation
Initial equation size
1. **In all contexts.**
Use equation size: Small

Context-specific rules

A format rule can define one or more possible contexts, with formatting changes for each context. The contexts are expressed in separate if, else/if, and else clauses. When applying a format rule to

an element, FrameMaker uses the first clause in the rule that is true for the instance of the element. (Clauses in other rules may also apply.)

To write a context-specific format rule, insert a `ContextRule` element. An `If` element and a nested `Specification` element are inserted automatically along with it. (The `Specification` element does not have a label in the document window.) Type one or more element tags to define the `If` context, and then define the formatting changes for that context. If you need additional clauses in the format rule, you can insert and define any number of `ElseIf` elements, ending with one `Else` element.

Defining a context

When defining a context, you can name the parent element or a list of ancestors. For a list of ancestors, begin with the parent and then name successively higher-level ancestors, separating the element tags with a less-than sign (<).

In this example, an `Item` begins with a bullet if it occurs in a `List` within a `Preface`, or with an incrementing number if it occurs in a `List` within a `Chapter`:

Element (Container): Item
General rule: <TEXT>
Text format rules
1. **If context is:** List < Preface
Numbering properties
Autonumber format: \b\t
Character format: bulletsymbol
Else, if context is: List < Chapter
Numbering properties
Autonumber format: <n+>\t

The ancestors in a list can also be instances of the same element, to describe nesting within that element. For example, this specification is true if an element's parent is a `Section` that is a child element of another `Section`:

Section < Section

Note that a nesting specification of this type is true whenever the current element is nested in *at least* as many levels as shown in the rule. That is, `Section < Section` applies a formatting change if the current element is nested within two or more `Section` elements. (For a way to describe nesting that means *exactly* the level indicated, see "Level rules" on page 257.)

Wildcards for ancestors

Use an asterisk (`*`) as a wildcard to represent an unspecified number of successive ancestors in the hierarchy. For example, this specification is true if an element's parent is a `Section` and any one of the parent's ancestors is also a `Section`:

Section < * < Section

OR indicators

Use OR indicators (|) to test the specification for any ancestor in a group. Separate the element tags of the ancestors with an OR indicator, and enclose the group in parentheses. For example, this specification is true if an element appears in a `List` within a `Preface` or a `Chapter`:

List < (Preface | Chapter)

Sibling indicators

Use a sibling indicator to describe an element's location relative to its siblings. You can use the indicator to describe the relationship of the current element to its siblings or of an ancestor element to its siblings. Enclose the sibling indicator in braces ({ }).

To describe the relationship of the current element to its siblings, type the sibling indicator and a less-than sign, and then continue with the parent and other ancestors. For example, this specification is true if an element is the first element in its parent `NumberList`:

{first} < NumberList

To describe the relationship of an ancestor to its siblings, append the sibling indicator to the ancestor's tag. For example, this specification is true if an element's parent is a `Section` in a `Chapter` and the `Section` immediately follows a `Title`:

Section {after Title} < Chapter

You can also use a sibling indicator by itself as the entire context specification. For example, this specification is true if an element is the only child element in its parent:

{only}

These are the sibling indicators you can use:

Indicator	Specification is true if the element is
`{first}`	The first element in its parent
`{middle}`	Neither the first element nor the last element in its parent
`{last}`	The last element in its parent
`{notfirst}`	Not the first element in its parent
`{notlast}`	Not the last element in its parent
`{only}`	The only element in its parent
`{before` *`sibling`*`}`	Followed by the named element or text content
`{after` *`sibling`*`}`	Preceded by the named element or text content
`{between` *`sibling1`*`,` *`sibling2`*`}`	Between named elements or text content
`{any}`	Anywhere in its parent (equivalent to no indicator)

The *sibling* argument with the before, after, and between indicators can be an element tag or the keyword <TEXT>. If you use <TEXT>, FrameMaker looks to see if the element is preceded or followed by text rather than by a sibling element. A string generated by a prefix, suffix, or autonumber is not considered to be text.

For Containers, Tables, and Footnotes: Rather than defining a first or last relationship between siblings, you may want to write a first or last format rule for the parent. See "Writing first and last format rules" on page 226.

Attribute indicators

You can also use an attribute name/value pair in a text format rule clause to more narrowly define context. For the context specification to be true, an instance of the element must have the attribute name and value specified.

If the element does not have an attribute value but the attribute is defined to have a default value, the default value is used. If the attribute does not have a default value, you can test for an attribute that has no value by specifying an empty string (*attr* ="") for the attribute value.

For object elements: When an end user inserts an object element that has an attribute in its format rule, the Edit Attributes dialog box appears right away so that the user can provide an attribute value—if the user has the element set to display this dialog box automatically.

To test an attribute of the current element, type the attribute name and value in brackets as the context specification. To use an attribute with an ancestor, type the attribute name and value in brackets after the ancestor tag. Separate the attribute name and value with an equal sign, and enclose the value in double quotation marks. If the specification has a sibling indicator, put the attribute information before the indicator.

For example, this rule specifies that an Item begins with a bullet if it occurs in a List that has a Type attribute with the value Bullet, or it begins with an incrementing number if it occurs in a List that has a Type attribute with no specified value:

Text format rules
1. **If context is:** List [Type = "Bullet"]
   Numbering properties
   Autonumber format: \b\t
   Character format: bulletsymbol
   Else, if context is: List [Type = ""]
   Numbering properties
   Autonumber format: <n+>\t

By using attributes in format rules, you may be able to define fewer elements than you would need to otherwise. With the List example above, you can have just one definition for List and rely on attribute values to determine whether an instance of List is bulleted or numbered. Without the attributes, you would need to define separate elements for bulleted and numbered lists.

You can type straight or curved quotation marks around the attribute values (they are automatically curved if you have Smart Quotes on in Text Options). If you need to type a double quotation mark as part of a value, escape the mark with a backslash (\).

To use a set of attribute name/value pairs, separate the pairs with an ampersand (&). For the specification to be true, an instance of the element must have all of the attribute pairs. For example, this specification is true if the element's parent is `List` and the element has a `Type` attribute with the value `Numbered` and a `Content` attribute with the value `Procedure`:

```
List [Type = "Numbered" & Content = "Procedure"]
```

These are the operators you can use in attribute name/value pairs:

Operator	With attributes of
= (equal to)	All types
! = (not equal to)	All types
> (greater than)	Choice and numeric types
< (less than)	Choice and numeric types
>= (greater than or equal to)	Choice and numeric types
<= (less than or equal to)	Choice and numeric types

The definition for a `Choice` attribute includes a list of possible values. If you use a greater-than sign or a less-than sign with a `Choice` attribute in a format rule, FrameMaker evaluates the name/value pair using the order in the list of values, with the "lowest value" being the one on the left. For example, this pair specifies any `Security` value that is to the left of `Classified` in the defined list for the `Security` attribute:

```
Report [Security < "Classified"]
```

With the numeric attributes, you can also express a range of values by combining attribute name/value pairs that use a greater-than sign or a less-than sign. For example, these pairs specify an inclusive range from 12 to 20:

```
Note [Width >= "12" & Width <= "20"]
```

For information on defining attributes, see Chapter 21, "Translating Elements and Their Attributes."

Order of context clauses

When a context-specific format rule has more than one clause, keep in mind that FrameMaker applies the first clause in the rule that is true for the instance of the element. You must write rule clauses from the most specific to the most general.

For example, suppose you want to apply a formatting change to an `Item` when it appears inside a nested `List` element (a `List` inside a `List`). If you put the context specifications for the `Item`

in the following order, FrameMaker would never apply the second clause because an `Item` in a nested `List` also matches the first specification:

List
List < List

You get the effect you want by reversing the clauses.

Level rules

When defining the nesting depth of an element within levels of another element, you may find it easier to use a *level rule* rather than a normal context rule. In a level rule, you name the ancestor to the current element and then in each clause count the number of times the ancestor appears above the current element.

For example, suppose you're describing the nesting depth of a `Head` in `Section` elements. This is how you would define it using a normal context rule:

Element (Container): Head
General rule: <TEXT>
Text format rules
1. **If context is:** Section < Section < Section
   Default font properties
   Font size: 12
   Else, if context is: Section < Section
   Default font properties
   Font size: 14
   Else, if context is: Section
   Default font properties
   Font size: 18

This is the same specification using a level rule instead:

Text format rules
1. **Count ancestors named:** Section
   If level is: 1
   Default font properties
   Font size: 18
   If level is: 2
   Default font properties
   Font size: 14
   If level is: 3
   Default font properties
   Font size: 12

Note that in context rules you need to go from the lowest level to the highest, but in level rules the order of clauses is arbitrary so you can go from highest to lowest if you prefer. In context rules `Section < Section < Section` means "nested in *at least* three `Section` elements," and

in level rules a Section count of 3 means "nested in *exactly* three Section elements." Because the level counts are exact, you can specify them in any order.

To write a level rule, insert a LevelRule element. In the CountAncestors element that is inserted automatically, type the element tag of the ancestor to count. Insert an If element (a nested Specification element is inserted automatically along with it), and type the level number of the ancestor. Then define the formatting changes for that level.

If you need additional clauses in the format rule, you can insert and define any number of ElseIf elements, ending with one Else element. Each specification needs a level number and formatting changes for that level. An Else clause includes a count of 0 unless there is an If or ElseIf clause for that case.

Level rules are usually simpler than context rules (especially with three or more levels), because you do not need to spell out the context in each clause.

You can name a list of ancestors in a level rule, and FrameMaker will count any ancestor in the list. Separate the ancestors with a comma. For example, this specification is true if an element is nested within the specified number of levels of any combination of Section and Chapter elements:

Section, Chapter

You cannot mix context clauses and level clauses together in a single format rule, though you can nest a context rule in a level clause or a level rule in a context clause, and one element definition can have both kinds of rules.

Using the current element in the count

A level rule can also count instances of the current element in the hierarchy. You do not need to provide the element's tag as ancestor information. After inserting the LevelRule element, delete the CountAncestors element that is inserted automatically and begin with the If specification. The current instance of the element is included in the count.

For example, this rule specifies that a Section is indented 0.5 inch if it is nested within another Section:

Element (Container): Section
General rule: Head, Para+
Text format rules
1. If level is: 2
Basic properties
Indents
Move left indent by: +0.5"

The rule in this example is also applied to any Head and Para descendants of the Section.

Stopping the count at an ancestor

You can have FrameMaker stop counting when it reaches a particular element in the hierarchy. This allows you to use different formatting changes for a nested element in different contexts. Insert the `StopCountingAt` element after the `CountAncestors` element, and type the tag of the element to stop counting at. Then continue with the `If`, `ElseIf`, and `Else` specifications.

For example, suppose you want list items in a table cell to be formatted based on the nesting of lists in the cell. If the entire table can occur in a list in the document, you want to test for the nesting level of lists only in the table, and not in the overall nesting of lists in the document. You do this by counting lists in successive ancestors until you reach a table. In the following rules, an `Item` in a second-level `List` can be indented 0.5 inch or 1 inch. The `Item` is indented 0.5 inch if the nested `List` is in a `Table`:

Element (Container): Item
General rule: <TEXT>
Text format rules
1. **Count ancestors named:** List
   Stop counting at first ancestor named: Table
   If level is: 2
   Basic properties
   Move left indent by: +0.5"
2. **Count ancestors named:** List
   Stop counting at first ancestor named: Chapter
   If level is: 2
   Basic properties
   Move left indent by: +1.0"

For the first rule to be true, the hierarchy between the current `Item` and the closest `Table` ancestor must include two `List` elements. For the second rule to be true, the hierarchy between the `Item` and the `Chapter` ancestor must include two `List` elements.

Nested format rules

Within any format rule clause, you can nest another entire format rule. By nesting rules, you can sometimes organize rules more efficiently and make them easier to read. Look for places in a format rule where context information is repeated, and see if you can combine all the formatting information for that context in a nested rule.

To add a nested format rule, insert a `Subrule` element after the context specification for the outer rule. Specify the nested rule in the same way that you write a main rule.

For example, suppose an `Item` element can occur in either a numbered list or a bulleted list, and in a numbered list the autonumber format is different for the first position in the list than for other positions. You might write the format rule in this way:

Element (Container): Item
General rule: <TEXT>
Text format rules
1. **If context is:** {first} < List[Type="Numbered"]
Numbering properties
Autonumber format: <n=1>\t
Else, if context is: List[Type="Numbered"]
Numbering properties
Autonumber format: <n+>\t
Else
Numbering properties
Autonumber format: \b\t

These context specifications can be combined.

Rather than having two rule clauses for the numbered-list contexts, you can have one main clause with a nested rule (using a `SubRule` element) that describes the two variations:

Text format rules
1. **If context is:** List[Type="Numbered"]
1.1 If context is: {first}
Numbering properties
Autonumber format: <n=1>\t
Else
Numbering properties
Autonumber format: <n+>\t
Else
Numbering properties
Autonumber: \b\t

The outer rule determines whether the context is a numbered list.

If it is a numbered list, the inner rule determines which autonumber format to use.

You can nest a level rule in a context clause, or a context rule in a level clause.

Multiple format rules

An element definition can have more than one format rule. Like nesting rules, writing separate format rules can help you organize information efficiently. Look for places in a format rule where formatting changes are repeated, and see if you can break out the changes into a separate rule.

To add another format rule for an element, insert an `AllContextsRule`, `ContextRule`, or `LevelRule` element after an existing rule, and specify the context and formatting changes. Write the format rules in the order you want them to be interpreted.

For example, suppose a `Head` element has the same font change in any context but a different autonumber format in different nesting levels. This shows the formatting changes described in a single rule:

Element (Container): Head
General rule: <TEXT>
Text format rules
1. **If context is:** Section < Section
Default font properties
Weight: Bold
Numbering properties
Autonumber format: <n>.<n+>\t
Else:
Default font properties
Weight: Bold
Numbering properties
Autonumber format: <n+>\t

These font properties can be combined.

Rather than repeating the `Bold` specification, you can break out that part into a separate rule for all contexts:

Text format rules
1. **In all contexts.**
Default font properties
Weight: Bold
2. **If context is:** Section < Section
Numbering properties
Autonumber: <n>.<n+>\t
Else:
Numbering properties
Autonumber: <n+>\t

The first rule sets the weight to Bold for every Head element.

Keep in mind that FrameMaker applies format rules in the order they appear in the definition, so it is possible for a format rule to override an earlier rule. For example, if rule 1 changes the weight of text in all contexts and rule 2 applies a different paragraph format in certain contexts, the paragraph format overrides the change in the font weight.

Context labels

In some dialog boxes, FrameMaker displays a list of element tags for an end user to select from. For example, a user selects element tags in the Set Up dialog box (Generate command) to set up a generated file such as an index or a table of contents.

You may want FrameMaker to distinguish among instances of some elements in these lists. For example, an end user might include `Head` elements in a table of contents when the parent of the `Head` is a first- or second-level `Section`, but not when the parent is a more deeply nested `Section`. To allow FrameMaker to distinguish element instances, you provide a context label with formatting variations of the element.

In a dialog box, the end user sees an element once for each context label it can have and once for all contexts in which no label applies. The user selects an element with a label to work with all instances of the element from the context associated with the label.

To provide a context label in a format rule clause, insert a `ContextLabel` element after the `If`, `ElseIf`, or `Else` specification, and type the text of the label. A context label cannot contain white-space characters or any of these special characters:

```
( ) & | , * + ? < > % [ ] = ! ; : { } "
```

For example, this rule has a context label for different levels of `Head` elements:

Element (Container): Head
General rule: <TEXT>
Text format rules
1. **Count ancestors named:** Section
If level is: 1
Context label: 1st-level
Default font properties
Font size: 18
Else, if level is: 2
Context label: 2nd-level
Default font properties
Font size: 14

In a dialog box that lists element tags, the end user sees the `Head` element like this:

Head(<no label>)
Head(1st-level)
Head(2nd-level)

You can use the same label in more than one clause or format rule. Context labels are not inherited by descendants.

If multiple labels apply to an instance of an element (because of multiple rules with labels that apply), only the last appropriate label in the definition is stored with the instance. Thus, only the last label determines whether the instance appears in a generated list.

Part IV Translating Between Markup Data and FrameMaker

Part III provides details of the model FrameMaker uses for translating between markup data and FrameMaker documents and on how you can modify this translation. You should be familiar with the material in Part II, "Developing a FrameMaker Structure Application" before using the material in this part.

The chapters in this part are:

- Chapter 17, "Introduction to Translating between Markup Data and FrameMaker" describes the types of modifications you can accomplish with read/write rules, with a simple example of using rules to create a structure application.
- Chapter 18, "Read/Write Rules and Their Syntax" describes the mechanics of writing read/write rules and the document in which you define them.
- Chapter 19, "Saving EDD Formatting Information as a CSS Stylesheet" describes how formatting information can be mapped to a CSS style sheet when you save a document as XML.
- The following chapters describe specific types of translation between markup elements and FrameMaker constructs. Each chapter first describes the default translation, and then the modification you can make with read/write rules:
  - –Chapter 20, "Translating Elements and Their Attributes"
  - –Chapter 21, "Translating Entities and Processing Instructions"
  - –Chapter 22, "Translating Tables"
  - –Chapter 23, "Translating Graphics and Equations"
  - –Chapter 24, "Translating Cross-References"
  - –Chapter 25, "Translating Variables and System Variable Elements"
  - –Chapter 26, "Translating Markers"
  - –Chapter 28, "Translating Conditional Text."

- Chapter 28, "Processing Multiple Files as Books" describes how markup languages handle collections of files, as compared to the FrameMaker book concept, It discusses how multiple markup files translate to book and document files on import, and how book and document files translate to multiple markup files on export.
- Chapter 4, "Conversion Tables for Adding Structure to Documents"Describes how to use the conversion table utility for adding structure to unstructured files.
- Chapter 29, "Additional XSL Transformation for XML" describes how you can specify additional transformations for XML using XML Style Language (XSL), which are applied before read rules on import, and after write rules on export.

17 Introduction to Translating between Markup Data and FrameMaker

FrameMaker can read and write markup data without any help from you. However, its default translation of markup constructs may not be suitable for your DTD. In such cases, you have to write a structure application to modify the translation.

In this chapter

This chapter talks about the information your application typically contains. It contains these sections.

What you can do with read/write rules

Read/write rules are needed for a variety of reasons. Most rules perform tasks falling in one of these categories:

Working with special constructs Because markup doesn't standardize a model for constructs such as tables, graphics, or cross-references, their handling is unique to each DTD. When creating a DTD from an EDD, FrameMaker makes assumptions about how to translate these FrameMaker constructs to markup elements and attributes. Your application may need only to modify the default translation in minor ways using rules. On the other hand, when creating an EDD from a DTD, FrameMaker cannot recognize these constructs, so your rules need to be more extensive.

Renaming elements and attributes FrameMaker element tags and attribute names are frequently more descriptive than their XML or SGML counterparts and you might use a read/write rule to establish a correspondence between names. For example, you could use a rule to change the FrameMaker tag `Employee Name` to the XML generic identifier `ename`.

Treating special characters In SGML and XML you can use an entity to represent special characters. You might use rules to translate such entities.

> **Note: XML:** The XML specification establishes *predefined* character entities for reserved characters such as angle brackets. By default FrameMaker translates these characters correctly on import and export. However, you will need to specify read/write rules to export such characters as a numeric references (`&#XXXX;`).

Making information implicit Not all information relevant to one representation is relevant to the other. For example, you may have a type of table that always has two columns. In XML you have an element for this table and do not need to explicitly indicate in the document instance that the table has two columns. Nevertheless, in FrameMaker you need to specify a number of columns when you insert the table. By default, the software writes an attribute containing the number of columns on export; you can choose instead to specify this information in a read/write rule.

Unwrapping structure You may have levels of hierarchical element structure in one representation that are unnecessary in the other. If you are translating documents from one system to the other and don't need to translate those documents back again, you may decide to simplify the element hierarchy.

Other rules There are a few rules that aren't used for any of the above purposes. These rules are discussed in detail in Developer Reference, Chapter 3, Read/Write Rules Reference Sections of the chapter that describe examples of such rules include:

- Developer Reference, page 49: character map describes the rule that tells FrameMaker how to map between characters in the SGML and FrameMaker character sets.
- Developer Reference, page 72: external dtd describes the rule that determines whether to include an external DTD subset or to copy the specified DTD into the internal DTD subset. By default, FrameMaker includes an external DTD subset.
- Developer Reference, page 141: line break describes the rule that tells FrameMaker how to interpret line breaks when importing an XML or SGML document and when to generate line breaks when exporting a FrameMaker document.
- Developer Reference, page 165: write structured document and Developer Reference, page 165: write structured document instance only describe the rules that tell FrameMaker when saving as markup whether it should write only a markup document instance or an entire document.

What you can do with structure API clients

In situations in which read/write rules are insufficient to express the proper translation between markup and FrameMaker, you can create a structure API client. You use the Frame Developer's Kit (FDK) to modify the import or export of markup documents. You cannot use the FDK to directly modify how the parser creates a DTD or an EDD.

Working with the FDK requires C and FDK programming skills. The FDK allows you an arbitrary amount of control over FrameMaker's processing. For this reason, there is no way to briefly describe the possibilities here.

During import, your FDK functions can inspect or modify the structure, format, or content of the FrameMaker document being constructed. FDK functions can process attribute values, processing instructions, and some entity references encountered in the imported markup document.

During export, your functions can inspect the structure, format, and content of the exported FrameMaker document. They can inspect and modify every generated portion of the markup document before it is actually written, and they can create new text (data and markup) to be inserted into the XML or SGML document.

This manual does not describe how to create a structure API client, although it does occasionally mention situations in which a client is appropriate. For information on creating structure API clients, see the *Structure Import/Export API Programmer's Guide*.

A detailed example

To give you a more concrete feeling for translating between markup and FrameMaker, this section contains an example of a DTD fragment and the corresponding EDD fragment, rules for translating between them, and snippets of related SGML and FrameMaker documents.

This section does not explain how to create the statements shown here. Later chapters provide the details.

DTD fragment

Here is a fragment of the DTD:

```
<!--Shorthand for name of the system being described-->
<!ENTITY plan CDATA "Transportation 2000">

<!--Entities for special characters.-->
<!ENTITY oquote SDATA "open-quote">
<!ENTITY cquote SDATA "close-quote">

<!--Parameter entity for text that can appear in a paragraph.-->
<!ENTITY % text "(#PCDATA | xref)*">

<!--Basic element structure-->
<!ELEMENT section - - (head, (para | list)*, section*)>
<!ELEMENT (head, para) - O (%text;)>
<!ELEMENT list - - (item+)>
<!ATTLIST list type (bullet | number) bullet>
<!ELEMENT item - O (%text;, list?)>
```

```
<!--Allow * as the start-tag of an item element-->
<!ENTITY strtitem STARTTAG "item">
<!SHORTREF listmap "*" strtitem>
<!USEMAP listmap list>

<!--Cross-references-->
<!ELEMENT xref - O EMPTY>
<!ATTLIST xref id IDREF #IMPLIED>
```

This DTD fragment defines several entities. The `plan` entity is shorthand for the string "Transportation 2000"; this makes it easy for an end user to change the plan name in one place. The `oquote` and `cquote` entities are system-specific characters—the directional open and close quotation marks. Finally, `%text` is a parameter entity that simplifies entering a frequently used portion of the content model in the DTD.

Next are several fairly straightforward element definitions for the `section`, `head`, `para`, `list`, and `item` elements. The `list` element has an attribute presumably directing a structure application on the appropriate formatting of an instance of the element. There are also several declarations that allow users to use an asterisk (*) as markup to indicate the start of an `item` element in a `list` element. The `xref` element and its attributes describe cross-references.

Document instance

A portion of a document instance written using this DTD looks like this:

```
<section>Summary of &plan; Plan Elements
<head>Highway System
<para>
A base network of roads for people and goods movement designed to
operate at maximum efficiency during off-peak and near capacity in
peak hours. Elements include freeways, expressways, and major
arterials.
<list type=number>
*Completion of Measure &oquote;A&cquote; program for Routes 101, 85,
237
*Emphasis on Commuter Lanes and bottleneck improvements including
new and upgraded interchanges
*Capacity improvements in 101 and Fremont/South Bay Corridors
*Operational improvements including signal synchronization and
freeway surveillance
</list>
</list>
```

EDD fragment

Here is an EDD fragment you might produce from the DTD fragment:

Element (Container): Section
  General rule: Head, (Para | List)*, Section*
  Text format rules
    Element paragraph format: body

Element (Container): Head
  General rule: (<TEXT> | Xref)*
  Text format rules
   1. Count ancestors named: Section
    If level is: 1
      Use paragraph format: head1
    Else:
      Use paragraph format: head2

Element (Container): Para
  General rule: (<TEXT> | Xref)*

Element (Container): List
  General rule: Item+
  Attribute list
    1. Name: Type **Choice** **Optional**
      Choices: Bullet, Number
      Default: Bullet

Element (Container): Item
  General rule: (<TEXT> | Xref)*, List?
  Text format rules
    1. If context is: {first} < List [Type = “Number”]
        Use paragraph format: 1Step
    Else, if context is: List [Type = “Number”]
      Use paragraph format: Step
    Else, if context is: List < Item
      Use paragraph format: 2Bullet
    Else:
      Use paragraph format: Bullet

Element (Cross-reference): Xref
  Initial cross-reference format
    1. In all contexts.
      Use cross-reference format: NumOnly

Each element definition specifies formatting appropriate for that element and its descendants. The `Section` element specifies body as the paragraph format for all text unless a descendant specifies otherwise. The `Head` element specifies Head1 as the paragraph format if it occurs as the

child of a single `Section` element and Head2 if it occurs as the child of more than one `Section` element. The `Item` element bases it's formatting on the value of the `Type` attribute of the parent `List` element.

Formatting and read/write rules

When creating an EDD from a DTD, FrameMaker automatically generates most of these FrameMaker element definitions (other than their formatting rules) from the corresponding markup declarations without any intervention. You only need to specify format rules, and one read/write rule that states that the markup element `xref` should become the special cross-reference element type.

There is more interesting information in the DTD than just the element structure, however. By default, FrameMaker creates a variable with the variable text "Transportation 2000" corresponding to the `plan` entity. This translation is probably what you want. It also creates variables for the `oquote` and `cquote` entities, with the variable text "open-quote" and "close-quote", respectively. This is probably not what you want for those entities. You can supply rules to have the `oquote` and `cquote` entities become the directional open and close quotation characters.

The `plan` variable and the information about the `oquote` and `cquote` entities aren't part of the EDD. The variable definition is stored directly in a FrameMaker document that uses that entity; the information about the `oquote` and `cquote` entities remains solely in the rules, although a FrameMaker document created from a markup document that uses those entities contains open and close quotation characters, as appropriate.

So, the complete set of read/write rules you'll need is as follows:

```
/* Change the xref element to a special element type. */
element "xref" is fm cross-reference element;

/* Translate SDATA entities directly into the FM character set. */
entity "oquote" is fm char "\“";
entity "cquote" is fm char "\”";
```

FrameMaker document

With these rules and an appropriate template, FrameMaker translates the earlier SGML markup to the following in a FrameMaker document window:

1.2 Summary of Transportation 2000 Plan Elements

1.2.1 Highway System

A base network of roads for people and goods movement, designed to operate at maximum efficiency during off-peak and near capacity in peak hours. Elements include freeways, expressways, and major arterials.

1. **Completion of Measure "A" program for Routes 101, 85, 237**
2. **Emphasis on Commuter Lanes and bottleneck improvements including new and upgraded interchanges**
3. **Capacity improvements in 101 and Fremont/South Bay Corridors**
4. **Operational improvements including signal synchronization and freeway surveillance**

Given the above structured FrameMaker document, on export FrameMaker produces SGML markup equivalent to that shown earlier. Without a structure API client, it doesn't produce short references or markup minimization.

Opening XML documents

When you open an SGML document, FrameMaker creates a structured FrameMaker document. It proposes a name for this new file by replacing any extension in the name of the SGML document with `.fm` (and appending `.fm` to the end of the SGML document's filename if that name has no extension). It uses the proposed name to label the document window, but allows you to change the name before saving the new document. When you save the document or use the File > Save As command, by default, FrameMaker saves the document as a FrameMaker document.

By default, when you open an XML document, however, FrameMaker retains the original name. When you save the imported document, FrameMaker saves the file back to XML, under the original filename. This default behavior resembles corresponding behavior for working with FrameMaker documents. It differs in the following ways:

- If you have not yet saved the file, you can use the File > Revert to Saved command to discard changes that have been made to a FrameMaker document. This command cannot be used with XML documents.
- You can use File > Preferences > General to set an automatic save option. When this option is set, FrameMaker automatically saves edited documents at specified time intervals. FrameMaker does automatically save XML documents, but it saves them as FrameMaker documents. If you need to open the autosave file for an XML document, perhaps after a system crash or power failure, you can save it as XML.

- If you have enabled Network File Locking through **File > Preferences > General**, FrameMaker locks both FrameMaker and XML documents that you are editing to prevent other users from modifying files you have open. However, it uses different mechanisms for locking FrameMaker and XML documents.

You can save an opened XML document as a FrameMaker document. You can work with the resulting FrameMaker document as you would any other FrameMaker document. When you have finished editing, you can save the result as XML.

Two configuration settings allow you to modify the default behavior. Since these settings are made in `maker.ini`, they apply to all your structure applications. To change them, you must exit from FrameMaker, change the setting, and then restart FrameMaker. Both settings are made in the `Preferences` section of the configuration file:

- `TreatXMLAsXML` can be set to either `On` or `Off`; the default is `On`. When it is `On`, FrameMaker retains the name of an imported XML document as described above. When it is `Off`, FrameMaker uses the extension `.fm` for imported XML documents and by default saves them as FrameMaker documents. In this case, the file naming and file typing for imported XML parallels that for SGML.
- `LockXmlFiles` can also be set to either `On` or `Off` with a default of `On`. If the setting is `On`, FrameMaker locks XML documents that are opened when `TreatXMLAsXml` is `On`. If `LockXmlFiles` is `Off`, FrameMaker does not lock open `XML` documents.

18 Read/Write Rules and Their Syntax

Read/write rules are your primary device for modifying FrameMaker's default translations. The Developer Reference, Chapter 3, Read/Write Rules Reference describes each of the rules in detail.

In this chapter

This chapter describes the rules and how you create them. It contains these sections.

The rules document

You create a FrameMaker document or an ASCII file containing read/write rules. The software uses this rules document to modify its import and export processing of markup data.

Assuming that you create a FrameMaker document for your rules, you enter a set of rules into the main flow on the body pages of the document. The document can be structured or unstructured and can use any element definitions or formatting properties desired. FrameMaker reads the

content of the document and not its structure. Keywords in rules cannot include non-breaking hyphens.

> **Important:** Eight-bit characters, greater than 0x80, are not supported in filenames occurring within read/write rule files. For portability, avoid using those characters in filenames within read/write rule files.

To create a new read/write rules document, choose File>Developer Tools>New Read/Write Rules. This command creates a new rules document, using the template it finds in `$STRUCTDIR/default.rw`. For information on the location of `$STRUCTDIR`, see "Location of structure files" on page 130.

The first rule in a rules document must be:

```
fmsgml version is "<version>";
```

The string only includes one level of "dot" releases. For example, you would use the string `"7.0"` even though the product version may be an incremental release above 7.0, such as 7.1 or 7.2.

Most rules are relevant all the time—for example, a rule to convert the general entity `pname` to the FrameMaker variable `Product Name`. Some, however, are only relevant to certain situations. For example, a rule to export graphics into external entities named `graphic1.mif`, `graphic2.mif`, and so forth is relevant only when exporting FrameMaker documents. It is not relevant when exporting an EDD or when importing markup documents or DTDs, because these situations do not require generating entity names for graphics.

There is so much overlap in the rules that FrameMaker adopts the strategy of having one document specify all rules, rather than using a separate document for each type of operation: import of a DTD, import of an XML or SGML document, export of an EDD, or export of a FrameMaker document. Most rules are expressed from the markup perspective; the *element* of an `element` rule is the XML or SGML generic identifier, not the FrameMaker element tag. The only exceptions to this convention are rules for FrameMaker constructs that have no counterpart in markup. For example, a rule might specify dropping FrameMaker markers on export.

A single rules document can also contain rules that do not apply to a particular document. This allows the same rules document to be used with several related document types, even if rules exist for constructs used in only some of them. Markup and FrameMaker provide the same flexibility, by allowing a DTD or EDD to define constructs that are never used. Because unused rules can also result from spelling errors in generic identifiers and in the names of other constructs, FrameMaker issues warning messages in the log file when rules refer to constructs that do not exist.

Rule order

Although the first rule in a read/write rules file must specify the product version, in general the order in which other rules appear is not significant. The only time rule order is significant is when

multiple rules apply equally to the same situation. In such cases, the software uses the rule that appears *first* in the document. For example, assume you have these rules:

```
element "list" is fm element "List";
element "list" is fm element "Procedure";
element "proc" is fm element "Procedure";
```

These rules say that two different FrameMaker elements correspond to the same markup element, and two markup elements correspond to the same FrameMaker element. FOr example, if you import an XML document to FrameMaker with these rules, all occurrences of the XML `list` element become `List` elements, and occurrences of the `proc` element become `Procedure` elements. If you export a FrameMaker document to XML, both `List` and `Procedure` elements become `list` elements. And no `proc` elements are created. The result is that if you start with an XML document containing a `proc` element, import that document to FrameMaker, and then export the result back to XML, the original `proc` element becomes a `list` element.

On the other hand, the order of the following rules does not matter:

```
fm attribute "XRefLabel" drop;

element "chapter"
   attribute "xreflab" is fm attribute "XRefLabel";
```

The first of these rules tells the software to discard the FrameMaker `XRefLabel` attribute in any element in which it occurs. The second rule tells the software to translate the FrameMaker `XRefLabel` attribute to the XML `xreflab` attribute within the context of the `Chapter` element. The effect of these two rules is to discard the `XRefLabel` attribute when it occurs in any element other than the `Chapter` element.

Rule syntax

Rules use a syntax similar to C language syntax, in which each rule begins with a keyword and ends either with a brace-enclosed group of subrules or with a semicolon. Names of markup and FrameMaker constructs, such as elements, attributes, and entities, are represented by strings. For information on strings, see "Strings and constants" on page 277. Rules can be *nested;* that is, they can occur inside one another. A rule nested inside another is called a *subrule*. One that is not nested is called a *highest-level rule*. A typical rule containing a brace-enclosed set of two subrules is:

```
element "vex" {
   is fm element "Verbatim Example";
   attribute "au" is fm attribute "author";
}
```

In this example, `element` is the highest-level rule. The `is fm element` and `attribute` rules are subrules of the `element` rule. The `is fm attribute` rule is a subrule of both the `element` and `attribute` rules.

A single subrule can appear at the end of a rule without the enclosing braces. For example:

```
element "list" {
   attribute "indent" drop;
}
```

is equivalent to

```
element "list" attribute "indent" drop;
```

Null rules, consisting simply of a semicolon, can appear wherever a rule is allowed.

Case conventions

For keywords of the read/write rules syntax, case is not significant. For example, the following are equivalent for rules in a structure application:

```
fm version is "8.0";
FM Version is "8.0";
```

To improve readability, this manual sometimes uses mixed case for keywords.

The case of FrameMaker element tags and other FrameMaker names is always significant. Thus, the following rules are not equivalent:

```
fm element "Default Element" drop;
fm element "default element" drop;
```

The significance of the case of SGML names is dependent on the `NAMECASE` parameter of the SGML declaration. For XML the case is always significant, and your rules must match the case used in the XML markup. For more information, see "Naming elements and attributes" on page 296.

Strings and constants

You use strings to specify FrameMaker element tags, generic identifiers in markup, attribute names, attribute values, entity names, notation names, and so on.

String syntax

Strings are enclosed in matching straight or directional double quotation marks. Strings cannot extend across more than one paragraph.

To incorporate special characters into strings, use the following conventions:

- `\x` followed by one or two hexadecimal digits is interpreted as a hexadecimal character code.
- `\0` (backslash zero) followed by one or two octal digits is interpreted as an octal character code.
- A backslash followed by one, two, or three other digits is interpreted as a decimal character code.

- A backslash immediately preceding any character other than a digit or the letter *x* indicates that the following character is part of the string. In particular, a double quotation mark or backslash can be preceded by a backslash to enter it into a string.

Constant syntax

Some rules and parameter literals for entities accept a single constant character code instead of a one-character string with a character code. In these cases, the constant character code takes a slightly different format than it does within a string. In particular:

- A number starting with `0` (zero) followed by other digits is interpreted as an octal number.
- A number starting with `0x` (zero x) followed by other digits is interpreted as a hexadecimal number.
- Any other number is interpreted as a decimal number.

Variables in strings

FrameMaker defines a small set of variables that can be used within strings in particular rules. These variables have the following syntax:

`$(`*`variable_name`*`)`

where *`variable_name`* must be a legal variable name. An example is the variable `$(Entity)`.

The following table illustrates the conventions for entering strings. The strings occur within a rule in which the variable `$(Entity)` is interpreted as `CHIPS`:

Syntax within rule	Interpreted string
`"Chips"`	Chips
`"Potato Chips"`	Potato Chips
`"\""`	"
`"\\"`	\
`"a$(entity)z"`	aCHIPSz
`"a\$(entity)z"`	a$(entity)z
`"a$entityz"`	a$entityz
`"\xa7Chips"`	ßChips

For more information on entities, see Chapter 22, "Translating Entities and Processing Instructions."

Comments

Comments can appear anywhere in a rules document where white space is permitted, except within quoted strings. Comments, like those in C code, are surrounded by the delimiters `/*` and

*/. Tables, graphics, and equations can appear within comments but are erroneous elsewhere in a rules document.

Include files

You can use the C notation for include files in a rules document. For example, assume you have the following line in a paragraph by itself in a rules document:

```
#include "fname"
```

FrameMaker processes the file named *fname* as though its contents were inserted in place of the include directive. The syntax of the filename is device-dependent.

You can specify a search path for include files in the structapps.fm file. The default search path for an include file consists of the directory containing the original rules document and the directory $STRUCTDIR/isoents/, where $STRUCTDIR is as defined in "Location of structure files" on page 130. In addition, these directories are added to the end of the search path you specify in structapps.fm.

If you plan to use the same rules on different systems, avoid specifying directory names in the #include directive. Instead, let FrameMaker find files in different directories through a search path.

> **Important:** File pathname tokens must be separated by a backslash character ("\") and you must escape the backslash character as follows:
>
> $STRUCTDIR\\isoents

For information on working with the structapps.fm file, see "Application definition file" on page 132.

Reserved element names

You may use any legal name for an XML or SGML element. If there are no rules to the contrary, FrameMaker assumes most markup elements correspond to FrameMaker elements. The following generic identifiers are used by the default declarations for translating FrameMaker cross-references, equations, footnotes, graphics, or tables to markup: CROSSREF, EQUATION, FOOTNOTE, GRAPHIC, TABLE, TITLE, HEADING, BODY, FOOTING, ROW, and CELL.

SGML: FrameMaker uses the following entity names for translating FrameMaker system variables to SGML: fm.pgcnt, fm.ldate, fm.sdate, fm.lcdat, fm.scdat, fm.lmdat, fm.smdat, fm.lfnam, fm.sfnam, fm.tcont, and fm.tsht.

If your DTD uses any of these names for another purpose, you will need to write a rule to provide alternate names for the constructs provided by the default declarations.

XML: In the absence of read/write rules, FrameMaker exports system variables as text. To preserve data as system variables for import and export, declare an empty element in your DTD

for each system variable your document uses, and specify rules to convert the element to a system variable.

Commands for working with a rules document

The **StructureTools** menu provides two commands for working with a read/write rules document: **New Read/Write Rules** and **Check Read/Write Rules**.

- **New Read/Write Rules**

  To create a new rules document, choose New Read/Write Rules. This command uses the rules template in `$STRUCTDIR/default.rw`, if there is one. For information on the `$STRUCTDIR` directory, see "Location of structure files" on page 130.

- **Check Read/Write Rules**

  To verify the correctness of a rules document, choose **Check Read/Write Rules**. This command works with the current document. That document must be a read/write rules file that has been saved to a file. When you choose the **Check Read/Write Rules** command, you get a dialog box asking you to pick the application to check the rules against. You can choose an application for this command, or check the rules without the added context of an application. If the application you choose specifies a different read/write rules document, that rules document is ignored for the purposes of this command.

  The **Check Read/Write Rules** command checks for syntax errors in rules and also reports semantic errors on the basis of the DTD and the FrameMaker template specified in the selected application, if there is one. Errors are reported in a log file. Note that your read/write rules document can refer to markup constructs not mentioned in the current DTD—for details, see "The rules document" on page 274.

For information on adding a DTD file or rules document to an application, see Developer Reference, page 9: Define an application.

19 Saving EDD Formatting Information as a CSS Stylesheet

Typically, an EDD contains a significant amount of formatting information. FrameMaker uses this information to map formatting properties to elements. This mapping depends on the given element's tag, its position in the document structure, and its values for specified attributes.

In XML this type of formatting information is typically expressed in a cascading stylesheet (CSS). However, if an XML document specifies a CSS FrameMaker does not use the CSS when it opens the XML. Instead, FrameMaker uses the formatting that is specified in the template—definitions in the format catalogs, plus the formatting rules specified in the template's EDD.

When you save a FrameMaker document as XML, you may want to generate a CSS for the XML or you may want to use a CSS that was provided with the original XML document. To make that decision, you should understand how to specify the actions FrameMaker will take, and you should understand how FrameMaker generates a CSS.

In this chapter

This chapter discusses how to generate a CSS in FrameMaker, and how EDD formatting information translates to a CSS. It contains these sections.

Default translation

Even though an EDD and a CSS both assign formatting properties to elements according to similar criteria, there are differences between the two. For example, in an EDD the indents of an element's text are specified absolutely; in CSS child elements inherit the indents of their parent elements, and additional indents accumulate with the inherited values.

In addition, the CSS statements generated by FrameMaker may not work on all browsers. For example, `AcrossAllColumns` and `AcrossAllSideheads` in an EDD translate to `display:compact` in CSS—this CSS statement is currently supported by the Opera6 browser, but not by NetScape Navigator 6.x or Internet Explorer 6.x.

Comparison of EDD format rules and CSS

Following is a table of format rule properties that can be used in an EDD, and how FrameMaker translates them to CSS. The table also indicates any CSS statements that are not supported by a specific browser. The browsers to be noted in this list are NetScape Navigator version 6.0 or later (NS), Internet Explorer version 6.0 or later (IE), and Opera version 6.0 or later (OP).

Font properties

Property in EDD	CSS statement	Values (and comments)	Browser support
Font size	font-size		ALL
Angle	font-style		ALL
Variation	font-variant		ALL
Weight	font-weight		ALL
Stretch	font-stretch		ALL
Case	text-transformation	uppercase or lowercase	ALL
Color	Color		ALL
Changebars	UNSUPPORTED		
CombinedFont	font-family		ALL
Family	font-family		ALL
OffsetHorizontal	UNSUPPORTED		
OffsetVertical	UNSUPPORTED		
Outline	UNSUPPORTED	No equivalent in CSS	
Overline	text-decoration	overline	ALL
PairKerning	UNSUPPORTED	No equivalent in CSS	
Shadow	text-shadow	Uses offsets of 0, 0 and same color	ALL
SizeChange	font-size	No equivalent in CSS	ALL
Spread	UNSUPPORTED	No equivalent in CSS	
SpreadChange	UNSUPPORTED	No equivalent in CSS	
StretchChange	UNSUPPORTED	No equivalent in CSS	
StrikeThrough	text-decoration	line-through	ALL
Superscript	vertical-align	Super	ALL
Subscript	vertical-align	Sub	ALL
Tracking	UNSUPPORTED		
TrackingChange	UNSUPPORTED		
Underline	text-decoration	underline	ALL
Language	UNSUPPORTED	No equivalent in CSS	
Tsume	UNSUPPORTED		

Indent properties

Property in EDD	CSS statement	Values (and comments)	Browser support
FirstIndent FirstIndentRelative FirstIndentChange	text-indent	Padding and border margins set to zero—NS and IE may not support negative indents	ALL
LeftIndent LeftIndentChange	margin-left	Padding and border margins set to zero	ALL
RightIndent RightIndentChange	margin-right	Padding and border margins set to zero	ALL
LineSpacing Height HeightChange	line-height		ALL
SpaceAbove SpaceAboveChange	margin-top		ALL
SpaceBelow SpaceBelowChange	margin-below		ALL
AsianSpacingroperties WesternAsianSpacing AsianAsianSpacing	UNSUPPORTED		ALL
Punctuation	UNSUPPORTED		ALL

Pagination properties

Property in EDD	CSS statement	Values (and comments)	Browser support
KeepWithNext	page-break-after:avoid		ALL
KeepWithPrevious	page-break-before:avoid		ALL
AcrossAllCols	display:compact		OP
AcrossCosSideHeads	display:compact		OP
InColumn	display:block		
RunInHead	display:run-in		NONE
SideHead	display:compact or float:left/right		ALL
DefaultPunctuation	after content		OP
Alignment	UNSUPPORTED	No equivalent in CSS	
Anywhere	UNSUPPORTED	Support unnecessary - this is unmodified pagination	

Property in EDD	CSS statement	Values (and comments)	Browser support
TopOfColumn	page-break-before:auto		ALL
TopOfLeftPage	page-break-before:right		ALL
TopOfPage	page-break-before:always		ALL
TopOfRightPage	page-break-before:left		ALL
WidowOrphanLines	widows/orphans		ALL
PgfAlignment	text-align		ALL

Autonumbering properties

Property in EDD	CSS statement	Values (and comments)	Browser support
AutoNumberFormat	list-style-type or counter	The choice of list-style-type or counter depends on the complexity of the autonumbering	ALL
AutoNumCharFormat	UNSUPPORTED	No equivalent in CSS—Generates char styles for element:before	OP
Position	beginning	Always position autonums at the beginning	ALL

Advanced properties

Property in EDD	CSS statement	Values (and comments)	Browser support
FrameAbove	border-top	FrameAbove and FrameBelow translate as border lines, only—the actual graphic in the reference frame does not translate to CSS	ALL
FrameBelow	border-bottom		ALL
LetterSpacing	letter-spacing		ALL
WordSpacing	word-spacing		ALL
Hyphenation	UNSUPPORTED	No equivalent in CSS	

Table cell properties

Property in EDD	CSS statement	Values (and comments)	Browser support
CellMargins	padding	In the EDD you can specify values for Change, Custom, or FromTableFormatPlus—padding only supports Custom	OP, NS
VerticalAlignment	vertical-alignment		ALL

Miscellaneous rules, properties, and elements

Property in EDD	CSS statement	Values (and comments)	Browser support
FirstParagraphRules	UNSUPPORTED		
LastParagraphRules	UNSUPPORTED		
Prefix/Suffix rules	before/after rules		OP
FormatChangeList	UNSUPPORTED	No equivalent in CSS	
Limits	UNSUPPORTED		
Rubi elements	UNSUPPORTED		
Use Paragraph Format		Generates appropriate CSS statements for the paragraph format	

EDD selectors This table lists element selectors for the element named XYZ

Selector in EDD	CSS selector	Browser support (and comments)
All contexts	XYZ{...}	ALL
List < Heading	Heading>List>XYZ	NS, OP
(List | Heading)	List > XYZ{...} Heading > XYZ{...}	NS, OP
{first} < Numbered	Numbered: first-child	NS
XYZ {after ABC}	ABC + XYZ	NS, OP
{middle} < Numbered	ABC > XYZ[fmcssattr="1"]	NS, OP
{last}< ABC	ABC > XYZ[fmcssattr="1"]	NS, OP
{notfirst} < ABC	ABC > XYZ[fmcssattr="1"]	NS, OP
{notlast} < ABC	ABC > XYZ[fmcssattr="1"]	NS, OP
only	ABC > XYZ[fmcssattr="1"]	NS, OP
{before ABC}	XYZ + ABC	NS, OP
{after ABC}	ABC + XYZ	NS, OP
{between ABC QRS}	ABC + XYZ + QRS {}	NS, OP
{any}	*	ALL

Selector in EDD	CSS selector	Browser support (and comments)
XYZ[att != val]	ABC > XYZ[fmcssattr="1"]	NS, OP
XYZ[att >= val]	ABC > XYZ[fmcssattr="2"]	NS, OP
XYZ[att <= val]	ABC > XYZ[fmcssattr="3"]	NS, OP
XYZ[att <val]	ABC > XYZ[fmcssattr="4"]	NS, OP
XYZ[att > val]	ABC > XYZ[fmcssattr="5"]	NS, OP
count ancestors named XYZ level is: 3	XYZ XYZ XYZ	NS, OP Supports up to 10 ancestors
count ancestors named XYZ stop counting at ancestor ABC If level is 2	ABC XYZ XYZ	NS, OP

Differences in translation

The format rules in an EDD express some things in ways that don't translate directly to CSS. The following section describes these differences, and how FrameMaker resolves them.

Right and left indents

In FrameMaker the right and left indents are expressed as absolute distances from the right or left edge of the text frame. For example, a section head may have a left indent of 24 pt from the text frame, and the body paragraphs following the head may have left indents of 48 pt. In FrameMaker these indents are expressed absolutely, 24 pt and 48 pt, respectively.

In CSS an element inherits indents from its ancestors. Using the example above, you would express the following:

```
Head{margin-left:24 pt;}
```

```
Head>Body{margin-left:24 pt;}
```

When FrameMaker generates a CSS, it translates its absolute indents to values for inherited margins that give you the same result.

Context-specific format rules

An EDD can include context-specific format rules that can define one or more possible contexts, with formatting changes for each context. The contexts are expressed in separate if, else/if, and else clauses. These clauses can include sibling indicators such as {middle}, {last}, {notfirst}, etc. For more information, see "Context-specific rules" on page 252.

CSS has no equivalent for these sibling indicators. For example, assume the following format rule for the Para element:

Element (Container): Para
General rule: <TEXT>
Text format rules
1. **If context is:** Section{notfirst}
Default font properties
Angle: Italic

In this case, the Para element uses italic text unless its parent Section element is the first of a group of siblings. In that case, the Para element will be italic.

There is no mechanism in CSS to express this selection. To reproduce the formatting, FrameMaker writes a special attribute to the affected Para element (in XML), and refers to that attribute in the CSS. These attributes have the reserved name of *fmcssattr*.

For example, to reproduce the above effect in CSS and XML, FrameMaker writes the following for the affected element:

para fmcssattr=”1”>

and it writes the following CSS statements:

para[fmcssattr=1] {font-style:italic;}

You should consider the following things when FrameMaker performs these actions:

- This is a modification to the XML, and it adds information that is specific to the FrameMaker product. This is deprecated in markup, because an important aspect of markup is that it doesn’t depend on any single product.
- If the XML complies with a DTD, the added attributes are likely to make the XML invalid.
- Unless you included these attributes in your EDD, you will get an invalid FrameMaker document when you import the XML that includes these attributes. To avoid this result, you can either define the attributes in the affected elements, or you can define read/write rules to drop these attributes on read.

Because of these issues, you can disable this effect in your application definition that you specify in `structapps.fm`. To disable this effect, you set *AddFmCSSAttrToXml* to *Disable* in your application definition.

If Else rules

The use of an Else statement in the EDD translates into two CSS statements. For example, with this statements in an EDD:

```
If Xref < Para Then color = Blue Else color = Black
```

the resulting CSS looks like:

```
Para > Xref{color: blue}
Xref {color = black;}
```

This will usually give appropriate results. There are circumstances when the EDD uses the order of processing to limit execution, which result in CSS that can't make the same distinction. For example, if the EDD includes:

```
If Emphasis < Emphasis color = blue Else if Emphasis Italic = Yes
```

the resulting CSS looks like this:

```
Emphasis > Emphasis {color: Blue;}
Emphasis {font-style: italic;}
```

In the EDD, if the first clause is true then the second clause will not take effect—Emphasis will be blue but not italic. With the CSS processing, the same element will appear both blue and italic.

Unsupported rules and statements

The following rules and statements in an EDD cannot translate to CSS. When it encounters these constructs, FrameMaker continues to generate the CSS, but ignores the constructs:

- Subrules and nested rules
- AND statements in selectors
- Table cell indents
- First and last paragraph rules

Generating a CSS

When working with FrameMaker you can choose to generate a CSS on command or automatically whenever you save a FrameMaker document as XML.

Generating a CSS on command

The simplest method to generate a CSS is to choose *Structure* > *Generate CSS2*. With this command, FrameMaker generates a CSS from the EDD information that is stored in the current document's element catalog. FrameMaker saves the CSS with the file name and location that you specify.

> **Important:** You should not perform this action with a template file. To correctly generate a CSS, FrameMaker must know which element is used as the root element for the document. An EDD can define a number of `ValidHighestLevel` elements, but the CSS must refer to only one. FrameMaker relies on actual use in the document to determine which is the root element. Since a template usually has no content, FrameMaker cannot make the determination. You should generate a CSS from a document that has content—either use an existing document, or insert a `ValidHighestLevel` element in your template before generating the CSS from it.

After you generate a CSS in this way, the stylesheet is not necessarily used by any XML documents. If you specify the same path and file name as an existing CSS, and that CSS is used by an XML document, the following will result:

- The new CSS will overwrite the older one.
- Any XML documents that specify a CSS of that path and file name will now use the new CSS.

Generating a CSS on Save As XML

When editing XML documents, you must control whether FrameMaker generates a CSS every time you save a document as XML. For example, if an XML document specifies an existing CSS, you may want to retain that specification each time you save the document as XML. On the other hand, if you generate a CSS on Save As XML, FrameMaker will change the CSS specification to use this generated CSS.

Specifying how FrameMaker generates a CSS

You specify how FrameMaker will generate a CSS by inserting elements in the structure application definition. These elements are all children of the *Stylesheets* element. You specify these elements as follows:

- To direct FrameMaker to generate a CSS, set *GenerateCSS2* to *Enable*.
- To specify a path for the stylesheet, provide a URI in *StylesheetURI*. If you don't provide a URI but have enabled *GenerateCSS2*, FrameMaker will save the generated stylesheet at the same location as the resulting XML document.
- To have FrameMaker support EDD context selectors in the CSS via a reserved attribute, set *AddFmCSSAttrToXml* to *Enable*.
- To direct FrameMaker to save the generated CSS in a specific location, provide a URI in the *StylesheetURI* element. Otherwise, FrameMaker saves a generated CSS in the same location as the ML document.
- To direct FrameMaker to save XML with a new stylesheet declaration that uses the URI specified in your structure application, set *RetainStylesheetPIs* to *Disable*.
- To ensure that the stylesheet declaration in the exported XML is the same as it was when you imported the XML, set *RetainStylesheetPIs* to *Enable*.
- To specify the type of stylesheet you are using, provide a string in the *StylesheetType* element. Currently, FrameMaker can only generate a CSS stylesheet. Even if you are not generating a stylesheet, you should provide a stylesheet type in case other processes rely on that data. For example, a structure API client may vary its processing depending on the value you provide here.

Stylesheet specifications in a structure application

You specify stylesheet parameters in the application definitions file, `structapps.fm`. For an XML application, you can include the following elements:

Stylesheets The parent element for the CSS specification. It contains *CssPreferences*, *RetainStyleSheetPIs*, *XmlStylesheet* and *XSLTPreferences* elements.

CssPreferences Specifies whether or not to generate a CSS, and whether to generate special attributes in the XML. It contains *GenerateCSS2*, *AddFmCSSAttrToXML*, and called *ProcessStylesheetPI* elements.

GenerateCSS2 Specifies whether FrameMaker will generate a CSS when you save the document as XML. It can be set to *Enable* or *Disable*. When this is set to Enable, FrameMaker generates a CSS. If a path is provided in StylesheetURI, FrameMaker saves the stylesheet to that location, with that filename. Otherwise, it saves the stylesheet to the same location as the XML document with a filename *xmldoc*.css, where *xmldoc* is the name of the XML document you're saving.

AddFmCSSAttrToXml Specifies whether FrameMaker will write instances of the *fmcssattr* attribute to elements in the XML document. It can be set to *Enable* or *Disable*. An EDD can include context selectors as criteria to assign format rules. CSS has no equivalent to this. When this is set to *Enable*, FrameMaker uses the *fmcssattribute* in certain elements so the CSS can achieve the same formatting as the EDD.

ProcessStylesheetPI This element can have one of the following values: Enable or Disable. If the value of the element is Enable, then the CSS file mentioned in the XML file is used while opening the XML file. The default value of the *ProcessStylesheetPI* element is Disable.

RetainStylesheetPIs Specifies whether FrameMaker will retain the stylesheet declaration for import and export of XML. It can be set to *Enable* or *Disable*. When this is set to *Enable*, FrameMaker does the following:

- On import, it stores the XML document's stylesheet PI as a marker in the FrameMaker document.
- On export, it writes the content of stylesheet PI marker in the resulting XML document.

XmlStylesheet Specifies the type of stylesheet and the URI for the stylesheet. It contains the *StylesheetType* and *StylesheetURI* elements.

StylesheetType Specifies the type of stylesheet. It contains a string for the stylesheet type. Currently, you can specify `CSS` (upper or lower case) or `XSL` (upper or lower case). If you specify `XSL`, FrameMaker will not generate a stylesheet.

StylesheetURI Specifies the URI for the stylesheet. It contains a string; for example, `/$STRUCTDIR/xml/xhtml/app/xhtml.css`.

XSLTPreferences Although this is an element of *Stylesheets*, is does not apply to CSS. Instead, it specifies how XSL transformations are to be performed before read rules are applied on import from XML, or after write rules are applied on export to XML. For details, see Developer Reference,

page 25: How the Stylesheets element affects XSL transformation and Chapter 30, "Additional XSL Transformation for XML."

20 Translating Elements and Their Attributes

Elements and their attributes are the fundamental components of both FrameMaker and markup documents.

Most information in this chapter is "bidirectional." It is about translations that can be recognized and performed when both reading and writing markup documents and DTDs, and the examples apply to both cases. You can start with a DTD containing element and attribute declarations and produce the corresponding FrameMaker element definition, or you can start with an EDD containing the FrameMaker element definition and produce the corresponding XML or SGML element and attribute declarations.

In this chapter

This chapter gives you an overview of how FrameMaker translates between its representation of elements and attributes and corresponding representations in markup.

This chapter provides general information about elements and their attributes. For specific information on how to translate elements and attributes used for a particular purpose, see Chapters 23 through 29.

This chapter contains these sections.

Default translation

The basic representations of elements and attributes in markup and in FrameMaker are very similar, facilitating the translation in both directions. This section provides information on how the parts of element and attribute definitions translate by default.

> **Important:** For importing and exporting XML, FrameMaker has limited support of double-byte characters in markup tokens such as GIs and attribute names. For more information, see "Supported characters in element and attribute names" on page 100 and Developer Reference, page 27: Specifying the character encoding for XML files.
>
> SGML does not support double-byte characters in markup tokens.

Translating model groups and general rules

The general rule of a FrameMaker element uses a syntax based on SGML model groups. FrameMaker uses the delimiter strings defined in the SGML reference concrete syntax for connectors and occurrence indicators. For example, suppose you have the following declaration:

```
<!ELEMENT lablist - - (head, par?, item+)>
```

By default, the corresponding FrameMaker element definition is:

Element (Container): Lablist
General rule: Head, Par?, Item+

When FrameMaker converts a DTD to an EDD, it performs the following translations:

- The token `#PCDATA` in a content model translates to the token `<TEXT>` in a FrameMaker general rule.
- A content model of `ANY` translates to the FrameMaker general rule `<ANY>`.
- A declared content of either `CDATA` or (for SGML) `RCDATA` translates to the FrameMaker general rule `<TEXTONLY>`.
- A declared content of `EMPTY` translates to the FrameMaker general rule `<EMPTY>`.

When FrameMaker converts an EDD to a DTD, it performs the reverse translations. In the case of a <TEXTONLY> general rule it produces a declared content of PCDATA for XML, and a declared content of RCDATA for SGML.

Translating attributes

In markup, attribute declarations for an element occur in a separate attribute definition list declaration. In FrameMaker, the attribute definitions for an element are directly part of that element's definition.

For example, assume you have the following declarations in SGML:

```
<!ELEMENT lablist - - (head, par?, item+)>

<!ATTLIST lablist
id ID #IMPLIED
type (num | bul) bul
sec (u | s | t) #REQUIRED>
```

By default, these translate to the following element definition in FrameMaker:

Element (Container): Lablist
General rule: Head, Par?, Item+
Attribute list

1. Name: Id	**Unique ID**	**Optional**
2. Name: Type	**Choice**	**Optional**
Choices: Num | Bul		
Default: Bul		
3. Name: Sec	**Choice**	**Required**
Choices: U | S | T		

Note that the first two markup attributes become optional attributes in FrameMaker. In addition, the interpreted attribute value specification given in the default value for the markup type attribute translates to a default value in FrameMaker.

In general, any markup attribute that is not a required attribute (that is, doesn't use the #REQUIRED declared value) becomes an optional attribute in FrameMaker. If the markup attribute has an attribute value specification, the interpreted version of that value becomes the attribute's default value in FrameMaker.

By default, a small number of markup attributes translate to formatting properties in FrameMaker. This happens only in the case of graphics, equations, and tables that use a standard representation recognized by FrameMaker. For information on these attributes, see Chapter 23, "Translating Tables," and Chapter 24, "Translating Graphics and Equations."

When FrameMaker writes a FrameMaker document as XML or SGML, it writes markup attribute specifications for attributes with an explicitly supplied value. Such explicit values may be entered directly by the end user, created by the FrameMaker cross-reference facility, or supplied by an FDK client.

Conversely, on import of a markup document, if an imported element doesn't include a value for an attribute, then FrameMaker only supplies a value if the EDD includes a default value for the attribute—otherwise FrameMaker doesn't supply a value for such an attribute.

Attribute types and declared values

FrameMaker has a set of attribute types that correspond to the declared values for attributes in XML or SGML, but neither is a subset of the other. That is, multiple declared values in markup can become the same FrameMaker attribute type, and conversely multiple FrameMaker attribute types can become the same markup declared value. (XML and SGML do not define the term *attribute type*. Loosely speaking, you can think of an attribute's declared value as its type.)

When you create an EDD from a DTD or a DTD from an EDD, FrameMaker uses a default translation between attribute types and declared values.

> **Note: XML:** XML does not support all the declared values that are supported in SGML. The following tables identify the declared values that are valid only in SGML. For XML, substitute these declared values as follows:
>
> - `NAME`, `NUMBER`, and `NUTOKEN` to `NMTOKEN`
> - `NAMES`, `NUMBERS`, and `NUTOKENS` to `NMTOKENS`

On import, FrameMaker makes the following conversions in the absence of read/write rules:

Declared Value in Markup		FrameMaker Attribute Type
`CDATA`		`String`
`ENTITY`		`String`
`ENTITIES`		`Strings`
`ID`		`UniqueID`
`IDREF`		`IDReference`
`IDREFS`		`IDReferences`
`NAME`	SGML-only	`String`
`NAMES`	SGML-only	`Strings`
`NMTOKEN`		`String`
`NMTOKENS`		`Strings`
`NUMBER`	SGML-only	`Integer`
`NUMBERS`	SGML-only	`Integers`
`NUTOKEN`	SGML-only	`String`
`NUTOKENS`	SGML-only	`Strings`
`NOTATION`		`Choice`
Name token group		`Choice`

On export, FrameMaker makes the following conversions in the absence of read/write rules:

FrameMaker Attribute Type	Declared Value in Markup
String	CDATA
Strings	CDATA
Integer	**SGML:** NUMBER if values are restricted to be positive, CDATA otherwise. **XML:** NMTOKEN
Integers	**SGML:** NUMBERS if values are restricted to be positive, CDATA otherwise **XML:** NMTOKENS
Real	**SGML:** NUMBER if value is restricted to be positive, CDATA otherwise **XML:** NMTOKEN
Reals	**SGML:** NUMBERS if values are restricted to be positive, CDATA otherwise **XML:** NMTOKENS
UniqueID	ID
IDReference	IDREF
IDReferences	IDREFS
Choice	Name token group

Naming elements and attributes

Legal names in FrameMaker and in markup are different. FrameMaker element tags and attribute names can be longer than allowed by the SGML reference concrete syntax and can contain characters (such as the $ character) that are not allowed by the naming rules of the reference concrete syntax.

In XML a name can contain Unicode characters. For information about FrameMaker support for unicode in names, see "Supported characters in element and attribute names" on page 100.

If you create an EDD by importing an SGML DTD, you won't have trouble with illegal names. Any name that is legal in the SGML reference concrete syntax is legal in FrameMaker. However, if you create an SGML DTD by exporting an EDD, you are likely to encounter naming difficulties unless you have adhered to SGML naming rules in your EDD. You will need to write rules to handle the differences.

Likewise, if you create an EDD by importing an XML DTD, you will have no problems. If you create an XML DTD by exporting an EDD, you should not encounter problems with name length, but you

will need to ensure the element and attribute names use only legal characters. Otherwise you will need to write rules to reflect the differences.

> **Important:** For importing and exporting XML, FrameMaker has limited support of double-byte characters in markup tokens such as GIs and attribute names. For more information, see "Supported characters in element and attribute names" on page 100and Developer Reference, page 27: Specifying the character encoding for XML files.
>
> SGML does not support double-byte characters in markup tokens.

If you do not specify the name for an element or attribute when translating between markup and FrameMaker, FrameMaker must pick a name to use. The chosen name has the same characters as the original. For XML, FrameMaker preserves the case of names on import and export.

For SGML the case conventions are often different than case conventions in FrameMaker. To ensure that the case of a name is appropriately chosen, FrameMaker uses this information:

- If specified, an `element` rule that explicitly translates a generic identifier to a FrameMaker element tag

  If an `element` rule gives both markup and FrameMaker names for an element, the software uses the names and their case as specified.

- If specified, an `attribute` rule that explicitly translates a markup attribute name to a FrameMaker attribute name

  If an `attribute` rule gives both markup and FrameMaker names for an attribute, the software uses the names and their case as specified.

- The `NAMECASE` parameter in the SGML declaration

  If the `NAMECASE` parameter states that names are case sensitive, then the software preserves the case of all characters on import and export. If the `NAMECASE` parameter states that names are not case sensitive (as it does for general names in the reference concrete syntax, which FrameMaker uses by default), then:

  – On import, the software converts a generic identifier to an element tag that has an initial capital letter followed by lowercase characters. For example, if a generic identifier is written in the SGML document as `part`, `Part`, `PART`, or `pArt`, it becomes the FrameMaker element tag `Part`. The software performs the same conversion for attribute names.

  – On export, the software converts a FrameMaker element tag to a generic identifier that has all lowercase characters. So, each of the FrameMaker element tags `part`, `Part`, `PART`, and

pArt, become the generic identifier part. The software performs the same conversion for attribute names.

Important: If the NAMECASE parameter states that names are not case sensitive (NAMECASE GENERAL YES), the same characters with different capitalization in FrameMaker export as the same element GI or attribute name. For example, FrameMaker elements named Part and part both export as the SGML element part, resulting in an error. If the SGML declaration states that names *are* case sensitive, these FrameMaker elements become distinct SGML elements as intended.

Inclusions and exclusions

Note: XML: The XML specification does not allow inclusions and exclusions. The following discussion pertains to SGML, only.

Just as SGML content models can specify exceptions to the model group, FrameMaker element definitions can specify exceptions to the general rule. The following is an element declaration that lists inclusions:

```
<!ELEMENT book - - (front, body, back) +(index)>
```

On import, FrameMaker produces this FrameMaker element definition:

Element (Container): Book
General rule: Front, Body, Back
Inclusions: Index

Similarly, the following is an element definition that lists exclusions:

Element (Container): Appendix
General rule: Title, Section+
Exclusions: List, Table

On export, FrameMaker produces this SGML element declaration:

```
<!ELEMENT appendix - - (title, section+) -(list, table)>
```

Line breaks and record ends

You can control the behavior of markup record ends and FrameMaker line breaks on import and export of documents.

By default, when importing a markup document, FrameMaker treats a data record end within a text segment as a space. It ignores record ends in other locations. You can change this behavior for record ends that are not ignored by the XML or SGML parser.

By default, when exporting a FrameMaker document, FrameMaker behaves as follows:

- When exporting the text of a paragraph, it ignores line breaks. It includes a space separating the two words on either side of a line break.

- It generates a record end at the end of every paragraph and flow in the FrameMaker document.

For information on how you can change this behavior, see Developer Reference, page 141: line break.

Modifications to the default translation

You can use rules to change the translation of elements and their attributes. The most common change is to rename them upon transfer between FrameMaker and markup.

The following sections describe some of the possible modifications to the default translations. Your situation may require different rules or a structure API client, or you may use these rules in ways not discussed in these sections; the cross-references point to related information. In addition, the other translation chapters point to further information on particular element types.

For a summary of read/write rules relevant to translating all elements and attributes, see Developer Reference, page 33: All Elements. For information on writing structure API clients, see the*Structure Import/Export API Programmer's Guide*.

Renaming elements

There are many reasons you may choose to rename elements when translating between markup and FrameMaker. As already mentioned, the FrameMaker naming conventions are less restrictive than those of SGML. If you wish to take advantage of this, you could rename your elements on import and export.

> **Important:** Note that read/write rules do not support double-byte characters, so you cannot use rules to convert element names that have double-byte characters.

The general form of the rule for renaming an element is:

```
element "gi"
   is fm element "fmtag";
```

where *gi* is a generic identifier and *fmtag* is a FrameMaker element tag. For example, if you have an XML element `par` the FrameMaker element's name is by default `par`. To change this default behavior, you could use this rule:

```
element "par"
   is fm element "Paragraph";
```

With this rule, import of a document or DTD translates the XML element `par` as the FrameMaker element `Paragraph`. Conversely, export of a FrameMaker document or EDD translates the FrameMaker element `Paragraph` as the XML element `par`.

For information on these rules, see Developer Reference, page 56: element and Developer Reference, page 110: is fm element.

Renaming attributes

Just as you can choose to rename elements, you can choose to rename their attributes when translating between markup and FrameMaker. You can rename an attribute either for all elements in which it occurs or only for a particular attribute. To do so, you use one of these rules:

```
attribute "attr"
   is fm attribute "fmattr";

element "gi"
   attribute "attr"
       is fm attribute "fmattr";
```

> **Important:** Note that read/write rules do not support double-byte characters, so you cannot use rules to convert attribute names that have double-byte characters.

where *attr* is a markup attribute name, *fmattr* is a FrameMaker attribute name, and *gi* is a generic identifier.

The first form renames the attribute no matter in which element it occurs. For example, if you have an XML attribute `sec`, the FrameMaker attribute's name is by default `sec`. If this attribute occurs in several elements and you want the same alternate name for all those elements, you could use this rule:

```
attribute "sec" is fm attribute "Security";
```

With this rule, import of a document or DTD translates the markup attribute `sec` as the FrameMaker attribute `Security`. Conversely, export of a FrameMaker document or EDD translates the FrameMaker attribute `Security` as the markup attribute `sec`.

Sometimes you may wish to rename an attribute differently for different elements. For example, assume you use an attribute named `ref` for the elements `article` and `xref`. If these attributes have different meanings you may want to choose different names for them in FrameMaker. You could use these rules:

```
element "article"
   attribute "ref" is fm attribute "Reference";

element "xref" {
   is fm cross-reference element "XRef";
   attribute "IDref" is fm attribute "ID";
}
```

For information on these rules, see

- Developer Reference, page 46: attribute
- Developer Reference, page 56: element
- Developer Reference, page 104: is fm attribute
- Developer Reference, page 109: is fm cross-reference element.

Renaming attribute values

As with attributes and elements, there are also reasons you may choose to rename the members of a name token group when translating between markup and FrameMaker. You can rename their values either for all occurrences, or within a single context. To do so, you use one of these rules:

```
value "val" is fm value "fmval";

attribute "attr" value "val" is fm value "fmval";

element "gi" attribute "attr" value "val" is fm value "fmval";
```

where *val* is a name token, *fmval* is a string, *attr* is an XML attribute, and *gi* is a generic identifier.

For example, assume the attribute `color` is used for many elements with the values `r`, `b`, and `g`. You could write this rule to rename the values in the same way for all occurrences of the `color` attribute:

```
attribute "color" {
   value "r" is fm value "Red";
   value "b" is fm value "Blue";
   value "g" is fm value "Green";
}
```

For information on these rules, see:

- Developer Reference, page 46: attribute
- Developer Reference, page 56: element
- Developer Reference, page 163: value
- Developer Reference, page 138: is fm value.

Translating a markup element to a footnote element

Many documents require footnotes to provide additional information. Handling of footnotes is primarily a formatting issue. Because of this, markup does not directly address them. FrameMaker, however, has a special type of element for creating and formatting footnotes.

Because FrameMaker maintains footnotes as elements, there is no problem when writing a FrameMaker document as markup data. A footnote element becomes an XML or SGML element. However, if you have a markup document or DTD with an element that you want to treat as a footnote in FrameMaker, you must write a rule to indicate this.

For example, to translate the markup element `fn` as a footnote element named `Footnote` in FrameMaker, use this rule:

```
element "fn" is fm footnote element "Footnote";
```

In FrameMaker, a footnote element is always a footnote element. That is, you can't import the same element as a footnote in some contexts and as a normal paragraph in other contexts. If you have a single markup element that can be formatted as both a footnote and as a normal paragraph, you'll need to translate the same markup element to two different FrameMaker elements. This translation cannot be accomplished with read/write rules. In this case, you'll need to write a structure API client.

For information on these rules, see Developer Reference, page 56: element and Developer Reference, page 112: is fm footnote element. For information on writing structure API clients, see the *Structure Import/Export API Programmer's Guide*.

Translating a markup element to a Rubi group element

Documents that include Japanese text most likely require Rubi to express the pronunciation of certain words. Handling of Rubi is primarily a formatting issue. Because of this markup does not directly address Rubi. However, FrameMaker has a special type of element for creating a *Rubi group*. The Rubi group includes a range of text for the Oyamoji, and it assigns Rubi text to the Oyamoji.

Since the Rubi group is an element, when writing a FrameMaker document as markup a Rubi Group element and its contents become markup elements. However, if you have markup document or DTD with an element you want to translate as a Rubi group in FrameMaker, you must use read/write rules to indicate this.

The minimal structure for a Rubi group is:

The XML specification suggests a representation of a Rubi group that includes Oyamoji in a container element as follows:

```
<ruby>
<rb>Oyamoji text</rb>
<rt>Rubi text</rt>
</ruby>
```

To translate this Rubi group, use these rules:

```
element "ruby" is fm rubi group element "MyRubiGroup";
element "rb" is fm rubi element "MyRubi";
element "rt" is fm rubi element "Oyamoji";
```

If the element named MyRubiGroup allows the Oyamoji element in its content rule (and if the EDD includes a definition for the Oyamoji element) the markup will translate to FrameMaker data with the following structure:

Another typical representation of a Rubi group in markup is:

```
<rubigroup rubi = "Rubi text">Oyamoji text</rubigroup>
```

FrameMaker does not support direct translation of this representation. To create a Rubi group when the Rubi text is represented as an attribute, you'll need to write an XSLT style sheet or a structure API client.

If you have an XML or SGML element that is used as a Rubi group in some cases and some other text range in other cases, you'll need to translate the same markup element to two different FrameMaker elements. This translation cannot be accomplished with read/write rules. In this case, you will need to write a structure API client.

For information on these rules, see Developer Reference, page 56: element, Developer Reference, page 130: is fm rubi group element, and Developer Reference, page 129: is fm rubi element. For information on writing structure API clients, see the *Structure Import/Export API Programmer's Guide*.

Changing declared content of a markup element associated with a text-only element

XML: `RCDATA` is not valid in XML, so you should never find yourself importing XML that has an element declared as `RCDATA`. When saving to XML, if an element is declared as `<TEXTONLY>`, FrameMaker translates its general rule as `(#PCDATA)`.

SGML: By default, FrameMaker translates an element with a general rule of `<TEXTONLY>` as an SGML element with declared content `RCDATA`.

An `RCDATA` element can include entity and character references. If you want your markup element not to allow entity references, you must manually edit the DTD and change the declared content of the element to `CDATA`.

Note, however, that the FrameMaker editing environment cannot enforce this change for you. Your end user can insert user variables or special characters into a `<TEXTONLY>` element, even if the corresponding markup element is `CDATA`. The entity or character references normally used to represent such data are not recognized within `CDATA` content.

If the end user uses such constructs and exports that document to markup, the software reports an error in the log file. If your application contains `CDATA` elements, you should make your end users aware of this restriction.

Retaining content but not structure of an element

There may be elements in your DTD that you do not need in your EDD. Similarly, there can be elements in your EDD that you do not need in your DTD. In this case, you can write rules to retain the contents of an element while removing the element itself. If you do this, however, any attributes associated with the element will be discarded. Also, if your document structure must be preserved on a round trip to and from markup, you must write a structure API client or XSLT style sheet to recreate the element on transfer in the other direction.

To remove element structure, but not content, use one of these rules:

```
element "gi" unwrap;

fm element "fmtag" unwrap;
```

For example, assume you have these element declarations in your DTD:

```
<!ELEMENT memo (top, body)>
<!ELEMENT top (to, from, date)>
<!ELEMENT body (par+)>
```

You can choose to omit the `top` and `body` elements on import to FrameMaker. Use these rules:

```
element "top" unwrap;
element "body" unwrap;
```

Import of the DTD produces this element definition:

Element (Container): Memo
General rule: ((To, From, Date), Par+)

For information on these rules, see Developer Reference, page 56: element, Developer Reference, page 77: fm element, and Developer Reference, page 160: unwrap.

If you use one of these rules, you may want to use the `preserve fm element definition` rule to aid maintenance of your EDD. If you do not, updating your EDD on the basis of a modified DTD may change your EDD definitions in ways you do not intend. For information on this rule, see Developer Reference, page 146: preserve fm element definition.

For information on structure API clients, see the *Structure Import/Export API Programmer's Guide*. For more information on XSLT transformation, see Chapter 30, "Additional XSL Transformation for XML."

Retaining structure but not content of an element

You can use some elements in markup or FrameMaker as place holders for information generated by a tool. For example, FrameMaker can generate a table of contents for a document. So when you import a markup element containing a table of contents you may choose to have the software generate a new version of the content instead of including the text of the element. To discard an element's content but retain its structure, use one of these rules:

```
element "gi"
   reader drop content;

element "gi"
   writer drop content;
```

For example, assume your DTD has this element declaration:

```
<!ELEMENT toc (#PCDATA)>
```

In markup, you depend on an application to create the appropriate content for the `toc` element. In FrameMaker, you want to use its tools for generating a table of contents. To do so, first write the rule:

```
element "toc" reader drop content;
```

To generate the appropriate content for the element on import to FrameMaker, you have two choices. You can require the end user to manually call the Generate command, or you can create an FDK client to generate the table of contents as the last step in importing the document. With this rule, when you export the FrameMaker document to markup the `toc` element retains its content in the exported document.

For information on these rules, see Developer Reference, page 56: element, Developer Reference, page 55: drop content, Developer Reference, page 151: reader, and Developer Reference, page 166: writer. For information on creating an API client, see the *FDK Programmer's Reference*.

Formatting an element as a boxed set of paragraphs

The formatting associated with your documents may require that the paragraphs in an element appear in a completely or partially boxed area. In FrameMaker, you must have a table element to get this formatting. For information on how to translate a single markup element as a one-cell table, see "Using a table to format an element as a boxed set of paragraphs" on page 360.

Suppressing the display of an element's content

You may have a markup element you want to retain but whose content you do not want displayed when you translate markup documents containing the element to documents in FrameMaker. The simplest method of accomplishing this is to translate the element to a marker element and its content to marker text.

For example, if you have a `comments` element that should not print in the document but whose content you do not want to lose, you can use these rules:

```
element "comments" {
   is fm marker element;
   marker text is content;
}
```

For more information on marker elements, see Chapter 27, "Translating Markers." For more information on these rules, see Developer Reference, page 56: element, Developer Reference, page 115: is fm marker element, and Developer Reference, page 142: marker text is.

Alternatively, you may not want to create a marker element from your markup element. If you do not, there are other ways you could suppress display of the element's content, but these methods require you to write a structure API client. For example, you could have an API client attach a condition tag to the element and then hide that condition tag. For information on creating structure API clients, see the *Structure Import/Export API Programmer's Guide*.

Discarding a markup or FrameMaker element

You can use markup elements for purposes that have no counterpart in FrameMaker, and the reverse is also true. For example, assume you use an element `break` to indicate in your markup that a page or line break is desired. You can discard this element when the markup document is imported to FrameMaker and have a structure API client insert the necessary break; there is no need to mark the break with an element. To drop an element on import, use this rule:

```
element "break" drop;
```

As another example, you may have a table that always has the same column headings. For this reason, you choose not to include an element for the column headings in the XML or SGML representation. If your table is to have column heading in FrameMaker, however, the headings must be in elements; you would discard these elements on export using this rule:

```
fm element "ColumnHead" drop;
```

The `drop` rule actually discards the data—if you use the `drop` rule and want the element to be restored when you are translating a document in the other direction, you must write a structure API client, or an XSLT style sheet.

For information on these rules, see Developer Reference, page 56: element, Developer Reference, page 77: fm element, and Developer Reference, page 53: drop.

If you use either of these rules, you may want to use the `preserve fm element definition` rule to aid in the maintenance of your EDD. If you do not, updating your EDD on the basis of a modified DTD may change your EDD definitions in ways you do not intend. For information on this rule, see Developer Reference, page 146: preserve fm element definition.

For information on creating a structure API client, see the *Structure Import/Export API Programmer's Guide*. For more information on XSLT transformation, see Chapter 30, "Additional XSL Transformation for XML."

Discarding a markup or FrameMaker attribute

If you use attributes in markup for purposes that have no counterpart in FrameMaker, or for purposes for which you must supply a structure API client, you can choose to discard these attributes. To discard an attribute for all elements that use it, use one of these rules:

```
attribute "attr" drop;

fm attribute "attr" drop;
```

where *attr* is an XML, SGML, or FrameMaker attribute name.

To discard the attribute for a single element, use one of these rules:

```
element "gi" attribute "attr" drop;

element "gi" fm attribute "attr" drop;
```

where *gi* is a generic identifier and *attr* is an XML, SGML, or FrameMaker attribute name.

For example, assume that you have these declarations in SGML:

```
<!ELEMENT par (#PCDATA)>
<!ATTLIST par fontsize NUMBER 10>
```

The `fontsize` attribute indicates formatting for the element. You'll have to specify that information in some other way, perhaps with a format rule in your EDD or with a structure API client. To have the software discard the attribute `fontsize` on import, use this rule:

```
element "par" attribute "fontsize" drop;
```

If you use the `drop` rule and want the attribute to be restored when you are translating a document in the other direction, you must write a structure API client.

For information on these rules, see Developer Reference, page 46: attribute, Developer Reference, page 53: drop, Developer Reference, page 56: element, and Developer Reference, page 76: fm attribute. For information on writing format rules, see Chapter 15, "Text Format Rules for Containers, Tables, and Footnotes." For information on creating a structure API client, see the *Structure Import/Export API Programmer's Guide*.

Specifying a default value for an attribute

An attribute in markup can be implicit, in which case end users don't need to provide a value for it. For example, your DTD may declare the following:

```
<!ATTLIST (appendix | chapter | reference | section)
    label (ch | app) #IMPLIED>
```

In markup, the appendix, chapter, reference, and section elements can include either ch or app as values for the label attribute, but they don't have to. You may want to set up your EDD to format these elements based on the value of the label attribute. Further, you may want to use a default attribute value to format these elements in case the markup doesn't include a value for the label attribute.

FrameMaker provides a rule to set up a default value for attributes in the EDD when you import a DTD. To do this at the highest level, for all elements that use the relevant attribute, use this rule:

```
attribute "attr" implied value is "val";
```

where *attr* is a markup attribute and *val* is the proposed value for that attribute.

To specify a value only for the attribute within a particular markup element, use this rule:

```
element "gi" attribute "attr" implied value is "val";
```

where *gi* is a generic identifier, *attr* is an attribute in the element, and *val* is the proposed value for that attribute.

In the above example, assume the default value for label should be ch for most of these elements, but app for an appendix element. You can specify values to use in all cases with these rules:

```
attribute "label" implied value is "ch";
element "appendix" attribute "label" implied value is "app";
```

Note that this rule is for importing DTDs and exporting EDDs. In FrameMaker, a default attribute value can only be specified in the EDD, so this rule has no effect when importing an XML or SGML instance, or when exporting a FrameMaker document. Also, the default value is used only to initiate formatting. FrameMaker does not actually provide an attribute value for the element. When you export the FrameMaker document, the software does not generate an attribute value where there previously was none.

For more information, see "Default value" on page 195 and "Default translation" on page 293 For information on these rules, see Developer Reference, page 56: element, Developer Reference, page 46: attribute, and Developer Reference, page 96: implied value is.

Changing an attribute's type or declared value

Declared values for attributes in markup, and FrameMaker attribute types are different. In the absence of read/write rules, FrameMaker must make decisions on how to translate between them. The default translations are listed in "Attribute types and declared values" on page 295. You can change the attribute type created for an EDD or the declared value for a DTD by using the following rule:

```
[sgmldv | xmldv] attribute "attr"
       is fm [read-only] [fmtype] attribute "fmattr"
       [range from a to b];
```

where *sgmldv* is an SGML declared value, *xmldv* is an XML declared value, *attr* is a markup attribute name, *fmtype* is a FrameMaker attribute type, *fmattr* is a FrameMaker attribute name, and *a* and *b* are numbers.

SGML: The possible values for *structdv* are:

```
cdata          id             name           number
entity         idref          names          numbers
entities       idrefs         nmtoken        nutoken
group          notation       nmtokens       nutokens
```

XML: The possible values for *xmldv* are:

```
cdata          id             notation
entity         idref          idrefs
entities       idrefs         notation
group
```

The possible values for *fmtype* are:

```
string         integers       unique-id
strings        real-number    id-reference
choice         real-numbers   id-references
integer
```

For example, assume your DTD defines `size` as an attribute of type `CDATA`. If you know that values of this attribute always represent real numbers, you can use this rule:

```
cdata attribute "size" is fm real-number attribute;
```

As another example, if your SGML DTD defines `perc` as an attribute of type NUMBER, you can use this rule to change the attribute name:

```
NUMBER attribute "perc" is fm integer attribute "Percentage";
```

Be careful with the use of this rule. FrameMaker allows you to translate any declared value to any attribute type. For instance, in the `size` example, if you are wrong that the attribute is always a real number, your end users may encounter errors reading their markup documents.

For information on read-only attributes, see "Creating read-only attributes," next. For information on these rules, see Developer Reference, page 46: attribute and Developer Reference, page 104: is fm attribute.

Creating read-only attributes

FrameMaker allows you to define read-only attributes. A read-only attribute is one whose value cannot be changed by an end user.

For example, assume the following situation. You use a structure API client to extract elements from an XML database that you then combine into a FrameMaker document. In the database, the elements are identified by keys. You need to be able to store an edited version of these elements back in the database, so you need to retain the keys to use. To do so, you have your structure API client store them as the value of an attribute of the element when read into FrameMaker. Also, you do not want your users to change the value of the attribute, as that would invalidate the key. To prevent that, you make the attribute a read-only attribute.

Another common use of read-only attributes is for `ID` and `IDREF` attributes associated with cross-references. Since FrameMaker can automatically maintain these attribute values for your end users, you may choose not to let them manually modify the values.

To specify an attribute to be read-only, use the following rule:

```
attribute "attr" is fm read-only attribute ["fmattr"];
```

where *attr* is an attribute name in markup, and *fmattr* is a FrameMaker attribute name.

For information on these rules, see Developer Reference, page 46: attribute and Developer Reference, page 104: is fm attribute.

Using markup attributes to specify FrameMaker formatting information

A common use of attributes is for storing formatting information. FrameMaker does not recognize markup attributes used in this way by default. However, your EDD can contain format rules that do use attributes in this way.

To create the appropriate translation, you may need to use read/write rules to rename attribute values. In addition, you must manually add the appropriate attribute-based format rules to your EDD.

For example, assume you have these declarations:

```
<!ELEMENT par - - (#PCDATA)>
<!ATTLIST par
 font (helv | times | courier) #REQUIRED
 weight (b | r) r>
```

The attributes `font` and `weight` indicate information needed to properly format the paragraph. You can use format rules on the `Par` element to set the font family and weight on the basis of these two attributes so you might use this element definition in your EDD:

Element (Container): Par
General rule: <TEXT>
Attribute list
1.Name: Font **Choice** **Required**
Choices: Helv | Times | Courier
1.Name: Weight **Choice** **Optional**
Choices: B | R
Default: R
Text format rules
1. If context is: [Font = "Helv"]
Default font properties
Family: Helvetica
Else, if context is: [Font = "Times"]
Default font properties
Family: Times
Else
Default font properties
Family: Courier
2. If context is: [Weight = "B"]
Default font properties
Weight: Bold
Else
Default font properties
Weight: Regular

In format rules, you can test for specific attribute values but cannot use the value of an attribute to explicitly set a formatting property. For this reason, if the legal attribute values cannot be enumerated in a format rule, you must write a structure API client to pick up the information. For example, if your `par` element has a numeric attribute `size` intended to specify the font size and you do not know what the legal set of font sizes will be, you cannot set the size font property from that attribute value in a format rule. Instead, you must write an API client to set the size on import of a markup document.

FrameMaker automatically handles certain FrameMaker formatting properties associated with tables, graphics, and equations. For information on these properties, see Chapter 23, "Translating Tables," and Chapter 24, "Translating Graphics and Equations."

For information on writing format rules, see Chapter 15, "Text Format Rules for Containers, Tables, and Footnotes." For information on creating a structure API, see the online manual *Structure Import/Export API Programmer's Guide*, included with the FDK.

21 Translating Entities and Processing Instructions

Entities in markup serve a variety of functions in markup documents. These functions can be represented in widely different ways in FrameMaker. FrameMaker does not have a single construct that corresponds with entities in markup. For information on common uses of entities, see "Entities" on page 92.

Processing instructions (PIs) in markup provide a mechanism for invoking system-specific actions from within an XML or SGML document. These actions can be almost anything. By default, FrameMaker interprets a small number of processing instructions. It stores others in the document as markers—you can write an FDK client to process them at any time in the authoring cycle. You can also use a structure API client to trap processing instructions and perform specific actions as the parser encounters them.

In this chapter

This chapter discusses the default translations used by FrameMaker for processing instructions and for entities of various kinds. The chapter also provides general information on how you can modify these translations. Subsequent chapters discuss further modifications appropriate to translating particular FrameMaker constructs such as variables or books. For information on the processing instructions used for conditional text, see Chapter 28, "Translating Conditional Text."; for those used for books and book components, see Chapter 29, "Processing Multiple Files as Books."

This chapter contains these sections.

Default translation

With few exceptions, FrameMaker preserves all of the entity structure of a document. When exporting a document to XML or SGML, FrameMaker generates entity references for user variables, graphics, equations, text insets, books, and book components by default. However, while it can read a DTD that uses parameter entities, the software does not mark the replacement text as an entity within the FrameMaker EDD. When you save the resulting EDD as a DTD, the result will not include the parameter entities that were in the original DTD.

FrameMaker creates processing instructions (PIs) for a small set of special cases on export. Similarly, on import it interprets only the same small set of processing instructions. It retains other PIs as markers and translates them back into PIs on export.

FrameMaker preserves text entities that are references to external text files. On import, these entities are translated as text insets to the referenced file. On export, the software writes the entity reference in the XML or SGML instance; if the entity is not defined in the structure application's DTD, the software writes an entity declaration in the internal DTD subset for the exported document instance.

If a referenced text file is an XML or SGML fragment (at least one element, but no SGML declaration or DTD subset), the software interprets its structure according to the current structure application. If the referenced file is text, the software treats the file's text as the content of the element containing the entity reference.

On export to markup

FrameMaker uses entities for text insets, user variables, graphics, equations, books, and book components. If the entities are not declared in the structure application's DTD, FrameMaker generates appropriate entity declarations in the internal DTD subset of the exported document instance.

If the FrameMaker document was made by importing an XML or SGML instance, and that instance had entities declared in the internal DTD subset, FrameMaker will regenerate those entity declarations when exporting the document to markup.

If the user created a FrameMaker document object that must export as an entity, and there is no information the software can use to map that object to an existing entity declaration, the software will generate a declaration in the internal DTD subset of the document instance.

For information on export of variables, graphics, equations, books, or book components, see Chapter 26, "Translating Variables and System Variable Elements," Chapter 24, "Translating Graphics and Equations," or Chapter 29, "Processing Multiple Files as Books."

Generating processing instructions on export

> **Note: XML and SGML:** The XML specification defines the PI closing delimiter as `?>`, while in SGML the closing delimiter is `>`. This section uses the XML specification to illustrate PI syntax.

By default, FrameMaker creates processing instructions under the following circumstances:

- When the exported markup turns out to be invalid, the software generates a `<?FM Validation Off?>` PI. When importing an XML or SGML instance with this PI in its DTD subset, FrameMaker imports the complete file whether it is valid or not. Note that this PI is only reliable for XML or SGML files that were exported by FrameMaker.
- When exporting a book, the software generates a `<?FM book?>` PI to identify a book, and an entity reference, `&bkc#;` to identify each book component, for example, `&bkc1;`, `&ckc2;`, and so forth. Each entities begins with a `<?FM document filename?>` PI. For information on export of books and book components, see Chapter 29, "Processing Multiple Files as Books."
- When exporting a document containing non-element markers of types other than `DOC PI` or `DOC Entity Reference`, the software generates a `<?FM MARKER [type] text?>` PI where *type* is the marker type and *text* is the marker text. For example, if you have an Index marker that is not in a marker element with the text "lionfish, ocellated", then on export, FrameMaker writes `<?FM MARKER [Index] lionfish, ocellated?>`.
- When exporting a marker of type `DOC PI`, the software generates a PI with `<?text?>`, where *text* is the marker text. (For information about how FrameMaker creates a DOC PI marker on export, see "Processing instructions" on page 321.)
- Generating comments on export
- By default, FrameMaker creates comments under the following circumstance:
- When exporting a marker of type `DOC Comment`, FrameMaker generates a comment with `<!-- text -->`, where `text` is the marker text. (For information about how FrameMaker creates a Doc Comment marker on export, see "Processing instructions" on page 321.)

Exporting special marker types as processing instructions and entity references

FrameMaker uses special marker types to store information on processing instructions, PI entities, and external data entities.

This marker type	Defines
DOC PI	a processing instruction
DOC Entity Reference	an entity reference, a PI entity, or a reference to a PI entity

Where *text* is the marker text, the software exports `DOC PI` markers as processing instructions of this form:

```
<?text?>
```

and for SGML it exports DOC Entity Reference markers as entity references of this form:

```
&text;
```

Note that you cannot use a read/write rule to change the marker type FrameMaker uses to store PI, entity reference, or PI entity information.

Exporting text insets

FrameMaker text insets generally export as external text entities. On import, if the external entity is declared in the internal DTD subset of the document instance, FrameMaker stores information about the entity on the *Entity Declarations* reference page. On export, the software translates a text inset as follows:

If the text inset references	And	FrameMaker exports it as
A text or markup file, or a markup fragment	The structure application's DTD has an entity definition that maps to the referenced file	An entity reference to the entity declaration in the structure application's DTD.
A text or markup file, or a markup fragment	The *Entity Declarations* reference page maps an entity declaration to the referenced file	An entity declaration in the internal DTD subset for the document instance, using the original entity name. It also writes a reference to that entity declaration.
A text or markup file, or a markup fragment	The *Entity Declarations* reference page *does not* map an entity declaration to the referenced file, nor is there a corresponding entity declaration in the structure application's DTD	An entity declaration in the internal DTD subset for the document instance, using a generated entity name. It also writes an entity reference to that entity declaration.
A FrameMaker text flow or any text inset not mentioned above		Markup and content in the document instance—the software does not export an entity or entity reference.

On import to FrameMaker

By default, FrameMaker interprets entity references as appropriate document objects, and saves enough information to recreate the associated entity declarations on export. The software stores information about the entity declaration in the following ways:

- The entity name is stored with the FrameMaker document object that corresponds with the original entity reference (as it was in the XML or SGM).
- If the entity was declared in the internal DTD subset of the document instance, the software stores that information on the Entity Declarations reference page of the resulting FrameMaker document.
- If the entity was declared in the external DTD subset, the software assumes it will be declared in your structure application's DTD. No entity declaration information is stored with the FrameMaker document.

In this way, the software can import and export entity references correctly. You can also modify your DTD or use read/write rules to have FrameMaker treat some entities differently.

The rest of this section describes how FrameMaker imports various entities and unknown processing instructions in the absence of rules or clients.

Internal text entities

> **Note: XML:** The XML specification doesn't treat markup data as literals in an internal text entity. For example, a reference to the following entity:
>
> ```
> <!ENTITY example " An ampersand (&) may be escaped
> numerically (&) or with a general entity
> (&). " >
> ```
>
> appears in a document as:
>
> An ampersand (&) may be escaped numerically (&) or with a general entity (&).

Internal XML or SGML text entities are text entities whose replacement text is determined solely by information in their declarations. For SGML, the parser recognizes any markup within the entity text as such. The following are examples of declarations of internal text entities—The first example is valid in XML and SGML, while the second and third examples are only valid in SGML:

```
<!ENTITY product "FrameMaker">
```

```
<!ENTITY blist "<list type=bulleted>">
```

```
<!ENTITY blist STARTTAG "list type=bulleted">
```

On import, FrameMaker tests the entity text to make sure it has no characters which can be recognized as markup in any context (within the SGML reference delimiter set).

If the entity text has no markup characters (as in the first example), FrameMaker translates it to a variable of the same name. If the document template does not define a variable of that name, FrameMaker uses the entity text as the variable's definition.

When importing SGML, if the template already contains a variable with that name, FrameMaker uses the template's variable definition. It is possible that this match between entity and variable names is unintentional, but FrameMaker does not write a message to the log file. The file imports without complaining and FrameMaker uses the template's definition for the variable. However, in all cases the information on the *Entity Declarations* reference page contains the full text from the original entity declaration.

When importing XML, FrameMaker discards any definition of the variable in the template and creates a new one from the XML document's entity declaration.

For information on structure log files, see "Log files" on page 134.

Note: SGML: The default maximum length is 240 characters when no `sgmldcl` file is included in your `sgmlapplication` file. If the replacement text is greater than 240 characters, FrameMaker imports the entity as plain text. Messages are written to the Error Log stating, "The Normalized length or literal exceeded 240; markup terminated." and "Length of name, number, or token exceeded the NAMELEN limit". If the entity text contains any markup characters (as in the last two examples above), FrameMaker imports the entity as plain text and discards all information about the entity reference.

Internal character data (CDATA) entities

Note: XML: The XML specification does not support CDATA entities—the following information pertains to SGML, only.

Internal `CDATA` entities are similar to internal SGML text entities, except that the parser treats the entity text as character data only. That is, the parser does not recognize any markup within the entity text. The following is an internal `CDATA` entity:

```
<!ENTITY tag CDATA "<TAG>">
```

FrameMaker always translates these entities as variables. In the above example, the resulting variable text would be `<TAG>`.

Internal special character data (SDATA) entities

> **Note: XML:** The XML specification does not support SDATA entities—the following information pertains to SGML, only. However, you can declare text entities that FrameMaker will convert to variables as follows:
>
> ```
> <!ENTITY date "FM variable: Current Date (Long)">
> ```

Internal SDATA entities carry specific information for the system to interpret when processing the document. You can use such entities to represent special characters or canned text. The following are examples of internal SDATA entities:

```
<!ENTITY bullet SDATA "[bullet]">

<!ENTITY copyright SDATA "[Copyrt]">

<!ENTITY date SDATA "FM variable: Current Date (Long)">
```

In the first example, the intent is to have the system interpret the entity as the bullet symbol character. In the second example, the intent is to insert canned text representing the copyright notice for the document. These two declarations are not specific to a particular system. On the other hand, the third declaration is specific to FrameMaker. In this case, the intent is to use the FrameMaker system variable Current Date (Long).

Since FrameMaker has access to the DTD containing the entity declaration, you can control the import and export of an SDATA entity by supplying an appropriate parameter literal in its entity declaration in the DTD. If you do so, you do not need to use read/write rules for this purpose. FrameMaker has a convention for interpreting certain parameter literals containing text that starts with one of these character sequences:

A literal beginning with:	Interprets the SDATA entity as:
`fm char:`	the specified FrameMaker character
`fm ref:`	the specified reference element
`fm text inset:`	a text inset importing the specified file
`fm variable:`	the named variable

For example, to translate the SDATA entity oquote to the directional open quotation mark (“) in FrameMaker, you could use this declaration in your DTD:

```
<!ENTITY oquote SDATA "FM char: \xd2">
```

With this entity declaration, when FrameMaker encounters a reference to the oquote entity as it is importing an SGML document, it replaces the reference with a directional open quotation mark. When it encounters a directional open quotation mark as it is exporting a FrameMaker document, it generates an oquote entity reference. In either case, you do not need rules to accomplish the translation. For information on how FrameMaker interprets each of these parameter literals, see the procedures in “Modifications to the default translation” on page 322.

If FrameMaker does not recognize the parameter literal, it translates the entity as a variable of the same name, with the variable text determined as for internal text entities. If the template already contains a variable with that name, FrameMaker uses the template's variable definition.

If FrameMaker automatically creates the variable definition during import, it wraps the variable text in the FmSdata character format. This character format distinguishes `SDATA` entity text from regular text and tells FrameMaker what type of entity to create on export.

External data entities

> **Note: XML:** This section discusses the use of external entities for graphic images and other types of non-SGML data. The XML specification refers to this type of entity as a nonparsed entity. According to the XML specification, any nonparsed entity must be referenced via an `entity` or `entities` attribute of an element.

External data entities, unlike any of the other types of entities, are associated with a `notation` name. A `notation` is a mechanism by which the system recognizes how to process the data referenced by the entity. While external data entities may be used for any number of purposes, they are typically used to reference text, graphics, equations, and tables residing in external files. The following is an example of the declarations needed for an external data entity that specifies a graphic file:

```
<!NOTATION cgm SYSTEM "cgm2mif">
<!ENTITY door SYSTEM "door.cgm" NDATA cgm>
```

> **Note: SGML:** The above example uses XML syntax. For SGML the notation would exclude quotes for the notation identifier as follows: `<!NOTATION cgm SYSTEM cgm2mif>`

Except for external text entities (see "External text entities," next), FrameMaker translates a direct reference to an external entity as a marker of type `DOC Entity Reference`. For this reason, we suggest graphic elements use an `entity` attribute to specify the entity reference.

For example, a reference to the above entity should be made in a graphic element's `entity` attribute. Given appropriate read/write rules to import the element as a FrameMaker graphic element, the software would create an anchored frame that imports `door.cgm` by reference. For more about importing graphic elements, see Chapter 24, "Translating Graphics and Equations."

> **Note: XML:** This use of an `entity` attribute to specify the entity reference is the only method allowed by the XML specification to reference a nonparsed entity (such as a graphic or a sound file).

If the entity declaration was made in the XML or SGML document's internal DTD subset, the software saves the entity information on the *Entity Declarations* reference page of the resulting FrameMaker document. This will be used to reconstruct the entity declaration on export.

You can use a read/write rule to drop references to all external data entities on import; see Developer Reference, page 71: external data entity reference.

If you reference an external data entity as the value of an attribute, you may want to write a rule to import it as a FrameMaker graphic. For more information about using entity references in attributes to import graphic files, see Chapter 24, "Translating Graphics and Equations."

External text entities

External text entities specify an external file which can contain either markup or character data. In effect, the contents of the external file replaces the entity reference and the parser recognizes any markup within the replacement text.

As examples:

```
<!ENTITY appendix SYSTEM "appendix.sgm">

<!ENTITY cpyright SYSTEM "cpyrt.txt">

<!ENTITY cpyright PUBLIC "//. . . public-id . . .//"
       "/a/corp/cpyrt.txt">
```

FrameMaker expands these entities and imports their replacement text (including any structure) as a text inset. If the entity declaration is specified in the internal DTD subset of the document instance, the software saves information about the entity declaration on the *Entity Declarations* reference page. This will be used to reconstruct the entity declaration on export.

In some cases, FrameMaker translates some external text entities as book components. On export, it translates these text insets and book components back into external text entities. For more information on working with books, see Chapter 29, "Processing Multiple Files as Books."

Parameter entities

Parameter entities can be referenced only within declarations. Typically they are used for such things as defining common segments of content models or common attribute definition lists. The following are examples of parameter entities:

```
<!ENTITY % subelts "(quote | emphasis | code | acronym)">

<!ENTITY % comnatt "id ID #IMPLIED
    type CDATA #IMPLIED
    security (ts, c, uc) uc">
```

FrameMaker expands these entities and imports their replacement text. On export, FrameMaker exports only the entity replacement text.

Subdocument (SUBDOC) entities

> **Note: XML:** The XML specification does not support SUBDOC entities—the following information pertains to SGML, only.

SUBDOC entities are inclusions of a complete document within another. The master document and its subdocuments each have their own document type definitions and are validated

accordingly. FrameMaker does not support SUBDOC entities. If you try to open an SGML document that uses SUBDOC entities, the software reports an error in the log file and does not open the file.

PI entities

> **Note: XML:** The XML specification does not support PI entities—the following information pertains to SGML, only.

PI entities are processing instructions in the form of entities. They are a convenient way of providing one level of indirection, allowing the user to change the processing instructions in one place if the document is moved to a different system. In addition, using a PI entity allows the processing instruction to contain the `PIC` delimiter (`>` in the reference concrete syntax). The following is an example of a PI entity:

```
<!ENTITY break PI "MYSYS: pgbrk">
```

Unless PI entities correspond to one of the forms supported by FrameMaker (to represent books and book components), FrameMaker stores them in markers of type `SGML Entity Reference` by default. In the above example, the marker text would be:

```
break
```

Processing instructions

> **Note: XML and SGML:** The XML specification defines the PI closing delimiter as `?>`, while in SGML the closing delimiter is `>`. This section uses the XML specification to illustrate PI syntax.

As stated at the beginning of this chapter, processing instructions in XML or SGML provide a way to perform system-specific actions on a markup document. By default, FrameMaker recognizes a small set of processing instructions. Those it does not recognize in a document instance it stores in a marker of type `DOC PI`. (You can change the marker type used for this purpose with a rule.) For example, if your document instance contains this processing instruction:

```
<?mypi?>
```

then FrameMaker creates a `DOC PI` marker with this marker text:

```
mypi
```

In addition to processing instructions for books, book components, and conditional text, FrameMaker recognizes another processing instruction format it uses to create non-element markers. For example, if your document instance contains this processing instruction:

```
<?FM MARKER [MyMarkerType] Some marker text here?>
```

then FrameMaker creates a `MyMarkerType` marker with this marker text:

```
Some marker text here
```

You can use a read/write rule to drop processing instructions on import or to specify a different marker type to store this information.

Comments

While importing an XML document into a FrameMaker document, FrameMaker stores comments in a marker of type `DOC Comment`.

Limitations to Writing Processing Instructions

An XML or SGML document should not have processing instructions after the end tag for the highest-level element in the document. For example, even though it is valid, the following document generates errors when you open it in FrameMaker:

```
<!DOCTYPE example [
<!ELEMENT example   ANY>
<!ELEMENT par     (#PCDATA)>
]>

<example>
<par>This is a paragraph.</par>
<?pi between pars?>
<par>Another paragraph.</par>
</example>
<?pi at end?>
```

FrameMaker aborts the import process when it encounters the `<?pi at end>` processing instruction. Because the document is complete when FrameMaker aborts the parsing, the result is a fully imported FrameMaker document. However, errors of this sort could affect structure import/export clients that rely on parsing events that occur after the document is complete.

You can write an XSLT style sheet that handles these processing instructions before the XML is parsed. You could discard them if they are not required, or, if the processing instructions are relocated to a position before the start tag for the highest level element they will be saved as markers on the *Entity Declarations* reference page.

Modifications to the default translation

> **Note: XML:** The XML specification doesn't allow `SDATA` or `CDATA` entities. Many of the following sections discuss modifications to default handling of these types of entities, and so they pertain to SGML, only.

You can handle internal `SDATA` entities in a variety of ways in FrameMaker, depending on the information they represent. In some cases, you can convert an `SDATA` entity to a FrameMaker variable or to particular characters using selected character formats. In other situations, you might convert the entity to one or more FrameMaker objects. For example, you may have an entity for a company's logo, which you represent in FrameMaker with an anchored frame.

As will be described in the following sections, you can modify the treatment of some entities either by changing the entity declaration in the DTD or by adding a rule to your read/write rules document. These modifications affect every instance of a reference to the specific entity. With read/write rules, you can either drop an entity, import it as a text inset, or import it as a specified non-element variable. You can also map the entity to an element that is stored on the SGML Utilities reference page of the resulting FrameMaker document.

You can declare SDATA entities with FrameMaker parameter literal text. A parameter literal is a way to declare in the entity itself how FrameMaker will translate the entity. If, for a single entity, you have both an entity declaration with a FrameMaker parameter literal and a rule that applies to the entity, the entity declaration takes precedence. For an example of parameter literal text, see "Translating entities as FrameMaker variables" on page 325.

For a summary of read/write rules relevant to translating entities, see Developer Reference, page 36: Entities.

Specifying the location of entity declarations

Your structure application may include files of entity declarations to use in addition to declarations in a particular XML or SGML document. You need to tell FrameMaker where it can find these declarations. You do so in the application definition. For more information, see "Application definition file" on page 132.

Renaming entities that become variables

FrameMaker translates many entities, by default, as variables. You may choose to change the name of the variable or entity. To do so, you use the following rule:

```
entity "ename" is fm variable "var";
```

where *ename* is an entity name in markup, and *var* is a FrameMaker variable. For example, if you have an entity `date`, the FrameMaker variable's name is by default `Date`. To change this behavior to use a system variable you could use this rule:

```
entity "date" is fm variable "Modification Date (Short)";
```

For information on these rules, see Developer Reference, page 61: entity and Developer Reference, page 139: is fm variable.

Translating entity references on import and export

> **Note: XML and SGML:** The entities discussed in this section are typically declared as `SDATA` entities in SGML. However, `SDATA` entities are not allowed in XML. The following sections discuss ways to make SDATA entities in SGML correspond with FrameMaker document objects. When working with XML, the preferred method is to use specific elements and map them to the document objects.

When working with XML or SGML, you can use read/write rules to convert an entity into a FrameMaker document object such as a variable or a reference element. With SGML, you can also specify a FrameMaker *parameter literal* in the entity declaration of the DTD—the parameter literal declares that FrameMaker will convert the entity into the desired document object. You must keep in mind that his approach introduces application-specific data into the DTD.

FrameMaker parameter literals and read/write rules have the same effect, but with read/write rules you keep the application-specific outside of the DTD. The parameter literals and corresponding uses of the `entity` rule are described in the following sections. In brief, they are:

For	Parameter Literal	Rule	Page
Special characters	`"fm char:"`	`entity "ename" is fm char`	325
Variables	`"fm variable:"`	`entity "ename" is fm variable`	325
Text insets	`"fm text inset:"`	`entity "ename" is fm text inset "fname"`	327
Other entities	`"fm ref:"`	`entity "ename" is fm reference element`	329

If you use the `entity` rule, you can choose to have it effective only when the software reads a markup document, not when it writes a FrameMaker document as XML or SGML. To do so, you make the `entity` rule a subrule of a highest-level `reader` rule. For example, an SGML document might use a `period` entity for some instances of the period (.) character. If you use this rule:

```
entity "period" is fm char ".";
```

then on export, all periods in the document become references to the `period` entity. To have periods remain the period character on export, use this version of the rule instead:

```
reader
   entity "period" is fm char ".";
```

For information on these rules, see Developer Reference, page 61: entity, Developer Reference, page 107: is fm char, and Developer Reference, page 151: reader.

Translating entities as FrameMaker variables

> **Note: XML:** The XML specification doesn't allow SDATA entities. To translate FrameMaker system variables in XML, you should translate them to specific XML elements. You can similarly translate user variables, or translate them as internal text entities.

You can equate an entity with a FrameMaker variable for import and export. You can do this by using either the appropriate version of the `entity` rule or the parameter literal. The parameter literal in the entity declaration has the form:

```
"FM variable: var"
```

The `entity` rule has the form:

```
entity "ename" is fm variable "var";
```

where *ename* is the entity in markup, and *var* is the desired FrameMaker variable.

For example, to have an entity named `date` correspond to the FrameMaker system variable Current Date (Long), you can add this entity declaration to your DTD:

```
<!ENTITY date SDATA "FM variable: Current Date (Long)">
```

Alternatively, if your DTD contains the entity declaration:

```
<!ENTITY date SDATA "[date]">
```

You could leave the DTD as is and add the following rule to your rules document:

```
entity "date" is fm variable "Current Date (Long)";
```

> **Important:** If you import an SGML document that inserts a FrameMaker system variable into the document, you must save the file and update variables in the document before system variables such as Creation Date (Long) appear. For information on updating variables, see the information about variables in the using manual for FrameMaker.

You can choose to have this translation only happen on import of an SGML document. For more information, see "Translating entity references on import and export," (the previous section).

For information on these rules, see Developer Reference, page 61: entity and Developer Reference, page 139: is fm variable. For more information on the treatment of FrameMaker variables, see Chapter 26, "Translating Variables and System Variable Elements."

Translating SDATA entities as special characters in FrameMaker

> **Note: XML:** The XML specification doesn't allow SDATA entities. However, you can translate special characters as internal text entities.

A common usage of `SDATA` entities is to enter special characters. In this situation, you could translate the `SDATA` entity in your SGML document directly as the appropriate character in FrameMaker, through either the appropriate version of the `entity` rule or the parameter literal.

The parameter literal in the entity declaration has the following form:

```
"fm char: code [in fmchartag]"
```

The `entity` rule has one of these forms:

```
entity "ename" is fm char "char" [in "fmchartag"];
entity "ename" is fm char code [in "fmchartag"];
```

where *ename* is an SGML entity, *code* is a character code (specified as either a 1-character string or an integer constant, using the syntax for an octal, hexadecimal, or decimal integer described in "Strings and constants" on page 277). Note that if the desired character is a digit or a white-space character, you must enter it as an integer. *char* is a one-character string and *fmchartag* is an optional FrameMaker character tag.

Without the `in` clause, this parameter literal or rule causes FrameMaker to simply insert the character specified by *code* or *char* when importing an SGML document.

Important: Special characters often require a font change. For either the rule or the parameter literal, the `in` clause tells FrameMaker to insert the indicated character using the specified character format. Note that the character format must specify a non-standard font such as Symbol or Zapf Dingbats. Otherwise, this clause cannot override the formatting for the parent element.

For example, to have an entity named `cquote` corresponding to the directional close quotation mark (”), you can add this entity declaration to your DTD:

```
<!ENTITY cquote SDATA "FM char: \xd3">
```

Alternatively, if your DTD contains an entity declaration such as:

```
<!ENTITY cquote SDATA "close-quote">
```

you can add one of the following rules to your rules document:

```
entity "cquote" is fm char "\”";
entity "cquote" is fm char "\xd3";
```

That is, your rule can specify a string with the special character or it can specify the numeric character code. The numeric code can be octal, hexadecimal, or decimal; `0xd3`, `\xd3`, and `211` all represent the same close-quote character.

For example, create a character format named Trademark, specify Symbol as the font family, and turn on the superscript property. To translate the `SDATA` entity `reg` as a superscripted version of the registered trademark character (®), you could use this declaration in your DTD:

```
<!ENTITY reg SDATA "FM char: 0xd2 in Trademark">
```

or have this rule in your rules document:

```
entity "reg" is fm char 0xd2 in "Trademark";
```

When FrameMaker encounters a reference to the `reg` entity when importing an SGML document, it replaces the reference with ® (assuming your FrameMaker template defines the Trademark character format appropriately). When exporting a document, if FrameMaker encounters ® in the Trademark character format, it generates a reference to the `reg` entity.

If you translate entities as special characters, you may want to create entity palettes to make it easier for your end users to put these special characters in a FrameMaker document. For information on how to create entity palettes, see "Facilitating entry of special characters that translate as entities" on page 333.

DTDs frequently use the entity sets defined in Annex D of the SGML Standard, often called ISO public entity sets, for providing commonly used special characters. FrameMaker includes copies of these entity sets and provides rules to handle them for your application. For information on how FrameMaker supports ISO public entities, see Developer Reference, Chapter 10, ISO Public Entities

You can choose to have this translation only happen on import of an SGML document. For more information, see "Translating entity references on import and export" on page 324.

For information on the rules used in these examples, see Developer Reference, page 61: entity and Developer Reference, page 107: is fm char. For information on the FrameMaker character set, see Developer Reference, Chapter 11, Character Set Mapping in this manual, and see the FrameMaker user's manual.

Translating SDATA entities as FrameMaker text insets

> **Note: XML:** The XML specification doesn't allow SDATA entities. However, FrameMaker translates external text entities as text insets by default. See "Translating external text entities as text insets" on page 330.

You may want to translate some SGML `SDATA` entities to text insets in the resulting document. Note that the file you use for such a text inset must be valid `SDATA`. In other words, the source file must not be an SGML file.

The source file can be of any document type that FrameMaker can filter automatically. Typically, such files will be FrameMaker documents or text files, although you can use files created by other word processing systems. If the source file is a FrameMaker document, a text inset is always the entire contents of a flow.

If the source of the text inset is a structured flow, the document that is importing it will not be valid if the flow has a highest-level element. There is one exception however, for structured flows that have `SGMLFragment` as the highest-level element. An SGML fragment is an SGML instance that contains neither an SGML declaration nor a DTD subset. Opening an SGML fragment in FrameMaker generates a structured document with a highest-level element named `SGMLFragment`. When you import such a structured flow as a text inset, FrameMaker automatically unwraps the `SGMLFragment` element so it's children will be valid in the document.

You can translate an SDATA entity to a text inset either through the appropriate version of the `entity` rule or the parameter literal.

The parameter literal in the entity declaration has one of the following forms:

```
"FM text inset: fname"
"FM text inset: fname in body flow flowname"
"FM text inset: fname in reference flow flowname"
```

The `entity` rule has one of these forms:

```
entity "ename" is fm text inset "fname";

entity "ename" is fm text inset "fname"
   in body flow "flowname";

entity "ename" is fm text inset "fname"
   in reference flow "flowname";
```

where *ename* is an SGML entity, *fname* is the document that is the source of the text inset, and *flowname* is a flow in fname. The file named by *fname* does not have to be a FrameMaker document. If it is, you can specify the flow to use. If you do not specify a flow, the main body flow of the document is used.

For example, if you have a copyright notice on the main flow of a document named `copy.fm`, you can use this entity declaration:

```
<!ENTITY copyrt SDATA
"FM text inset: copy.fm in body flow A">
```

Alternatively, you can use this rule:

```
entity "copyrt" is fm text inset "copy.fm" in body flow "A";
```

Insertion of a FrameMaker text inset in a document always inserts an end of paragraph in the document. For this reason, you should only use this rule for entities that translate to entire paragraphs or that occur only at the end of a paragraph.

By default, the structure of a text inset is retained and the text reformatted to match the FrameMaker document into which it is placed. You may choose to change this behavior. For information on how to do so, see "Changing the structure and formatting of a text inset on import" on page 331.

You can choose to have this translation only happen on import of an SGML document. For more information, see "Translating entity references on import and export" on page 324.

For information on these rules, see Developer Reference, page 61: entity and Developer Reference, page 135: is fm text inset.

Translating SDATA entities as FrameMaker reference elements

> **Note: XML:** The XML specification doesn't allow SDATA entities. This section pertains to SGML, only.

In addition to the common situations discussed in the previous sections, you may use an SDATA entity for almost any set of objects in your document. For example, an entity can be used to hold one or more anchored frames. To handle some of these cases, you can create a correspondence between an SDATA entity and an element on a *reference page* in the FrameMaker template associated with your application. To do so, use either the appropriate version of the entity rule or the parameter literal.

The parameter literal in the entity declaration or the SDATA rule has the form:

```
"fm ref: fmtag"
```

The entity rule has the form:

```
entity "ename" is fm reference element "fmtag";
```

where *fmtag* is the name of an element on a FrameMaker reference page in the associated FrameMaker template and *ename* is an SGML entity.

The *fmtag* element must occur in a flow named Reference Elements. That flow must be on a reference page with a name that starts with SGML Utilities Page—for example, SGML Utilities Page 1 or SGML Utilities Page Logos. For information on working with reference pages, see the FrameMaker using manual.

When FrameMaker encounters references to the specified entity while importing an SGML document, it copies the appropriate element from its reference page in the associated FrameMaker template. When it encounters an instance of an element associated with one of the reference pages while exporting a document, it generates an entity reference.

For example, to have an entity named logo correspond to an anchored frame with your company's logo, you can add this entity declaration to your DTD:

```
<!ENTITY logo SDATA "fm ref: Logo">
```

Alternatively, if your DTD contains an entity declaration such as:

```
<!ENTITY logo SDATA "[logo]">
```

you can add the following rule to your rules document:

```
entity "logo" is fm reference element "Logo";
```

You can choose to have this translation only happen on import of an SGML document. For more information, see "Translating entity references on import and export" on page 324.

For information on these rules, see Developer Reference, page 61: entity and Developer Reference, page 127: is fm reference element.

Translating external text entities as text insets

External text entities specify a reference to an external file that is either markup or text. By default, on import FrameMaker translates the entity as a text inset. Note that the specified file cannot be a FrameMaker document because such an entity declaration would not be valid markup.

If the referenced entity declaration was made in the DTD subset of the markup document, FrameMaker stores information about it on the Entity Declarations reference page. On export, it uses that information to generate entity declarations in the DTD subset of the markup.

Unless the source file is markup, insertion of a FrameMaker text inset in a document always inserts an end of paragraph in the document. For this reason, you should be careful about which external text entities must translate to entire paragraphs or occur only at the end of a paragraph.

FrameMaker has options to update text insets manually or automatically whenever the user opens the document containing the text insets. By default, external text entities import as text insets that update manually. Authors can select these text insets to change this setting through the user interface.

An author can manually import markup fragments into a document as text insets. When exporting to markup, FrameMaker might not have entity declarations for such user-created text insets. In the absence of entity declarations, FrameMaker generates entity declarations with entity names of `ti1`, `ti2`, etc.

Translating internal text entities as text insets

By default, on import FrameMaker translates internal text entities either as variables or as plain text. You can choose to have it instead import a text entity as a FrameMaker text inset. You may do so if the FrameMaker cannot import the entity as a variable. For example, the entity might contain too many characters, or (for SGML) it might contain SGML elements.

To import a text entity as a FrameMaker text inset, use one of these forms of the `is fm text inset` rule:

```
entity "ename" is fm text inset fname;

entity "ename"
   is fm text inset "fname" in body flow "flowname";

entity "ename"
   is fm text inset "fname" in reference flow "flowname";
```

where *`ename`* is the entity name, *`fname`* is a file path to a FrameMaker document, and *`flowname`* is a flow in the FrameMaker document. This rule translates the entity *`ename`* as a text inset. The inset is a reference to the flow named *`flowname`* in the document named *`fname`*.

For example, assume you have this entity declaration:

```
<!ENTITY copyrt
   "COPYRIGHT (c) 1500-1995 OLD GUYS CORPORATION
   The source code contained herein is our very own.
   You can't use it unless we say so.">
```

To import `copyrt` as a text inset that references a flow named Copyright on the reference page of an existing FrameMaker document named copyrt.fm, use this rule:

```
entity "copyrt"
is fm text inset "copyrt.fm" in reference flow "Copyright";
```

The source file can be of any document type that FrameMaker can filter automatically. Typically, such files will be FrameMaker documents or text files, although you can use files created by other word processing systems. Also, a FrameMaker text inset is always the entire contents of a flow or file.

Insertion of a FrameMaker text inset in a document always inserts an end of paragraph in the document. For this reason, you should use this rule for entities that translate to entire paragraphs or that occur only at the end of a paragraph.

If the source of the text inset is a structured flow, the document that is importing it will not be valid if the flow has a highest-level element. There is one exception however, for structured flows that have `SGMLFragment` as the highest-level element. An SGML fragment is an SGML instance that contains neither an SGML declaration nor a DTD subset. Opening an SGML fragment in FrameMaker generates a structured document with a highest-level element named `SGMLFragment`. When you import such a structured flow as a text inset, FrameMaker automatically unwraps the `SGMLFragment` element so it's children will be valid in the document.

By default, the structure of a text inset is retained and the text reformatted to match the FrameMaker document into which it is placed. You may choose to change this behavior. For information on how to do so, see "Changing the structure and formatting of a text inset on import," next.

You can choose to have this translation only happen on import of a markup document. For more information, see "Translating entity references on import and export" on page 324.

For information on these rules, see Developer Reference, page 61: entity and Developer Reference, page 135: is fm text inset.

Changing the structure and formatting of a text inset on import

As described in "Translating SDATA entities as FrameMaker text insets" on page 327 and "Translating internal text entities as text insets," (the previous section), you can choose to import some entities as FrameMaker text insets. When you do so, the inset is to a FrameMaker document file. The source document's structure (if any) is retained in the resulting text inset by default. The

text is also reformatted according to the format rules of the target document. You may wish to change this behavior. To do so, you use one of the following rules:

```
entity "ename" {
   is fm text inset "fname";
   reformat using target document catalogs;
}

entity "ename" {
   is fm text inset "fname";
   reformat as plain text;
}

entity "ename" {
   is fm text inset "fname";
   retain source document formatting;
}
```

where *ename* is the entity name and the `is fm text inset` rule has the arguments described in "Translating internal text entities as text insets," (the previous section).

The meaning of these rules is as follows:

- `reformat using target document catalogs`: Retain the structure of the source document and use the formatting contained in the target document. This is the default behavior.
- `reformat as plain text`: Remove the structure of the source and use the formatting contained in the target document.
- `retain source document formatting`: Remove the structure of the source and use the formatting of the source text.

In a single `entity` rule, you can use one of these rules at most. Also, you cannot use one of these rules as a subrule of an `entity` rule unless that `entity` rule also has an `is fm text inset` subrule.

You can use any of these rules at highest level, to set the default treatment of all text insets. If the source file is text or markup, these rules will have no effect. For example, for insets to FrameMaker documents, if you always want the structure stripped from the inset and you want the text to retain the formatting of the original document, you use this rule:

```
retain source document formatting;
```

For more information on these rules, see

- Developer Reference, page 61: entity
- Developer Reference, page 135: is fm text inset
- Developer Reference, page 153: reformat as plain text
- Developer Reference, page 153: reformat using target document catalogs

- Developer Reference, page 154: retain source document formatting.

Discarding external data entity references

By default, FrameMaker imports direct references to external data entities as markers of type `DOC Entity Reference`. Instead, you can choose to have it discard these references. To do so, use the following highest-level rule:

```
external data entity reference drop;
```

In SGML, the values of general entity name attributes, such as those used with graphics, are not considered entity references. This rule does not affect how FrameMaker treats general entity name attributes. For example, if a graphic element specifies an entity named `mygraphic` in its `entity` attribute, the entity `mygraphic` will not be affected.

For more information on these rules, see Developer Reference, page 71: external data entity reference and Developer Reference, page 53: drop.

Translating ISO public entities

For information on how FrameMaker supports ISO public entities, see Developer Reference, Chapter 10, ISO Public Entities

Facilitating entry of special characters that translate as entities

> **Note: XML:** The XML specification allows UNICODE in XML. This section is for SGML, only.

Your end users do not need to know the details of the entity names or definitions used to represent special characters within a FrameMaker document. They can simply insert any special character into the document and have FrameMaker automatically translate the characters back and forth to the corresponding entity references.

However, you may want to provide a special interface for inserting these characters. You can create a hypertext palette for this purpose.

You build a hypertext palette for inserting special characters with whatever layout you want. Below each active region representing a special character within the palette, you store a hypertext command to have FrameMaker insert the appropriate entity. This hypertext command has the following form:

```
message FmDispatcher insert entity ename
```

where *ename* is the entity name. When your end user clicks on a particular character within the palette, the hypertext message is sent to FrameMaker requesting the insertion of an entity with the given name. FrameMaker determines what to insert on the basis of current read/write rules and the entity declaration. This could result in inserting, for example, a special character, an `SDATA` variable, or a text inset.

Note that FrameMaker can insert the document objects you specify for these entities, whether or not there is a matching entity declared in the DTD. You should be sure to declare a matching entity for each document object. Without an entity declaration, on export the software cannot convert the object into a DOC Entity Reference.

For more on creating hypertext documents, see the information about hypertext documents in the *FrameMaker User Guide*.

Creating book components from general entities

You can break large documents across multiple files and manage those files with a single document containing general entity references in markup, and with a book file in FrameMaker. For information on this use of general entities, see Chapter 29, "Processing Multiple Files as Books."

Discarding unknown processing instructions

By default, FrameMaker imports processing instructions it does not understand as markers of type `DOC PI`. (In previous versions of FrameMaker, these were Type 11 markers.) You can choose to have it discard these processing instructions instead. To do so, use the following highest-level rule:

```
processing instruction drop;
```

For more information on these rules, see Developer Reference, page 149: processing instruction and Developer Reference, page 53: drop.

Using entities for storing graphics or equations

In markup, you often store graphics and equations in separate files and then include them in the document with general entity name attributes. For information on this use of general entities, see Chapter 24, "Translating Graphics and Equations."

22 Translating Tables

Many documents require the use of tables to organize information into cells in rows and columns. FrameMaker has a complete tool for creating tables and has a specialized element structure for representing them. Markup does not standardize the representation of tables; each DTD can represent tables differently. In practice however, many DTDs use the element and attribute declarations for tables developed for the CALS initiative, which FrameMaker directly supports.

DTDs that do not use the CALS table model can have arbitrary representations for tables. To support arbitrary table models, FrameMaker needs you to provide information in the form of read/write rules.

FrameMaker supports the CALS table model in the sense that you can import XML and SGML documents that use the CALS table model without providing read/write rules for the translation. The software automatically recognizes these elements and attributes and creates corresponding FrameMaker tables. In some situations, of course, you may choose to modify how the software creates these tables.

In this chapter

This chapter discusses how FrameMaker interprets the CALS table model: what it does by default when reading a markup document that uses CALS tables and how you can change that with read/write rules. It also discusses the default handling of arbitrary tables and how you can change that handling. It contains these sections.

Default translation

FrameMaker represents tables and table parts as elements with substructure. The CALS model also represents tables and table parts as elements with substructure. The two models are analogous, but have several differences. You need to understand these models in themselves before you can understand how the software translates between them. If you need help with these basics, read the information described in the following paragraph before going on with the sections on import and export that follow it.

For a description of how you use FrameMaker elements to represent tables, see Chapter 13, "Structure Rules for Containers, Tables, and Footnotes." For a description of the element and attribute structure of the CALS table model, see Developer Reference, Chapter 7, The CALS/OASIS Table Model

On import to FrameMaker

If your DTD does not use the CALS table model and you create an EDD from the DTD or read a markup document into FrameMaker, the software cannot identify elements that correspond to tables and table parts. In this case, you need to write read/write rules to set up the correspondence.

However, if the DTD uses the CALS table fragment, the software creates container and table part elements corresponding to these declarations. The software retains some of the attributes in the markup as attributes in FrameMaker. For other CALS attributes the software uses their values to format the table when reading markup data. The following sections provide more details of the default translation of CALS tables.

You can think of the software's default behavior In translating CALS tables as though it had a built-in set of read/write rules that identify the element and attribute structure of those tables. Developer Reference, Chapter 8, Read/Write Rules for the CALS/OASIS Table Model describes the rules that could be used to mimic this default behavior.

How CALS elements translate

If you create an EDD from a DTD that contains the CALS table declarations, the software translates the CALS `table` element to a container element and translates the `tgroup` element to a FrameMaker table element. Other elements, such as `thead` and `entry`, become table part elements of the appropriate type.

The content model for a CALS `tgroup` requires `thead` and `tfoot` to precede `tbody`. However, a FrameMaker table requires the table body to precede the table footing. When importing a `tgroup` element definition, FrameMaker can switch the order of the `tbody` and `tfoot` elements to match the order required for its tables, but only if the content model for a Tgroup is the following:

```
<!ELEMENT tgroup - O (colspec*, spanspec*, thead?, tfoot?, tbody)>
```

For any other `tgroup` content model, FrameMaker might try to switch the order of the `tbody` and `tfoot` elements upon import. However, because a `tgroup` content model can be arbitrarily complex, we cannot guarantee a valid result. For that reason, the software displays a warning message that says the content model for your table might be incorrect.

> **Important:** In a FrameMaker table, the table heading must precede the table body, and the table body must precede the table footing. If the resulting Table content model in your EDD does not specify the correct order for table parts, you must modify the EDD. Otherwise, tables will not import into your FrameMaker document.

Also, FrameMaker does not create elements corresponding to the `colspec` and `spanspec` elements. These elements exist only for their attributes. When you open a markup document, the software uses the attribute values for `colspec` and `spanspec` elements to create the corresponding table. For more information, see "How colspec and spanspec elements translate" on page 339.

If you have this SGML element structure in a DTD:

```
<!ELEMENT table - - (title?, tgroup+)>
<!ELEMENT tgroup - O (colspec*, spanspec*, thead?, tfoot?, tbody)>
```

the software creates this FrameMaker element structure in the corresponding EDD:

Element (Container): Table
General rule: Title?, Tgroup+

Element (Table): Tgroup
General rule: Thead?, Tbody, Tfoot?

Notice that the software does not create an element of type table from the CALS `table` element. The FrameMaker table model does not allow tables within tables. Because the software makes the CALS `table` element a container element, it can support multiple `tgroup` elements within the single `table` element.

In practice, many markup documents that use the CALS table model use only a single `tgroup` element within a table. In this situation, it is more natural to translate the CALS `table` element as a FrameMaker element of type table and to unwrap the `tgroup` element. If your structure application includes a FrameMaker template, the software accommodates this behavior without read/write rules.

That is, if the definition of the `Table` element in your FrameMaker template is as follows:

Element (Table): Table
General rule: Title?, Thead?, Tbody, Tfoot?

then when FrameMaker opens a markup document that uses the CALS table model, it automatically unwraps the `tgroup` element to get this effect.

How CALS attributes translate

The CALS attributes all relate in some way to the formatting and layout of the table. Most CALS attributes become formatting properties of the resulting FrameMaker table. This reflects the fact that an EDD's format rules cannot use these properties to change the layout of the table.

Four of the CALS attributes, however, relate to the formatting of the text in a table cell. By default, the attributes `align`, `char`, `charoff`, and `valign` remain as attributes in the FrameMaker representation. This allows you to use the values of these attributes in format rules for the table and its parts.

As the last step in creating a FrameMaker document from a markup document, the software applies all the format rules in the corresponding EDD. If your markup documents use the CALS attributes that remain attributes in FrameMaker, you must add format rules to your EDD for them. If you do not add format rules for these attributes, this final step of applying format rules from the EDD removes formatting specified by those attributes in the markup document. The formatting supplied by CALS attributes that correspond only to formatting properties in FrameMaker (and not to attributes) is not overridden during this step. For more information, see "Adding format rules that use CALS attributes (CALS only)" on page 348.

How colspec and spanspec elements translate

As stated earlier, the `colspec` and `spanspec` elements do not appear in the final FrameMaker table structure. These elements exist in markup as a convenience for storing formatting information about the table part in which they occur. For example, a `tgroup` element may have separate `colspec` elements to describe characteristics of each column in the `tgroup`.

There are other mechanisms for storing much of this information that are more natural for FrameMaker tables. For example, instead of specifying column and row rulings for a table with `colspec` elements, you might choose to define particular table formats for the tables that use those elements.

If your CALS tables use `colspec` and `spanspec` elements, the software correctly interprets the attribute values of those elements even though the FrameMaker table does not retain the elements themselves. For example, the `colsep` attribute determines whether or not the cells of

a table should have a ruling on the right side. Assume the `colspec` child of a `tgroup` element has this attribute set to `1`. When processing a particular `cell` element within that `tgroup`, the software checks to see if the `cell` element specifies a value for `colsep`. If it does not, then the software picks up the value from the `colspec` element of the ancestor `tgroup` and puts a ruling on the right. However, if the `cell` element has this attribute set to `0`, then that cell does not have a ruling on the right.

If you want to change how FrameMaker processes any attribute of a `colspec` or `spanspec` element, you refer to the attribute as a formatting property.

On export to markup

By default, if your EDD does not use the CALS table model, FrameMaker performs no special changes when writing the table and table part elements as elements in markup. It writes only the element and attribute structure visible in the document. However, if your EDD or FrameMaker document contains elements using CALS names, the software interprets those as CALS elements unless you specify rules to the contrary. It creates the appropriate attributes corresponding to attributes and formatting information of the table.

For example, assume you have this element structure in an EDD and you save it as an SGML DTD:

Element (Container): Table
General rule: Title?, Tgroup+

Element (Table): Tgroup
General rule: Thead?, Tbody, Tfoot?

FrameMaker creates these corresponding element definitions:

```
<!ELEMENT Table - - (Title?, Tgroup+)>
<!ELEMENT Tgroup - - (Thead?, Tbody, Tfoot')>
```

If instead you have no element named `Tgroup`, and have this definition for `Table`:

Element (Table): Table
General rule: Title?, Thead?, Tbody, Tfoo?

the software creates the same element definitions as before. That is, it automatically creates the required `tgroup` element, even though that element was not present in the FrameMaker structure.

In addition to the element structure, the software creates the CALS attributes for the table's formatting properties and for any defined attributes.

Assume you have a document with CALS names for table elements. When you save it as markup, FrameMaker does not create `spanspec` elements; it puts the equivalent information in the `colspec` and `entry` elements for your table.

Modifications to the default translation

The following sections describe some of the ways you can modify the default translation of tables. If you're translating CALS tables, the default behavior is probably mostly what you want. You'll use rules to make minor modifications such as renaming elements or changing the ruling style for a table. If you've got a different table model, you'll have to use rules more extensively to create the correspondence between markup and FrameMaker element structures.

For additional ways to modify the translation of tables, see the cross-references at the end of each section. For a summary of read/write rules relevant to translating tables, see Developer Reference, page 40: Tables.

Specifying a table element in Schema

In order for FrameMaker to translate a Schema element to a table upon import of an XML file associated with a Schema declaration, you must use element definitions and rules such as the following:

Schema element definition for a table

```
<xsd:element name="Table1"><xsd:complexType><xsd:sequence>
<xsd:element name="Title" type="xsd:string" minOccurs="0"/>
<xsd:element name="THead" type="tHd" minOccurs="0"/>
<xsd:element name="TBody" type="tBdy"/>
<xsd:element name="TFoot" type="tFt" minOccurs="0"/>
</xsd:sequence></xsd:complexType></xsd:element>

<xsd:element name="Row"><xsd:complexType><xsd:sequence>
<xsd:element name="entry" type="xsd:string" minOccurs="0"
maxOccurs="unbounded"/>
</xsd:sequence></xsd:complexType></xsd:element>

<xsd:complexType name="tBdy"><xsd:sequence>
<xsd:element ref="Row" minOccurs="1" maxOccurs="unbounded"/>
</xsd:sequence></xsd:complexType>

<xsd:complexType name="tHd"><xsd:sequence>
<xsd:element ref="Row" minOccurs="1" maxOccurs="unbounded"/>
</xsd:sequence></xsd:complexType>

<xsd:complexType name="tFt"><xsd:sequence>
<xsd:element ref="Row" minOccurs="1" maxOccurs="unbounded"/>
</xsd:sequence></xsd:complexType>
```

Read/write rule for a table

```
element "Table1" {
is fm table element "Table1";
fm property columns value is "4";
}
element "Title" is fm table title element;
```

For information on the rules used in these examples, see Developer Reference, page 56: element, Developer Reference, page 133: is fm table element, and Developer Reference, page 134: is fm table part element.

Formatting properties for tables

There are many formatting properties associated with FrameMaker tables that you might want to reference in your read/write rules. These properties are for describing general table properties such as the number of columns, straddles, or formatting properties of table cells. Some of the read/write rules that follow apply to specific formatting properties.

In translating a CALS table, the software associates some of these formatting properties with appropriate CALS table attributes by default.For example the number of columns in a CALS table becomes the `cols` attribute by default. For any other table representation, you need to write a rule to associate the appropriate formatting property with an attribute.

Properties for general table formatting

The formatting properties listed in the following table describe general table properties. More information about individual properties follows the table.

FrameMaker Property	For elements of type	CALS attribute
`column ruling`	table, cell, colspec, spanspec	`colsep`
`column width`	colspec	`colwidth`
`column widths`	table	—
`columns`	table	`cols`
`maximum height`	row	—
`minimum height`	row	—
`page wide`	table	`pgwide`
`rotate`	cell	`rotate`
`row type`	row	—
`row ruling`	table, cell, row, colspec, spanspec	`rowsep`
`table border ruling`	table	`frame`
`table format`	table	`tabstyle, tgroupstyle`

`Column ruling`: whether the specified column should having rulings on its right side. This attribute says only whether or not to have the specified ruling; it does not specify what the ruling should look like.

`Column width`: width of a single column.

`Column widths`: width of successive columns in the table. Each value is either an absolute width or a width proportional to the size of the entire table. If proportional widths are used, the `pgwide` attribute determines the table width.

For example, to specify that the first two columns are each one-quarter the size of the table and the third column is half the size of the table, you could write a rule to specify your column widths as `"25* 25* 50*"`. Valid units and abbreviations for the `column width` formatting property are:

Unit	Abbreviation
centimeter	cm
cicero	cc
didot	dd
inch	in
millimeter	mm
pica	pc (or pi)
point	pt

In FrameMaker dialog boxes, the inch unit can be represented by a double quotation mark (") as well as the `in` abbreviation. However, this convention is not supported for the `column width` formatting property. For example, `2"` is invalid as an alternative to `2in`.

`Columns`: number of columns in the table.

> **Important:** If you plan to translate markup documents to FrameMaker and your table declaration in markup does not include an attribute that corresponds to the number of columns, you *must* use the `fm property value is` rule to set a value for this property.

`Maximum height`: maximum height of a row in a table.

`Minimum height`: minimum height of a row in the table.

`Page wide`: relevant only to tables whose columns use proportional widths. In this case, the attribute indicates whether the entire table should be the width of the column in which it sits or of its text frame. If the value is unspecified or zero, the table is the width of the column; otherwise, it is the width of the enclosing text frame.

`Rotate`: how much to rotate the contents of a cell. The CALS model restricts the value of this attribute to a boolean, where 1 indicates a rotation of 90 degrees clockwise. FrameMaker extends the possible values to allow rotations of 0, 90, 180, and 270 degrees. On export, if the attribute

has a positive value other than 180 or 270, the software interprets the value as 90 to be consistent with the CALS model.

`Row type`: whether the associated table row is a heading, footing, or body row, or the associated table cell occurs in a row of that type.

`Row ruling`: whether the cells of a row should having rulings on their bottom sides. This attribute says only whether or not to have the specified ruling; it does not specify what the ruling should look like.

`Table border ruling`: whether or not there is a ruling around the entire table. The possible values are `all`, `top`, `bottom`, `top and bottom`, `sides`, and `none`. This attribute says only whether or not to have the specified ruling; it does not specify what the ruling should look like. (You can use a rule to set the look of the ruling.)

Formatting properties for straddles

FrameMaker provides multiple ways to specify straddling information, some of which correspond to the different ways available in the CALS model.

The formatting properties listed in the following table describe straddling properties of tables. More information about individual properties follows the table.

FrameMaker Property	For elements of type	CALS Attribute
`column name`	cell, colspec	`colname`
`column number`	cell, colspec	`colnum`
`end column name`	cell, spanspec	`nameend`
`horizontal straddle`	cell	—
`more rows`	cell	`morerows`
`span name`	cell, spanspec	`spanname`
`start column name`	cell, spanspec	`namest`
`vertical straddle`	cell	—

`Column name`: associates a name with a given column (specified with `colnum`).

`Column number`: specifies the number of the column named with `colname`. Columns are numbered from left to right starting at 1. Also used to specify the column in which a cell appears.

`End column name`: specifies the name of a column that ends a straddle.

`Horizontal straddle`: how many columns this straddled cell spans.

`More rows`: specifies row straddling for a cell, so that the total number of rows the cell occupies is `morerows+1`.

`Span name`: names a CALS `spanspec` element to allow an `entry` element to reference it.

`Start column name`: specifies the name of a column that begins a straddle.

`Vertical straddle`: how many rows this straddled cell spans.

> **Important:** If you are not using the CALS table model and you want to specify straddling information on export to markup, you should use only the `horizontal straddle` and `vertical straddle` properties. The other straddle properties exist to support the alternatives available in the CALS model.

For information on how to use the formatting properties for straddles with the CALS table model, see Developer Reference, page 219: Attribute structure.

Formatting properties for cell paragraph formatting

The formatting properties listed in the following table describe characteristics of a cell's paragraph format and are defined only for CALS colspecs and spanspecs. You can use format rules to map these CALS attribute values to the associated paragraph format properties.

FrameMaker Property	CALS Attribute
cell alignment character	`char`
cell alignment offset	`charoff`
cell alignment type	`align`
vertical alignment	`valign`

Note that these CALS attributes are retained in the FrameMaker document. This means you must write format rules to use these attributes to format a table, but it allows you to control the paragraph properties explicitly. For more information, see "Adding format rules that use CALS attributes (CALS only)" on page 348.

These properties only exist for `colspec` and `spanspec` elements. Other elements must use attributes to refer to this information.

`Cell alignment character`: relevant only if the `align` attribute is `char`. Determines the character on which text aligns.

`Cell alignment offset`: relevant only if the `align` attribute is `char`. Determines the location of the tab stop.

`Cell alignment type`: determines horizontal justification within a cell. Its legal values are `left`, `right`, `center`, `justify`, and `char`. If this attribute is set to `char`, the cell acts as though its autonumber were a tab character and is aligned around a character specified with the `char` attribute.

`Vertical alignment`: determines vertical positioning of the text in a cell.

Identifying and renaming table parts

If your DTD uses a table model other than the CALS model, you must identify the individual parts of the table such as the title and body rows. Additionally, you may choose to rename these

elements. If you're using the CALS table model, you don't need to identify the table parts, but you may want to use different element names in FrameMaker.

FrameMaker provides several rules for these purposes. If you include these rules, the software uses them to determine what elements to create in the EDD and to translate instances of table elements between FrameMaker and markup.

The rules for identifying and renaming table parts are as follows:

```
element "gi" is fm table element ["fmtag"];
element "gi" is fm table title element ["fmtag"];
element "gi" is fm table heading element ["fmtag"];
element "gi" is fm table body element ["fmtag"];
element "gi" is fm table footing element ["fmtag"];
element "gi" is fm table row element ["fmtag"];
element "gi" is fm table cell element ["fmtag"];
```

where *gi* is an generic identifier and *fmtag* is a FrameMaker element tag. The optional *fmtag* argument allows the element to be renamed. If *fmtag* is not specified, the name remains the same in the two representations.

If you identify a FrameMaker element as a table part, your document cannot use that element outside a table. For example, assume you have the rule:

```
element "entry" is fm table cell element "Cell";
```

The corresponding EDD contains an element `Cell`. Documents created with this EDD should not place the `Cell` element anywhere other than as a table cell. If they do so, the resultant document will be invalid.

If your DTD has an element you identify as a table element and another element you identify as a table part such as a table cell, it may not also include other elements that correspond to the intervening table parts. For information on how FrameMaker handles this, see "Omitting explicit representation of table parts" on page 351.

For information on these rules, see Developer Reference, page 56: element, Developer Reference, page 133: is fm table element, and Developer Reference, page 134: is fm table part element.

Representing FrameMaker table properties as attributes in markup

If you use the CALS table model, FrameMaker automatically represents some table properties as attributes by default. If you use another table model, FrameMaker does not recognize any attributes as table properties.

If you have a variant of the CALS model, you can choose different names for these attributes. If you have any other table model and you want to map attributes to formatting properties, you can do so. To perform either of these tasks, use this version of the `attribute` rule:

```
attribute "attr" is fm property prop;
```

If your DTD uses a particular attribute name for the same purpose within multiple elements, you may want to use this rule as a highest-level rule to set a default. For example, assume you have four different markup elements representing different types of tables. All four elements use the attribute `numc` to represent the number of columns in the table. In this case, you would use the rule:

```
attribute "numc" is fm property columns;
```

With this rule, the software interprets the attribute `numc` as the number of columns in a table for all elements in which it occurs. If you use the same attribute for another purpose in another element, you must write a local `attribute` rule to handle it appropriately.

Alternatively, you may have only one element, `tab`, representing a table. In this case, you should restrict the association of `numc` with the `columns` property only to that element using this rule:

```
element "tab" {
   is fm table element;
   attribute "numc" is fm property columns;
}
```

With this rule, other elements can use the `numc` attribute for different purposes.

In both of these examples, the software doesn't create structure that corresponds to the `numc` attribute, but uses the attribute to read or write the appropriate information from instances of the table element.

For information on the available table formatting properties and on the CALS attributes that map to formatting properties, see "Formatting properties for tables" on page 342. For information on the rules used in these examples, see Developer Reference, page 46: attribute, Developer Reference, page 56: element, Developer Reference, page 133: is fm table element, and Developer Reference, page 116: is fm property.

Representing FrameMaker table properties implicitly in markup

A table formatting property in FrameMaker may always have the same value when applied to a particular markup element representing tables in your application. If you don't have an attribute in markup for this property, you cannot assume that the software will appropriately format the table on the basis of the element. In that case you can use the following rule to specify the value explicitly:

```
fm property prop {
   value is "propval";
}
```

where *`prop`* is the name of the property and *`propval`* is the property value.

This rule tells the software to assign a particular value to one of the formatting properties on import and not to write an attribute with the value on export.

The `fm property` rule can be used at highest level to set a default or within an `element` rule to be restricted to a single element.

For example, you may have a markup element `tab2` that represents tables with a 1-inch column followed by a 2-inch column. The `tab2` element does not use an attribute for this information but you can translate `tab2` to a FrameMaker table element as follows:

```
element "tab2" {
   is fm table element "Two Table";
   fm property columns value is "2";
   fm property column widths value is "1in 2in";
}
```

In this case, when the software encounters a start-tag for a `tab2` element on import, it creates a table element named `Two Table`. The associated table has two columns, 1 and 2 inches wide. When it encounters a `Two Table` table element on export, it writes a start-tag for a `tab2` element. It does not write attributes for the number or widths of its columns.

> **Important:** If your markup table declarations do not include an attribute that corresponds to the `columns` property and you plan to open an associated markup document in FrameMaker, you *must* use this rule to supply a value for the number of columns in the table.

For information on the rules used in this example, see Developer Reference, page 56: element, Developer Reference, page 133: is fm table element, and Developer Reference, page 80: fm property.

Adding format rules that use CALS attributes (CALS only)

Four attributes of the CALS table model remain attributes when a markup document is read into FrameMaker. These attributes, `align`, `valign`, `char`, and `charoff`, provide information on the formatting of text within table cells. In FrameMaker, this formatting is controlled by format rules in the EDD. Keeping the attributes in the FrameMaker representation allows you to control this information explicitly.

For example, if you wanted to make use of values of the `valign` attribute, you could have this definition for a table cell:

Element (Table Cell): Entry
General rule: <TEXT>
Text format rules
1. If context is: [Valign = "Top"]
Table cell properties
Vertical alignment: Top
Else, if context is: [Valign = "Middle"]
Table cell properties
Vertical alignment: Middle
Else
Table cell properties
Vertical alignment: Bottom

For information on writing format rules, see Chapter 15, "Text Format Rules for Containers, Tables, and Footnotes."

Working with colspecs and spanspecs (CALS only)

You may use a table model that is essentially the CALS table model but you choose to rename some of the elements, even in the DTD. If you do not use the default names for the elements that represent `colspecs` and `spanspecs`, you need to let the software know which elements to use.

The rules for identifying colspecs and spanspecs are:

```
element "gi" is fm colspec;
element "gi" is fm spanspec;
```

where *gi* is a generic identifier.

As usual with `colspec` and `spanspec` elements, the named markup element does not become an element in FrameMaker when you use these rules. Rather, its attributes are used in the creation of the table element.

For more information on these rules, see Developer Reference, page 56: element.

Specifying which part of a table a row or cell occurs in

In markup your table may not have elements for rows or particular table parts such as the heading or body. Instead, the element type of a table row or cell may determine what part of the table the element goes in. For example, an element named `hrow` might only be used for a row in the heading of a table. You might not have a separate element for the heading.

If a table row does not occur inside a specified table part, the software assumes the row belongs in the table body by default. You can change this behavior using the following rule:

```
element "gi" {
   is fm table row_or_cell element;
   fm property row type value is "part";
}
```

where *gi* is a generic identifier; *row_or_cell* is one of the keywords `row` or `cell`; and *part* is one of `Heading`, `Body`, or `Footing`.

For an example of the use of these rules, see "Omitting explicit representation of table parts" on page 351.

For more information on these rules, see Developer Reference, page 56: element, Developer Reference, page 134: is fm table part element, and Developer Reference, page 80: fm property.

Specifying which column a table cell occurs in

In markup your table may not have elements for rows or particular table parts such as the heading or body. As indicated in "Specifying which part of a table a row or cell occurs in," (the previous section), the element type of a table row or cell may determine what part of the table the element goes in. In this case, you may also need to give FrameMaker other information such as which column a table cell should be in and the fact that an element of this type indicates the start of a new row.

To tell FrameMaker which column a table cell goes in, you set the `column number` property on that element, using this rule:

```
element "gi" {
   is fm table cell element;
   fm property column number value is "n";
}
```

where *gi* is a generic identifier and *n* is an integer indicating the table column. Table columns are numbered starting with 1.

If you tell FrameMaker to put a table cell element in a particular column and there is already content in that column, FrameMaker creates a new table row to hold the element. For example, if you specify that the `term` element always occurs in column 1 and there are no vertical straddles in that column, FrameMaker creates a new table row whenever it encounters that element. For an example of this use of the column number property, see "Omitting explicit representation of table parts," next.

Your tables can have vertical straddles: the element structure of such a table reflects a vertical straddle by not having table cell elements in the straddled rows. FrameMaker cannot tell the difference between a table cell element missing because of a straddle and a table cell element missing because that cell has not yet been filled in. For this reason, it may not be enough to tell FrameMaker that an element belongs in the first column to force it to start a new row for the

element. If your table can have vertical straddles and you want a particular element always to occur in a new row, you should use these rules:

```
element "gi" {
   is fm table cell element;
   fm property column number value is "n";
   reader start new row;
}
```

where *gi* is a generic identifier and *n* is an integer indicating the table column. Table columns are numbered starting with 1.

For an example of this use of these rules, see "Creating parts of a table even when those parts have no content" on page 354.

For more information on these rules, see:

- Developer Reference, page 56: element
- Developer Reference, page 134: is fm table part element
- Developer Reference, page 80: fm property
- Developer Reference, page 157: start new row.

Omitting explicit representation of table parts

In markup you have the flexibility of thinking of tables as either inherently structured with rows and cells or as simply a formatting choice for some completely different element structure. In FrameMaker, though, elements must be table (and table part) elements if they are to be formatted as tables (and table parts). To facilitate formatting markup element structures as tables even when those elements do not reflect table structure, FrameMaker will create missing pieces of table structure for you.

For example, assume you have a table of terms and their definitions. Your markup can be as simple as this:

```
<glossary>
   <label>Term</label><label>Definition</label>
   <term>Mouse</term>
   <defn>A small animal</defn>
   <term>Cat</term>
   <defn>A bigger animal</defn>
   <term>Elephant</term>
   <defn>An even bigger animal</defn>
</glossary>
```

This structure does nothing to identify the rows and columns of the table. Nevertheless, you can have this structure become a table in FrameMaker by specifying mappings for the existing

elements and having definitions for the missing table parts in your EDD, even though the missing table parts won't appear in your markup document.

Assume your EDD has these definitions:

Element (Table): Glossary
General rule: GlossaryHead, GlossaryBody
Text format rules
1. In all contexts.
Use paragraph format: TableCell

Element (Table Heading): GlossaryHead
General rule: GlossaryHeadRow
Text format rules
1. In all contexts.
Default font properties
Weight: Bold

Element (Table Row): GlossaryHeadRow
General rule: Label, Label

Element (Table Cell): Label
General rule: <TEXT>

Element (Table Body): GlossaryBody
General rule: GlossaryRow+

Element (Table Row): GlossaryRow
General rule: Term, Definition

Element (Table Cell): Term
General rule: <TEXT>
Text format rules
1. In all contexts.
Default font properties
Angle: Italic

Element (Table Cell): Definition
General rule: <TEXT>

And you have the following rules:

```
element "glossary" {
   is fm table element;
   fm property columns value is "2";
}

element "label" {
   is fm table cell element;
   fm property row type value is "Heading";
}
```

```
element "term" {
   is fm table cell element;
   fm property column number value is "1";
   fm property row type value is "Body";
}

element "defn" is fm table cell element "Definition";

fm element "GlossaryHead" unwrap;
fm element "GlossaryBody" unwrap;
fm element "GlossaryHeadRow" unwrap;
fm element "GlossaryRow" unwrap;
```

With these rules, the software does the following when you import the markup document containing the `glossary` example:

1. When it encounters the start-tag for `glossary`, the software creates a new 2-column `Glossary` table element. Since `glossary` does not have an attribute corresponding to the `columns` property, you had to set this value explicitly.
2. When it encounters the start-tag for the first `label` element, the software notes that this element is a table cell element and that it belongs in a table heading. However, there is not yet a table heading or row in which to put it. The software checks the definition of `Glossary` and creates the intervening `GlossaryHead` and `GlossaryHeadRow` elements. It then adds the `label` element as the first cell in the `GlossaryHeadRow`.
3. When it encounters the second `label` element, the software fills in the second column of the current heading row.
4. When it encounters the first `term` element, the software notes that this table cell element belongs in a table body. It finishes creating the heading row, checks the `Glossary` definition again, and creates the intervening `GlossaryBody` and `GlossaryRow` elements. It then adds the `term` element as the first cell in the `GlossaryRow`.
5. When it encounters the first `defn` element, the software notes that this is another table cell element and that nothing special has been said about it. The software therefore adds this element as the second cell in the first row of the table body.
6. When it encounters the second `term` element, the software notes that this table cell element is supposed to be in the first column of the table. Accordingly, it creates a new table row and puts the `term` element in its first column.
7. And so on.

In this way, the markup structure becomes the following table in a FrameMaker document:

Term	**Definition**
Mouse	A small animal
Cat	A bigger animal
Elephant	An even bigger animal

When this FrameMaker table is written as markup, the `fm element` rules specify that the intervening levels of table structure are not written out. Consequently, the resulting markup looks as it originally did.

For more information on the `row type` formatting property, see "Specifying which part of a table a row or cell occurs in," (the previous section).

For more information on these rules, see

- Developer Reference, page 56: element
- Developer Reference, page 77: fm element
- Developer Reference, page 133: is fm table element
- Developer Reference, page 134: is fm table part element
- Developer Reference, page 80: fm property
- Developer Reference, page 160: unwrap.

Creating parts of a table even when those parts have no content

When FrameMaker creates a table while importing a markup document, by default it creates only the table parts that have content. For example, your table definition may state that a table has a title but if the table instance in markup does not include a title, then the software does not create a title for the table.

To have the software create a special table part even if it has no content, use one of these rules:

```
reader insert table title element "fmtag";
reader insert table heading element "fmtag";
reader insert table footing element "fmtag";
```

where *fmtag* is a FrameMaker element tag.

For example, assume you have a variant of the example used in the previous section. Instead of having to specify the labels for the heading rows explicitly, the markup representation assumes that the software will put in the appropriate labels. In this case, the markup for the table is:

```
<glossary>
   <term>Mouse</term>
   <defn>A small animal</defn>
   <term>Cat</term>
   <defn>A bigger animal</defn>
   <term>Elephant</term>
   <defn>An even bigger animal</defn>
</glossary>
```

However, the intent is to have the table appear as before. In this case, your EDD definitions are similar to before. In place of this definition:

Element (Table Cell): Label
General rule: <TEXT>

you now have this definition:

Element (Table Cell): Label
General rule: <EMPTY>
Text format rules
1. **If context is:** {first}
Numbering properties
Autonumber format: Term
Else
Numbering properties
Autonumber format: Definition

This definition says that if a `Label` element occurs as the first child of its parent (the first column in a row), then the word "Term" appears at the beginning of its (otherwise empty) text. The word "Definition" appears in all other columns.

With the modified definitions, you use modified rules. Since `label` is no longer part of the markup element structure, you replace these rules:

```
element "glossary" {
   is fm table element;
   fm property columns value is "2";
}

element "label" {
   is fm table cell element;
   fm property row type value is "Heading";
}

fm element "GlossaryHead" unwrap;
fm element "GlossaryHeadRow" unwrap;
```

with these rules:

```
element "glossary" {
   is fm table element;
   fm property columns value is "2";
   reader insert table heading element "GlossaryHead";
}

fm element "GlossaryHead" drop;
```

For more information on these rules, see:

- Developer Reference, page 56: element

- Developer Reference, page 77: fm element
- Developer Reference, page 133: is fm table element
- Developer Reference, page 134: is fm table part element
- Developer Reference, page 116: is fm property
- Developer Reference, page 151: reader
- Developer Reference, page 101: insert table part element
- Developer Reference, page 53: drop
- Developer Reference, page 160: unwrap.

Specifying the ruling style for a table

The `frame`, `colsep`, and `rowsep` CALS attributes determine whether or not a ruling should be applied to the appropriate part of a table. These attributes are all booleans; that is, they specify simply whether or not a ruling should be applied. FrameMaker supports several ruling styles. To specify the ruling style for the entire table, you can use the following rule:

```
reader table ruling style is "style";
```

where *`style`* is the name of a ruling style. This rule sets the ruling style for all tables. For example, to set the outer borders of all tables that have outer borders to use a thick ruling style, you would use this rule:

```
reader table ruling style is "Thick";
```

A ruling set in this manner is considered as custom ruling and shading by the software.

For information on formatting tables, see the FrameMaker user's manual. For more information on these rules, see Developer Reference, page 151: reader and Developer Reference, page 159: table ruling style is of this manual.

Exporting table widths proportionally

When you export a table, the software writes the width of the columns as absolute measurements by default. If you want to use proportional widths instead, you can use these rules:

```
writer use proportional widths;
writer proportional width resolution is "value";
```

where *`value`* is an integer. The proportions of all columns in a table add to *`value`*. If you do not specify the `proportional width resolution` rule, the default is 100; that is, the proportional widths of all columns add to 100. If you do not specify the `use proportional widths` rule, the software writes absolute widths for all tables.

For example, assume you use the CALS table model and you've added these rules:

```
writer use proportional widths;
writer proportional width resolution is "4";
```

If you export a FrameMaker document containing a 3-column table whose columns are, respectively, 1 in, 1 in, and 2 in wide, the software writes the following `colspec` start-tags for the table:

```
<colspec colname = "1" colnum = "1" colsep = "0" colwidth = "1*">
<colspec colname = "2" colnum = "2" colsep = "0" colwidth = "1*">
<colspec colname = "3" colnum = "3" colsep = "0" colwidth = "2*">
```

If the table's actual column widths do not add to the resolution, FrameMaker rounds the values as necessary. For example, if the above table columns were actually 0.75 in, 1.2 in, and 2.2 in, the software would still write the same `colspec` start-tag.

You can use the `use proportional widths` rule with any attribute that has been associated with the `column widths` property, not just the CALS attributes.

For more information on these rules, see Developer Reference, page 166: writer, Developer Reference, page 162: use proportional widths, and Developer Reference, page 150: proportional width resolution is.

Creating vertical straddles

FrameMaker provides two rules for you to use if your table structure defines rows that are always straddled. In an `element` rule for a table cell, you can use this rule:

```
reader start vertical straddle "name";
```

In an `element` rule for a table row, you can use this rule:

```
reader end vertical straddle "name1 . . . nameN";
reader end vertical straddle "name1 . . . nameN" before this row;
```

where *name* identifies the element that starts a vertical straddle, and each $name_i$ is a previously named straddle that ends with this element.

For example, a book on marine life might have tables of fish, including the general category and several subtypes of that category, with locations in which you can find the subtypes. Here's an example of such a table:

General type	Subtype	Location
Lionfish	Clearfin	Egypt
	Ocellated	French Polynesia
	Spotfin	Papua New Guinea
Eel	Blue ribbon	Fiji
	Moray	Pretty much everywhere

In your markup representation, you may choose to use an element structure such as the following:

```
<range>
   <type>Lionfish</type>
      <subtype>Clearfin</subtype><loc>Egypt</loc>
      <subtype>Ocellated</subtype><loc>French Polynesia</loc>
      <subtype>Spotfin</subtype><loc>Papua New Guinea</loc>
   <type>Eel</type>
      <subtype>Blue ribbon</subtype><loc>Fiji</loc>
      <subtype>Moray</subtype><loc>Pretty much everywhere</loc>
</range>
```

The markup representation assumes that the formatting software knows that this should become a 3-column table and that a cell containing a `type` element straddles all the rows that contain the following `subtype` elements. The straddle ends just before the next `type` element.

To produce the table above, include these definitions in your EDD:

Element (Table): Range
General rule: RangeHead, RangeBody
Text format rules
1. In all contexts.
Use paragraph format: TableCell

Element (Table Heading): RangeHead
General rule: RangeHeadRow
Text format rules
1. In all contexts.
Default font properties
Weight: Bold

Element (Table Row): RangeHeadRow
General rule: Label, Label, Label

Element (Table Cell): Label
General rule: <EMPTY>
Text format rules
1. If context is: {first}
Numbering properties
Autonumber format: General type
Else, if context is: {last}
Numbering properties
Autonumber format: Location
Else
Numbering properties
Autonumber format: Subtype

Element (Table Body): RangeBody
General rule: RangeRow+

Element (Table Row): RangeRow
General rule: Type?, Subtype, Location

Element (Table Cell): Type
General rule: <TEXT>

Element (Table Cell): Subtype
General rule: <TEXT>

Element (Table Cell): Location
General rule: <TEXT>

And the following rules in your read/write rules document:

```
element "range" {
   is fm table element;
   fm property columns value is "3";
   reader insert table heading element "RangeHead";
}

element "type" {
   is fm table cell element;
   fm property column number value is "1";
   fm property row type value is "Body";
   reader start vertical straddle "Type";
   reader end vertical straddle "Type" before this row;
   reader start new row;
}

element "subtype" {
   is fm table cell element;
   fm property column number value is "2";
}
```

```
element "loc" {
   is fm table cell element "Location";
   fm property column number value is "3";
}

fm element "RangeHead" drop;
fm element "RangeBody" unwrap;
fm element "RangeRow" unwrap;
```

This example builds on information in "Specifying which part of a table a row or cell occurs in" on page 349, "Omitting explicit representation of table parts" on page 351, and "Creating parts of a table even when those parts have no content" on page 354.

For more information on these rules, see

- Developer Reference, page 56: element
- Developer Reference, page 77: fm element
- Developer Reference, page 133: is fm table element
- Developer Reference, page 134: is fm table part element
- Developer Reference, page 80: fm property
- Developer Reference, page 151: reader
- Developer Reference, page 101: insert table part element
- Developer Reference, page 59: end vertical straddle
- Developer Reference, page 158: start vertical straddle
- Developer Reference, page 157: start new row
- Developer Reference, page 53: drop
- Developer Reference, page 160: unwrap

Using a table to format an element as a boxed set of paragraphs

The formatting associated with your documents may require that the paragraphs in an element appear in a completely or partially boxed area. In SGML, assume you have the following element declaration:

```
<!element note - - ((#PCDATA | emphasis | code)+)>
```

To format this element as a boxed paragraph in FrameMaker, you use a one-cell table with appropriately defined ruling properties. The corresponding EDD looks as follows:

Element (Table): Note
General rule: NoteBody
Initial table format
1. In all contexts.
Table Format: NoteTable
Text format rules
1. In all contexts.
Use paragraph format: note

Element (Table Body): NoteBody
General rule: NoteRow

Element (Table Row): NoteRow
General rule: NoteCell

Element (Table Cell): NoteCell
General rule: (<TEXT> | Emphasis | Code)+

The `NoteTable` table format can specify ruling that boxes the paragraph. For example, the EDD for this manual uses this technique to format important information such as the following:

> **Note:** This boxed paragraph is implemented as a one-cell table.

Using read/write rules, to you can translate a single markup element that needs to be formatted as one or more boxed paragraphs into a one-cell table in FrameMaker. To create the element definitions above, use the following rules:

```
element "note" {
   is fm table element;
   fm property columns value is "1";
}
fm element "NoteBody" unwrap;
fm element "NoteRow" unwrap;
fm element "NoteCell" unwrap;
```

With these rules, FrameMaker creates the `Note` table element when it encounters a `note` element while importing a markup document. The next thing it encounters is text to go into the table, but text can't directly be put into a table—it must go in a table cell—so the intervening parts of the table need to be supplied. Since the markup doesn't specify any child elements, the software uses the structure specified in the EDD for a `Note` element. That is, it creates `NoteBody`, `NoteRow`, and `NoteCell` elements and places the text in the `NoteCell` element.

On export, the software unwraps the `NoteBody`, `NoteRow`, and `NoteCell` elements and writes out only the single `note` element.

For information on these rules, see:

- Developer Reference, page 56: element
- Developer Reference, page 77: fm element
- Developer Reference, page 133: is fm table element
- Developer Reference, page 80: fm property
- Developer Reference, page 160: unwrap.

For information on creating table formats, see the FrameMaker User Guide.

Creating tables inside other tables

The FrameMaker table model does not allow you to place a table directly inside another table. To put a table inside another table, you must put the inner table inside an anchored frame. Note that the table inside the anchored frame is in a different text flow than the outer table. For this reason, if your EDD allows for this situation and you need to export such tables to markup, you'll need to write a structure API client to do so.

For information on writing structure API clients, see the*Structure Import/Export API Programmer's Guide*.

Rotating tables on the page

In FrameMaker, you cannot directly specify that an entire table be rotated on the page. If you need to rotate tables, the *Using FrameMaker* manual suggests two approaches: you can have the table format start at the top of a page and apply a rotated master page. Or you can put the table inside an anchored frame and rotate the table inside the frame.

The first method does not require special processing on export to markup. Since the second method places the table in a separate text flow, you must write a structure API client to perform export to markup if you use it.

For information on writing structure API clients, see the*Structure Import/Export API Programmer's Guide*.

23 Translating Graphics and Equations

FrameMaker provides a set of tools for creating graphics or equations. It also provides tools for importing graphic objects created with another software package into a FrameMaker document. Markup, on the other hand, does not standardize the representation of either graphics or equations; each DTD can treat them differently.

In FrameMaker equations are very similar to graphics. Both are placed within anchored frames, and the anchored frames contain one or more objects—for graphics these are graphic objects, and for equations they are terms of the equation. In the context of structure and import/export of markup, an equation is treated as a single graphic object within an anchored frame. FrameMaker does nothing to represent the equation's terms as any type of structure. You cannot use rules to describe the internal structure of equations. However, you can write a structure API client to support a completely different model for equations (such as MathML).

FrameMaker has a default set of element and attribute definition list declarations for representing graphics and equations as elements. You can use read/write rules to support variations of the default representation, whether you start with an EDD or a DTD.

In this chapter

This chapter describes how FrameMaker translates graphics and equations and how you can change that translation. It contains these sections.

–"Creating graphic files on export" on page 373
–"Specifying the entity name on export" on page 387
–"Changing how FrameMaker writes out the size of a graphic" on page 389

Default translation

FrameMaker has well-defined representations for graphics and equations. They are given element structure as *graphic elements* and *equation elements*. You can create a graphic in an external tool and then import it into FrameMaker, you can use the software's tools to create the graphic, or you can combine methods. Whatever the method, the software puts the graphic into an anchored frame in the document. For equations, you create an equation using the software's equations tools. FrameMaker treats the equation as a single graphic object inside an anchored frame and exports this anchored frame to markup.

Supported graphic file formats

FrameMaker supports multiple graphic file formats for graphic objects created in an external tool, such as the QuickDraw PICT (PICT) format or the Computer Graphics Metafile (CGM) format. If you have a document with a graphic in a file format that the software doesn't support, you can supply your own graphic filter to allow the graphic to be processed. You can have such a document regardless of whether you are starting from markup or from FrameMaker.

The available export graphic formats for FrameMaker are:

Code	Meaning
CDR	CorelDRAW
CGM	Computer Graphics Metafile
DIB	Device-independent bitmap (Windows® import only)
DRW	Micrografx CAD
DWG	Autocad Drawing
DXF	Autodesk Drawing eXchange file (CAD files)
EMF	Enhanced Metafile
EPSB	Encapsulated PostScript® Binary
EPSD	Encapsulated PostScript with Desktop Control Separations (DCS)
EPSI	Encapsulated PostScript Interchange
FRMI	FrameImage
FRMV	FrameVector
G4IM/GP4	CCITT Group 4 to Image
GEM	GEM file

Code	Meaning
GIF	Graphics Interchange Format (Compuserve)
HPGL	Hewlett-Packard® Graphics Language
IGES	Initial Graphics Exchange Specification (CAD files)
JPG, JPE (JFIF)	Joint Photographer Experts Group (actual file format is JPEG File Interchange Format
MooV	QuickTime Movie
OLE	Object Linking and Embedding Client (Microsoft®)
PCX	PC Paintbrush
PICT	QuickDraw PICT
PNG	Portable Network Graphic
PNTG	MacPaint
SNRF	Sun Raster File
SRGB	SGI™ RGB
SVG	Scalable Vector Graphics
SWF	Adobe Flash®
TIFF	Tag Image File Format
U3D	Universal 3D
WMF	Windows Metafile
XWD	X Windows System Window Dump file

General import and export of graphic elements

Markup does not standardize the representation of graphics or equations in a DTD. Consequently, their representation can be unique to a DTD. Without rules, the software can't identify elements and attributes representing a graphic or an equation when creating an EDD from a DTD. However, you can easily use the element and attribute structure provided with FrameMaker, or a variant of that structure.

Even though markup does not have a standard representation for graphics, there are some commonly used frameworks and FrameMaker provides support for two of them—the read/write rules work well to modify them. If your element and attribute structure matches one of these frameworks, it should be relatively straightforward for you to import or export graphics or equations. However, if your DTD uses a radically different framework, you'll have to write a structure API client.

Basically, FrameMaker assumes that in markup a graphic or equation is an empty element with either an `ENTITY` attribute identifying an external data entity that is the actual graphic or a `CDATA` attribute whose value is a filename containing the graphic. Other attributes can represent

properties of the FrameMaker anchored frame in which the graphic sits inside a FrameMaker document.

> **Note: XML:** The XML specification states that an external graphic is a non-parsed entity and must be referenced via an `ENTITY` or `ENTITIES` attribute in an empty element. This corresponds very well with the preferred framework for representing graphics in a FrameMaker structure application.

If the graphic file is specified in an entity declaration, the entity name is always kept with the graphic inset in the FrameMaker document. Note, however, that there is no way to see the entity name by inspecting a graphic inset. On export, this entity name becomes the value of the entity attribute of the graphic element. If the graphic's entity declaration is in the internal DTD subset of the imported markup document, FrameMaker stores the information so it can recreate the entity declaration on export. For information on how FrameMaker saves the entity definition, see "Importing graphic entities" on page 375. For information on how FrameMaker exports entity declarations, see "Exporting entity declarations" on page 373. For information on how FrameMaker exports graphic files, see "Creating graphic files on export" on page 373.

An external data entity has an associated notation, designed to tell the structure application how to render the entity data. When the software reads a markup document, rather than storing this information in attributes or variables in FrameMaker, it stores the information directly in the graphic's associated anchored frame. This results in fewer attributes on the FrameMaker element than were on the markup element. When exporting a document to markup, the software recreates this information in the attributes and entities it writes.

You can use rules to modify some of what FrameMaker writes on export, such as the name of the entity or which, if any, of the graphic's facets get written as files. There are few rules that relate specifically to modifying what the software does to graphics on import.

On export to markup

Some properties of FrameMaker graphics and equations are not explicit in their element and attribute structure and the software translates these properties as attributes in markup. These properties are named so that you can refer to them in rules. For example, the `align` attribute corresponds to the `alignment` property and indicates how an anchored frame is aligned on the page.

Also, in some circumstances FrameMaker will write out a new graphic file when exporting a document to markup. For more information, see "Creating graphic files on export" on page 373

Text of default graphics and equations declarations

> **Note: XML and SGML:** The following examples show declarations for XML. For SGML, the instances of NMTOKEN can be expressed as NUMBER or NAME

The default DTD declarations for graphics and equations are described in detail in the following sections. In summary, the following examples show the default declarations.

```
<!-- graphic and equation elements-->
<!ELEMENT mygraphic    EMPTY>
<!ELEMENT myequation    EMPTY>

<!--Attributes for equations-->
<!ATTLIST myequation
entity      ENTITY  #IMPLIED <!-- graphic entity -->
file        CDATA   #IMPLIED <!-- graphic file, SGML only -->
align       NMTOKEN #IMPLIED <!--frame alignment on page-->
angle       CDATA   #IMPLIED <!--frame angle in degrees-->
bloffset    CDATA   #IMPLIED <!--frame baseline offset-->
cropped     NUMBER  #IMPLIED <!--nonzero for cropped-->
float       NUMBER  #IMPLIED <!--nonzero for floating-->
height      CDATA   #IMPLIED <!--frame height-->
nsoffset    CDATA   #IMPLIED <!--frame near-side offset-->
position    NAME    #IMPLIED <!--frame position-->
width       CDATA   #IMPLIED <!--frame width-->
attribute_declarations_specific_to_this_equation_element
>
```

```
<!--Attributes for graphics-->
<!ATTLIST mygraphic
entity   ENTITY  #IMPLIED
file     CDATA   #IMPLIED <!-- graphic file, SGML only -->
align    NMTOKEN #IMPLIED <!--frame alignment on page-->
angle    CDATA   #IMPLIED <!--frame angle in degrees-->
bloffset CDATA   #IMPLIED <!--frame baseline offset-->
cropped  NMTOKEN #IMPLIED <!--nonzero for cropped-->
float    NMTOKEN #IMPLIED <!--nonzero for floating-->
height   CDATA   #IMPLIED <!--anchored frame height-->
nsoffset CDATA   #IMPLIED <!--frame near-side offset-->
position NMTOKEN #IMPLIED <!--frame position-->
width    CDATA   #IMPLIED <!--frame width-->
dpi      NMTOKEN #IMPLIED <!--ignored if impsize specified-->
impang   CDATA   #IMPLIED <!--angle in frame, in deg-->
impby    (ref|copy) #IMPLIED <!--import by reference or copy-->
impsize  CDATA   #IMPLIED <!--import size (width & height)-->
sideways NMTOKEN #IMPLIED <!--1 if object turned sideways-->
xoffset  CDATA   #IMPLIED <!--horizontal offset in frame-->
yoffset  CDATA   #IMPLIED <!--vertical offset in frame-->
attribute_declarations_specific_to_this_graphic_element
>
```

Element and attribute structure

This set of declarations represents graphics and equations using elements with a declared content of `EMPTY` and two primary attributes, `entity` and `file`. For a given element, the software writes only one of `entity` or `file`. The `entity` attribute is of type `ENTITY` and identifies an external data entity containing the graphic or equation. The `file` attribute is of type `CDATA` and its value is the name of a file containing the graphic or equation.

> **Note: XML:** For XML you should use the `entity` attribute to treat a graphic file as an unparsed entity.

When FrameMaker encounters a graphic or equation on export, it writes a start-tag for an empty element, including values for its attributes, as appropriate. Under some circumstances, it also writes a file containing the graphic or equation itself. You can use read/write rules or a structure API client to change these behaviors. For information about exporting graphic or equation files, see "Creating graphic files on export" on page 373. For information on facets, see the FrameMaker user's manual.

Graphic and equation elements have the same set of attributes describing common properties of anchored frames. Graphic elements have additional attributes for properties relevant only to graphics created outside FrameMaker. Finally, your EDD might define other attributes for a particular graphic or equation element. The software exports these attributes as well.

Entity and file attributes

The default declarations include both an `entity` and a `file` attribute for each markup element that corresponds to a graphic or equation. When exporting a FrameMaker document, the software stores the location of the graphic file in one or the other of those attributes, depending on which is present in the markup element declaration.

If the markup element has an `entity` attribute defined, then on export FrameMaker writes a value for the `entity` attribute in markup. If there is no corresponding entity in the structure application's DTD, it also generates the corresponding entity declaration in the document's internal DTD subset. By default, FrameMaker doesn't generate a public identifier; you must supply a structure API client to do so.

If a markup element has a `file` attribute but no `entity` attribute, the software writes a value for the `file` attribute. On import, if the markup element has no value for the `entity` attribute, the value of the `file` attribute is used. Otherwise, the `entity` value is used.

Anchored frame properties

All markup elements corresponding to graphics and equations have the following implied attributes that supply information about the anchored frame containing the graphic or equation:

- `align` corresponds to the `alignment` property and indicates the anchored frame's horizontal alignment on the page. Its possible values and the corresponding FrameMaker property values are as follows:

Attribute value	Property value
`aleft`	`align left`
`acenter`	`align center`
`aright`	`align right`
`ainside`	`align inside`
`aoutside`	`align outside`

- `angle` corresponds to the `angle` property and indicates an angle of rotation for the anchored frame containing the graphic. The value is assumed to represent a number. The software interprets this attribute as a number of degrees of rotation. You must specify exact multiples of 90 degrees. Otherwise, the value is ignored and the graphic is imported at 0 degrees (default). For example, if 89 degrees is specified, the graphic imports at 0 degrees.
- `bloffset` corresponds to the `baseline offset` property and indicates how far from the baseline of a paragraph to place an anchored frame. The value is assumed to represent a number. If not supplied, the value is `0`. The `bloffset` attribute is relevant only for anchored frames whose `position` attribute is one of `inline`, `sleft`, `sright`, `snear`, or `sfar`.
- `cropped` corresponds to the `cropped` property and indicates whether a wide graphic should be allowed to extend past the margins of the text frame. The value is either `0` or `1`. If not supplied, the value is `1`, indicating that the graphic should not extend past the margins. The

cropped attribute is relevant only for anchored frames whose position attribute is one of top, below, or bottom.

- float corresponds to the floating property and indicates whether the graphic should be allowed to float from the paragraph to which it is attached. The value is 0 or 1. If not supplied, the value is 0, indicating that the graphic must stay with the paragraph. The float attribute is relevant only for anchored frames whose position attribute is one of top, below, or bottom.
- height corresponds to the height property and indicates the height of the anchored frame. If not supplied, the value for a single imported graphic object is the sum of the height of the object plus twice the value of the yoffset attribute. For all other graphics and for equations, the value is the height of the object.
- nsoffset corresponds to the near-side offset property and indicates how far to set a frame from the text frame to which the frame is anchored. The value is assumed to represent a number. If not supplied, the value is 0. The nsoffset attribute is relevant only for anchored frames whose position attribute is one of sleft, sright, snear, or sfar.
- position corresponds to the position property and indicates where on the page to put the anchored frame. If not supplied, the value is below. Possible values of position and the corresponding FrameMaker property values are as follows:

Attribute value	Property value
inline	inline
top	top
below	below
bottom	bottom
sleft	subcol left
sright	subcol right
snear	subcol nearest
sfar	subcol farthest
sinside	subcol inside
soutside	subcol outside
tleft	textframe left
tright	textframe right
tnear	textframe nearest
tfar	textframe farthest
tinside	textframe inside
toutside	textframe outside
runin	run into paragraph

- width corresponds to the width property and indicates the width of the anchored frame. If not supplied, the value for a single imported graphic object is the sum of the width of the object plus twice the value of the xoffset attribute. For all other graphics and for equations, the value is the width of the object.

For more on these properties, see the *Using FrameMaker* for information about anchored frames.

Other graphic properties

Markup elements corresponding to graphic elements can have the following additional implied attributes:

- alt-text corresponds to the Alternate property and provides a string to display when the XML formatting software cannot display the graphic.
- dpi corresponds to the dpi property and indicates how to scale an imported graphic object. The software ignores this attribute if it finds a value for the impsize attribute. It produces only one of the attributes—dpi or impsize. The value of the dpi attribute is a number. If not supplied, the value is 72.
- impang corresponds to the import angle property and indicates an angle of rotation for the graphic inside its anchored frame. The value is assumed to represent a number. FrameMaker interprets this attribute as a number of degrees of rotation. If not supplied, the value is 0.0.
- impsize corresponds to the import size property and indicates the size of the imported graphic object by specifying a width and height. This property is set by choosing the option Fit in Selected Rectangle from the Imported Graphic Scaling dialog box. If not supplied, a dpi attribute value must be supplied.
- impby corresponds to the import by reference or copy property and indicates whether an imported graphic object remains in a separate file or is incorporated in the FrameMaker document on import from markup. The value is either ref or copy. If not supplied, the value is ref, indicating that the object should not be copied into the document.
- rasterdpi corresponds to the rasterdpi property of SVG graphics and indicates how to scale the FrameImage facet of an imported SVG graphic. It also determines the dpi when converting SVG to a raster format on export. The value of the dpi attribute is a number. If not supplied, the value is 72.
- sideways corresponds to the sideways property and indicates whether the imported graphic should be inverted around its vertical axis. The value is 0 or 1. If not supplied, the value is 0, indicating that the graphic shouldn't be inverted.
- xoffset corresponds to the horizontal offset property and indicates how far the graphic object is offset from the right and left edges of the anchored frame. The value is assumed to represent a number. If not supplied, the value is 6.0pt.
- yoffset corresponds to the vertical offset property and indicates how far the graphic object is offset from the top and bottom edges of the anchored frame. The value is assumed to represent a number. If not supplied, the value is 6.0pt.

For more on these properties, see the *Using FrameMaker* for information about importing graphics.

Exporting entity declarations

When importing a markup document, use read/write rules to import markup elements as graphic elements or equations. If the graphic file is specified by an entity declaration, the software stores enough information about the entity declaration to recreate it on export.

If the entity declaration was made in the internal DTD subset of the document instance, that information is stored on the Entity Declarations reference page of the FrameMaker document. On export, the software will use this information to recreate the entity declarations in the resulting markup document. For more about saving entity declaration information in a FrameMaker document, see "Importing graphic entities" on page 375.

On export to markup, the software ensures the appropriate entities are in the document instance as follows. Unless otherwise specified, this table assumes the graphic element's declaration in markup includes an `entity` attribute:

If:	FrameMaker:
The graphic filename and entity name match an entity declaration in the structure application's DTD...	Uses the matching entity name as the value of the graphic element's `entity` attribute in markup.
The graphic filename matches an entity declaration on the Entity Declarations reference page...	Declares the entity in the DTD subset of the resulting markup document. It uses the entity name that was stored with the graphic inset for the entity declaration and for the `entity` attribute in the resulting graphic element in markup.
The graphic filename matches no entity declarations, or the graphic inset has no entity name associated with it...	Generates an entity declaration in the DTD subset of the resulting document instance. It names the entity graphic1, graphic2, etc. It writes the entity name to the `entity` attribute in the resulting graphic element.
The SGML graphic element does not include an `entity` attribute...	Does not generate an entity declaration. The filename for the graphic file is used as the value for the `file` attribute of the SGML graphic element.

Creating graphic files on export

For graphics imported by reference, the software uses the same file for the markup document as it does for the FrameMaker document. On export, it creates new files for graphic and equation elements that meet any of the following conditions:

- The graphic file was imported by copy.
- The user changed the graphic content in any way while editing the document in FrameMaker. This includes adding graphic content via the graphics palette, or importing an additional file

into the anchored frame. Note that the user may delete the existing graphic file and import another one. If the new file matches a file in the DTD's entity declarations, or it matches a file on the Entity Declarations reference page, the exported markup will refer to this newly corresponding entity.

For each one of such graphics and equations, the software creates a new graphic file. For information on whether the software references the file via an entity or via the file attribute of the graphic element in markup, see "Exporting entity declarations," (the previous section).

A graphic element may be an anchored frame containing only a single imported graphic object. If so, it is likely to have a single facet that isn't one of the software's internal facets. In this case, the written file is in the graphic format indicated by the facet. For all other graphics and for equations, the written file is in CGM format.

If FrameMaker exports the single facet of a graphic element, the software exports the file in the indicated format and uses the facet name as the notation name. In all other cases, the software exports the file in CGM format and its notation name is `CGM`.

For example, assume you have an instance of the `Graph` graphic element that contains graphics you created with FrameMaker's graphics tools. By default, the software creates the following markup entity for it:

```
<!ENTITY graph1 SYSTEM "graph1.cgm" NDATA cgm>
```

Also, the `entity` attribute of the graphic element has a value of `graph1` to correspond with the above entity.

On import to FrameMaker

In the absence of read/write rules, FrameMaker cannot identify the elements and attributes of a markup document that correspond to a graphic or equation. Therefore, it translates them as container elements in the EDD. You must supply rules and perhaps a structure API client to reflect the appropriate structure.

If your DTD uses the default graphic and equation declarations described in the preceding sections, the only rule you need is one to identify the element as a graphic or equation. If you do so, the attributes translate automatically. The translation occurs either if you started by creating your DTD from an EDD or if you started with an existing DTD without declarations for graphics or equations and added the default declarations. For a description of these declarations, see "Text of default graphics and equations declarations" on page 368.

Your markup document can use either the `entity` attribute or (for SGML) the `file` attribute to specify a graphic or equation. If a single element has specified values for both attributes, the software uses the value of the `entity` attribute, ignoring the value of the `file` attribute. FrameMaker accepts the name of any external data entity as a value for `entity`, regardless of whether the entity is declared to be `CDATA`, `SDATA`, or `NDATA`.

Importing graphic entities

If the graphic element in markup uses the `entity` attribute to identify the graphic file, then the actual graphic file must be identified in an entity declaration. FrameMaker will read the markup, importing the graphic file by reference into an anchored frame. If the entity declaration is not in the main DTD, then it should be in the document's internal DTD subset. In that case, FrameMaker saves information about the entity declaration on the *Entity Declarations* reference page of the resulting document.

The software saves the entity name with the imported graphic inset. The software uses the entity name, plus the information on the *Entity Declarations* reference page, to recreate entity declarations when exporting to markup. For graphic entities the software can use the same graphic file for import and export under most circumstances. For information about exporting entity declarations, see "Exporting entity declarations" on page 373. For information about when the software does and does not use the same graphic file, see "Creating graphic files on export" on page 373.

Graphic attributes and properties

The attributes defined in the default declarations for graphics and equations specify properties for the graphic or equation object in FrameMaker. On import, none of these attributes translate to FrameMaker element attributes. They are all saved as properties of the object or the graphic element's anchored frame.

If the graphic is a MIF file, and the MIF file contains an anchored frame, the MIF will also have anchored frame property values. The attributes specified in the markup document take precedence over the anchored frame property values of the MIF file. However, the object properties specified in the MIF file take precedence over the values specified in the markup document. If a MIF file doesn't contain an anchored frame and the graphic element's start-tag doesn't provide attribute values, then the software uses default values in creating the anchored frame.

Graphic file formats

FrameMaker expects the graphic file to be in one of the formats it supports. For the list of graphic file formats FrameMaker supports, see "Supported graphic file formats" on page 365.

If the file is in a format the software does not support, you may be able to add a filter to support that format. For information on the graphic filters included with the software and on how to add a new one, see the online manual about using filters for Frame products.

If the graphic is in a MIF file, FrameMaker assumes that the first anchored frame or equation on a body page in the file is the content of the element. Some filters create MIF files in which the graphic objects aren't put into an anchored frame. When it reads such a file, the software processes all the graphic objects on the first body page as a single anchored frame.

Modifications to the default translation

The read/write rules for graphics and equations allow you, among other things, to:

- identify which elements in markup correspond to graphic or equation elements
- associate attributes markup with FrameMaker formatting properties
- or specify the graphic file format for an exported graphic

The rules for equations and for graphics of various sorts are very similar. Typically, you start with an element, identify it either as a graphic or an equation, indicate what to do with attributes that represent formatting information, and indicate how to treat the element under different export circumstances.

The following sections describe some of the ways you can modify the default translation of graphics and equations. The first two procedures are relevant for translating any graphic or equation elements. For additional ways to modify the translation of graphics and equations, see the cross-references at the end of each section.

For a summary of read/write rules relevant to translating graphics, see Developer Reference, page 37: Graphics. For a summary of read/write rules relevant to translating equations, see Developer Reference, page 36: Equations. For a list of the formatting properties associated with translating graphics and equations, see "Text of default graphics and equations declarations" on page 368.

Identifying and renaming graphic and equation elements

If you create or update your EDD from an existing DTD, or if you want to rename elements on import or export, you must use a rule to identify which markup elements correspond to graphic or equation elements in FrameMaker. To do so, use one of these rules:

```
element "gi" is fm graphic element ["fmtag"];
element "gi" is fm equation element ["fmtag"];
```

where *gi* is a generic identifier and the optional *fmtag* argument allows renaming the element. If *fmtag* is not specified, the name remains the same in the two representations. For example, to specify that the markup element `pict` corresponds to the graphic element `Picture` in FrameMaker, use this rule:

```
element "pict" is fm graphic element "Picture";
```

For information on the rules used in this example, see

- Developer Reference, page 56: element
- Developer Reference, page 111: is fm equation element
- Developer Reference, page 113: is fm graphic element.

Specifying a graphic or equation element in Schema

In order for FrameMaker to translate a Schema element to a graphic object or equation upon import of an XML file associated with a Schema declaration, you must use element definitions and rules such as the following:

Schema element definition for a graphic object

```
<xsd:element name="Graphic1"><xsd:complexType>
<xsd:attribute name="entity" type="xsd:ENTITY" use="optional"/>
<xsd:attribute name="align" type="xsd:NMTOKEN" use="optional"/>
<xsd:attribute name="angle" type="xsd:string" use="optional"/>
<xsd:attribute name="bloffset" type="xsd:string"
                use="optional"/>
<xsd:attribute name="cropped" type="xsd:NMTOKEN"
                use="optional"/>
<xsd:attribute name="float" type="xsd:NMTOKEN" use="optional"/>
<xsd:attribute name="height" type="xsd:string" use="optional"/>
<xsd:attribute name="nsoffset" type="xsd:string"
                use="optional"/>
<xsd:attribute name="position" type="xsd:NMTOKEN"
                use="optional"/>
<xsd:attribute name="dpi" type="xsd:NMTOKEN" use="optional"/>
<xsd:attribute name="impang" type="xsd:string" use="optional"/>
<xsd:attribute name="impby" use="optional">
   <xsd:simpleType>
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value="ref"/>
         <xsd:enumeration value="copy"/>
      </xsd:restriction></xsd:simpleType>
</xsd:attribute>
<xsd:attribute name="impsize" type="xsd:string"
                use="optional"/>
<xsd:attribute name="sideways" type="xsd:NMTOKEN"
                use="optional"/>
<xsd:attribute name="xoffset" type="xsd:string"
                use="optional"/>
<xsd:attribute name="yoffset" type="xsd:string"
                use="optional"/>
</xsd:complexType></xsd:element>
```

Read/write rule for a graphic object

```
element "Graphic1" is fm graphic element “Graphic1”;
```

Schema element definition for an equation

```
<xsd:element name="Equation1"><xsd:complexType>
<xsd:attribute name="entity" type="xsd:ENTITY" use="optional"/>
<xsd:attribute name="align" type="xsd:NMTOKEN" use="optional"/>
<xsd:attribute name="angle" type="xsd:string" use="optional"/>
<xsd:attribute name="bloffset" type="xsd:string"
                use="optional"/>
<xsd:attribute name="cropped" type="xsd:decimal"
                use="optional"/>
<xsd:attribute name="float" type="xsd:decimal" use="optional"/>
<xsd:attribute name="height" type="xsd:string" use="optional"/>
<xsd:attribute name="nsoffset" type="xsd:string"
                use="optional"/>
<xsd:attribute name="position" type="xsd:NMTOKEN"
                use="optional"/>
<xsd:attribute name="width" type="xsd:string" use="optional"/>
</xsd:complexType></xsd:element>
```

Read/write rule for an equation

```
element "Equation1" is fm equation element "Equation1";
```

For information on the rules used in these examples, see

- Developer Reference, page 56: element
- Developer Reference, page 111: is fm equation element
- Developer Reference, page 113: is fm graphic element.

Exporting graphic and equation elements

FrameMaker supplies several rules for modifying the software's behavior when exporting a graphic or equation element. These rules are explained in later sections of this chapter. They occur as subrules of a rule that also indicates the type of graphic or equation being exported. There are three rule constructions for this purpose:

```
element "gi" {
   is fm equation element ["fmtag"];
   writer equation { subrules }
}

element "gi" {
   is fm graphic element ["fmtag"];
   writer facet "facet" { subrules }
}
```

```
element "gi" {
   is fm graphic element ["fmtag"];
   writer anchored frame { subrules }
}
```

where *gi* is a generic identifier, *fmtag* is an optional FrameMaker element tag, *subrules* are described later, and *facet* is a graphic facet.

Use the first of these constructions to specify how the software exports an equation element under all circumstances.

Use the second construction to specify how it exports a graphic element when that graphic element contains only a single facet with the name specified by *facet*. This corresponds to the situation where the graphic element is an anchored frame containing only a single imported graphic object whose original file was in the *facet* graphic format.

Use the third construction to tell it how to export a graphic element under all other circumstances. You can use the `facet` construction multiple times if you want the software to treat file formats differently.

For example, assume you use the `Graphic` element for all graphic elements. If the graphic contains any single facet, you assume the graphic was imported as an entity and you want the default behavior. However, if the author used FrameMaker graphic tools to create the objects in the graphic element, you want the file written in QuickDraw PICT format.

To accomplish all this, you would use this rule:

```
element "graphic" {
   is fm graphic element;
   writer anchored frame export to file "$(entity).pic" as
   "PICT";
}
```

Because the entities specify graphic files that are unchanged, the software doesn't create new graphic files for those elements. However, if the author created a graphic using the FrameMaker graphic tools, there is no corresponding entity. The software will write the graphic file in PICT format, and create a corresponding graphic entity. The markup graphic element will refer to that entity via the `entity` attribute.

For more information on these export options, see "Creating graphic files on export" on page 373 and "Changing the file format of the graphic file on export" on page 385. For information on these rules, see

- Developer Reference, page 43: anchored frame
- Developer Reference, page 74: facet
- Developer Reference, page 65: equation
- Developer Reference, page 56: element
- Developer Reference, page 111: is fm equation element

- Developer Reference, page 113: is fm graphic element
- Developer Reference, page 69: export to file
- Developer Reference, page 166: writer

Representing the internal structure of equations

As mentioned earlier in this chapter, FrameMaker treats an equation in the same way it does a graphic element. That is, on export to markup an equation is represented as a single empty element with an external data entity pointing to a CGM file containing the equation.

If you want to represent a FrameMaker equation as an element with subelement structure instead, you must write a structure API client. If you do so, you will use the FDK object `FO_Math` and manipulate the equation structure as represented in MIF. For information on using the FDK to manipulate equations, see the *FDK Programmer's Guide*. For information on writing structure API clients, see the *Structure Import/Export API Programmer's Guide*.

Renaming markup attributes that correspond to graphic properties

FrameMaker represents properties of graphics (such as the height or width of the enclosing anchored frame) directly as part of the anchored frame, rather than as attributes on the graphic element. In markup however, these properties may be represented as attributes. In other words, there are certain markup attributes that do not translate to FrameMaker attributes, but instead directly to graphic properties. "Anchored frame properties" on page 370 and "Other graphic properties" on page 372 describe the default names for the markup attributes corresponding to graphic properties.

You can choose names other than the default ones for these markup attributes. To do so, use this version of the `attribute` rule:

```
attribute "attr" is fm property prop;
```

If your DTD uses a particular attribute name for the same purpose within multiple elements, you may want to use this rule as a highest-level rule to set a default. Otherwise, specify it within an `element` rule.

For example, assume you have four different markup elements representing graphics and equations. All four elements use the attribute `h` to represent the height of the anchored frame. Use the rule:

```
attribute "h" is fm property height;
```

With this rule, the software interprets the attribute `h` as the height of an anchored frame for all elements in which it occurs. If you use the same attribute for another purpose in another element, you'll have to write another `attribute` rule to handle it appropriately.

As another example, you may have only one element, `pic`, representing a graphic element and use the attribute `h` for unrelated purposes in other elements. In this case, you should restrict the

association of h with the height property to the pic element instead of using the form of the rule that applies to all elements. You can use this rule:

```
element "pic" {
   is fm graphic element "Picture";
   attribute "h" is fm property height;
}
```

With this rule, other elements can use the h attribute for different purposes.

In both of these examples, the software creates an EDD from a DTD without creating an attribute that corresponds to the h attribute. When importing or exporting markup documents, it uses the attribute to read or write the appropriate information from the graphic.

The dpi and impsize attributes defined for graphic elements deserve special attention. For each generic identifier that represents an imported graphic object, you can decide whether its size is specified with the dpi or the impsize attribute. By default, the software uses the dpi attribute, but you can change this default with the specify size rule. For information on this rule, see "Changing how FrameMaker writes out the size of a graphic" on page 389.

For information on the rules used in these examples, see Developer Reference, page 46: attribute, Developer Reference, page 56: element, Developer Reference, page 113: is fm graphic element, and Developer Reference, page 80: fm property.

Omitting representation of graphic properties in markup

Some properties of graphics and equations have no explicit representation in markup. In such cases, you may want to use a rule to make the information explicit in FrameMaker. You can use this rule to do so:

```
fm property prop value is "propval";
```

where *prop* is the FrameMaker property and *propval* is the value to use.

With this rule, the software creates a DTD from an EDD without creating the corresponding markup attribute. When importing a markup document, it causes the software to assign a particular value to one of the formatting properties. When exporting a document, it tells FrameMaker not to write an attribute with the value.

The fm property rule can be used at the highest level to set a default or within an element rule to be restricted to a single element.

For example, you may have a markup element bitmap treated as a graphic element in FrameMaker. Furthermore, you know that you never want to make a copy of such an object in the

FrameMaker document; you do not want your end users to have the option of importing by copy. You can accomplish this as follows:

```
element "bitmap" {
   is fm graphic element;
   fm property import by reference or copy value is "ref";
}
```

When creating an EDD from a DTD, the software does not create an `impby` attribute for the `bitmap` element. Consequently, when exporting a FrameMaker document to markup, it does not try to write a value for the `impby` attribute. When importing a markup document, it automatically sets the property value to `ref`.

For a summary of the FrameMaker graphic properties, see "Text of default graphics and equations declarations" on page 368. For information on the rules used in this example, see Developer Reference, page 56: element, Developer Reference, page 113: is fm graphic element, and Developer Reference, page 80: fm property.

Omitting optional elements and attributes from the default DTD declarations

The default DTD declarations provided with FrameMaker may be more general than your situation requires. If you create an initial version of your DTD from an EDD, it will always include the full set of default declarations for all graphic and equation elements. If you don't need all of this functionality, you may want to simplify the declarations.

For example, if your graphic elements are always created using FrameMaker's graphic tools, you could remove these declarations:

```
<!ATTLIST graphic_element
   dpi      NMTOKEN #IMPLIED <!--ignored if impsize specified-->
   impang   CDATA  #IMPLIED <!--import angle in frame in deg-->
   impby    (ref|copy) #IMPLIED <!--import by reference or copy-->
   impsize  CDATA  #IMPLIED <!--import size (width & height)-->
   sideways NMTOKEN #IMPLIED <!--1 if object turned sideways-->
   xoffset  CDATA  #IMPLIED <!--horizontal offset in frame-->
   yoffset  CDATA  #IMPLIED <!--vertical offset in frame-->
>
```

If you do so, FrameMaker won't produce or expect values for these attributes.

> **Note: SGML:** In the above example, the `dpi` and `sideways` attributes are specified in XML using `NMTOKEN` for the numeric value. For SGML you can use `NUMBER`.

Specifying the data content notation on export

When the software writes an external data entity for a graphic or equation element, it uses the first eight characters of the facet name as the data content notation by default. If the graphic or equation element has only internal FrameMaker facets, it uses CGM as the data content notation. Your markup declarations may use different data content notations. If so, you can use the `notation` rule to associate a notation name with a facet or with general anchored frames or equations. The `notation` rule is of this form:

```
element "gi" {
   is fm graphic_or_equation element ["fmtag"];
   writer type ["name"] notation is "notation_name";
}
```

where *`gi`* is a generic identifier, *`graphic_or_equation`* is one of the keywords `graphic` or `equation`; *`fmtag`* is an optional FrameMaker element tag; *`type`* is one of the keywords `anchored frame`, `facet`, or `equation`; *`name`* is a facet name you supply only if *`type`* is `facet`; and *`notation_name`* is the data content notation in the corresponding markup entity declaration.

For example, assume you have this rule:

```
element "graphic" {
   is fm graphic element;
   writer {
   facet "XWD" {
      notation is "xwd";
      export to file "$(docname).xwd";
}}}
```

In this case, when the software creates an external data entity for a `Graphic` element that has a facet whose name is `xwd`, it creates an entity declaration of this form:

```
<!ENTITY graphic1 SYSTEM "docname.xwd" NDATA xwd>
```

where docname is the name of the FrameMaker document. This example assumes that you have added an export filter to export a graphic in the XDump file format. For information on changing the filename written for this entity, see "Changing the name of the graphic file on export," next.

For more information on these rules, see:

- Developer Reference, page 144: notation is
- Developer Reference, page 56: element
- Developer Reference, page 113: is fm graphic element
- Developer Reference, page 111: is fm equation element
- Developer Reference, page 43: anchored frame
- Developer Reference, page 65: equation

- Developer Reference, page 74: facet
- Developer Reference, page 166: writer.

Changing the name of the graphic file on export

Under certain conditions, when it exports a FrameMaker document, the software creates a file for each graphic and equation. For more information, see "Creating graphic files on export" on page 373. For the circumstances where the software will create a graphic file, you can change the name of the exported graphic file using the following rule:

```
element "gi" {
   is fm graphic_or_equation element ["fmtag"];
   writer type ["name"] export to file "fname";
}
```

where *gi* is a generic identifier; *graphic_or_equation* is one of the keywords `graphic` or `equation`; *fmtag* is an optional FrameMaker element tag; *type* is one of the keywords `anchored frame`, `facet`, or `equation`; *name* is a facet name you supply only if *type* is `facet`; and *fname* is the new filename. The *fname* argument can use these variables:

Variable	Meaning
`$(entity)`	The value of the corresponding markup element's `entity` attribute. If the source of the graphic inset wasn't originally a markup entity, this variable evaluates to a unique name based on the name of the element.
`$(docname)`	The name of the FrameMaker file, excluding any extension or directory information.

For example, assume you have the default declarations for the `graphic` element and you have this rule:

```
element "graphic" {
   is fm graphic element;
   attribute "entity" {
      is fm property entity;
   }
   writer facet "XWD" {
      notation is "xwd";
   convert referenced graphics;
      export to file "$(entity).xwd";
}}
```

With these rules, assume you imported a graphic element whose `entity` attribute had the value of `flower`. When you export the FrameMaker document to markup, the software writes this entity declaration:

```
<!ENTITY flower SYSTEM "flower.xwd" NDATA xwd>
```

It writes the graphic to a file named `flower.xwd` using the X Windows Dump format.

For more information on these rules, see:

- Developer Reference, page 69: export to file
- Developer Reference, page 144: notation is
- Developer Reference, page 46: attribute
- Developer Reference, page 116: is fm property
- Developer Reference, page 104: is fm attribute
- Developer Reference, page 56: element
- Developer Reference, page 113: is fm graphic element
- Developer Reference, page 111: is fm equation element
- Developer Reference, page 43: anchored frame
- Developer Reference, page 65: equation
- Developer Reference, page 74: facet
- Developer Reference, page 166: writer

Changing the file format of the graphic file on export

By default, when it creates a graphic file on export, FrameMaker writes the graphic file for a graphic or equation element in either CGM format or the format of the single facet it exports. For information on when the software creates a graphic file, see "Creating graphic files on export" on page 373.

You can tell the software to write graphics files in a different file format. To do so, you must first make sure that the format is one known to FrameMaker. For the list of graphic file formats FrameMaker supports, see "Supported graphic file formats" on page 365.

For information on which graphic export filters the software provides and on how to add new ones, see the online manual that describes using filters with FrameMaker products.

Once you are sure that the software can export your format of choice, you use a variant of the `export to file` rule described in "Changing the name of the graphic file on export," (the previous section).

The general form of this rule is:

```
element "gi" {
   is fm graphic_or_equation element ["fmtag"];
   writer type ["name"] export to file "fname" as "format";
}
```

where *gi* is a generic identifier; *graphic_or_equation* is one of the keywords `graphic` or `equation`; *fmtag* is an optional FrameMaker element tag; *type* is one of the keywords `anchored frame`, `facet`, or `equation`; *name* is a facet name you supply only if *type* is `facet`; *fname* is the new filename; and *format* is the file format. The *fname* argument can use these variables:

Variable	Meaning
`$(entity)`	The value of the corresponding markup element's `entity` attribute. If the source of the graphic inset wasn't originally a markup entity, this variable evaluates to a unique name based on the name of the element.
`$(docname)`	The name of the FrameMaker file, excluding any extension or directory information.

For example, FrameMaker writes CGM files for all equation elements by default. If you want it to write QuickDraw PICT files instead, you can use this rule:

```
element "eqn" writer equation
export to file "eqn.pic" as "PICT";
```

For more information on these rules, see

- Developer Reference, page 69: export to file
- Developer Reference, page 56: element
- Developer Reference, page 113: is fm graphic element
- Developer Reference, page 111: is fm equation element
- Developer Reference, page 43: anchored frame
- Developer Reference, page 65: equation
- Developer Reference, page 74: facet
- Developer Reference, page 166: writer.

Changing the file format for graphics imported by reference

By default, if a graphic or equation element contains a single graphic file that is imported by reference, when exporting a FrameMaker document the software does not create a new graphic file for the element. However, you can force the software to create a new graphic file for such elements. To do this, you use the `convert referenced graphics` rule. Note that this rule can only be used as a subrule of a `facet` rule.

For example, assume you want to convert all graphic files to the PICT format. With the following example, the software would create PICT files for every graphic:

```
element "graphic" {
   is fm graphic element;
   writer facet default {
   convert referenced graphics;
   export to file "$(entity).pic" as
      "PICT";
}}
```

Depending on how a graphic element was created in your FrameMaker document, it would export as follows:

For a graphic element:	**On export to markup the software:**
Imported as a markup graphic element that used the `entity` attribute to refer to an external data entity named *`ename`* (where *`ename`* is the name of the entity)...	Writes a graphic file named *`ename`*`.pic`. It also creates a graphic element in markup with *`ename`* as the value of the `entity` attribute, and an entity named *`ename`* that references the graphic file.
Imported as an SGML graphic element that used the `file` attribute to refer to a graphic file named *`fname`* (where *`fname`* is the name of the graphic file)...	Writes a graphic file named *`fname`*`.pic`. It also creates an SGML graphic element with *`fname`* as the value of the `file` attribute.
Created in the FrameMaker document by the author...	Writes a graphic file named `graphic1.pic` (`graphic2.pic`, `graphic3.pic`, etc.). It also creates a graphic element in markup with `graphic1` as the value of the `entity` attribute, and an entity named `graphic1` that references the graphic file.

You can use a similar rule to convert all graphic files of one format to another. With the following example you can convert all TIFF files to PICT:

```
element "graphic" {
   is fm graphic element;
   writer facet "TIFF"{
   convert referenced graphics;
   export to file "$(entity).pic" as
      "PICT";
}}
```

Specifying the entity name on export

Your organization may have common graphics used by all authors. For example, you may have a particular graphic file that contains your company's logo. To facilitate authors using the same graphic for the logo, you can create an element that always points to the same file. In this case, you always want the entity name to be the same for elements of this type.

In the absence of an entity property value for the graphic inset, when the software exports an external data entity for a graphic or equation, it generates a name for the entity based on the element name. You can change the name with the `entity name` rule. The format of the `entity name` rule is:

```
element "gi" {
   is fm graphic_or_equation element ["fmtag"];
   writer type ["name"] entity name is "ename";
}
```

where *gi* is a generic identifier; *graphic_or_equation* is one of the keywords `graphic` or `equation`; *fmtag* is an optional FrameMaker element tag; *type* is one of the keywords `anchored frame`, `facet`, or `equation`; *name* is a facet name you supply only if *type* is `facet`; and *ename* is the entity name. In situations where there could be more than one entity for a given graphic element, the software ensures unique entity names by appending an integer to the end of the *ename* argument.

Assume authors import a graphic by reference into a FrameMaker element named Logo. Also assume the graphic is in a file named `cologo.tif` in TIFF format, so you could use the rule:

```
element "logo" {
   is fm graphic element;
   writer facet "TIFF" {
      entity name is "cologo";
}}
```

With this rule, the software creates a single instance of the following entity declaration in the markup document's internal DTD subset:

```
<!ENTITY cologo1 SYSTEM cologo1.tif NDATA TIFF>
```

For more information, see "Creating graphic files on export" on page 373. For information on specifying a graphic filename, see "Changing the name of the graphic file on export" on page 384. For more information on these rules, see:

- Developer Reference, page 63: entity name is
- Developer Reference, page 56: element
- Developer Reference, page 113: is fm graphic element
- Developer Reference, page 111: is fm equation element
- Developer Reference, page 43: anchored frame
- Developer Reference, page 65: equation
- Developer Reference, page 74: facet
- Developer Reference, page 166: writer

Changing how FrameMaker writes out the size of a graphic

The default declarations for graphics include both `dpi` and `impsize` attributes with the software using one or the other of these attributes when it exports a single graphic facet. However, you can override the software's default behavior by using the `specify size` rule.

This rule determines which of these attributes the software writes. In addition, it indicates what units to use for `impsize`. If there is no `specify size` rule, FrameMaker uses the `dpi` attribute. In all cases it uses a resolution of 0.001 (to 3 decimal points). The general form of this rule is:

```
element "gi" {
   is fm graphic_or_equation element ["fmtag"];
   writer type ["name"]
      specify size in units;
}
```

where *gi* is a generic identifier; *graphic_or_equation* is one of the keywords `graphic` or `equation`; *fmtag* is an optional FrameMaker element tag; *type* is one of the keywords `anchored frame`, `facet`, or `equation`; *name* is a facet name you supply only if *type* is `facet`; *units* indicates the unit of measure in for the graphic.

FrameMaker reports the size of the imported graphics object in the indicated units. It calculates a value of `impsize` by determining the width and height of the smallest possible rectangle that can contain the imported object using the default resolution and unit type.

For example, assume the graphic is a circle with a diameter of 3.1115 centimeters. Given the rule:

```
specify size in cm;
```

FrameMaker generates the attribute `impsize="3.112cm 3.112cm"`.

For more information on these rules, see:

- Developer Reference, page 155: specify size in
- Developer Reference, page 56: element
- Developer Reference, page 113: is fm graphic element
- Developer Reference, page 111: is fm equation element
- Developer Reference, page 43: anchored frame
- Developer Reference, page 65: equation
- Developer Reference, page 74: facet
- Developer Reference, page 166: writer

24 Translating Cross-References

A *cross-reference* is a passage in one place in a document that refers to another place, a *source*, in the same document or a different document. While markup does not explicitly support cross-references, it does provide the declared values `ID`, `IDREF`, and `IDREFS` for attributes; and attributes using these declared values customarily represent cross-references. FrameMaker can also use this model for cross-references within a FrameMaker document.

There are several differences between the FrameMaker cross-reference mechanism and the customary way of interpreting the related markup attributes. For information on these differences, see "Cross-references" on page 96.

In this chapter

This chapter starts by describing the default translation provided by FrameMaker. What FrameMaker does by default differs depending on whether you start from an EDD or from a DTD and whether you are working with SGML or XML. The chapter then describes how to modify the default behavior. It contains these sections.

Default translation

On both import and export, FrameMaker assumes that elements having an attribute with a declared value of `ID` can be the source of a cross-reference. It also assumes markup language elements with declared content `EMPTY` that have an attribute with a declared value of `IDREF` are for cross-references. So, by default, when processing a single markup document, FrameMaker

document, or FrameMaker book, FrameMaker automatically translates EMPTY markup language elements that use an IDREF attribute to or from FrameMaker cross-reference elements.

FrameMaker interprets internal and external cross-references in multiple-file markup documents and FrameMaker books in special ways:

- When you export a FrameMaker *book* to markup, it becomes a single document instance in markup. Similarly, when you import a markup *document* to a FrameMaker book, it becomes multiple FrameMaker documents. Because of this, cross-references that are external in FrameMaker may be internal to a single document instance. For a discussion of the importance of the distinction between internal and external cross-references, see "Cross-references" on page 96.
- When you export a *single* FrameMaker document to markup, FrameMaker treats all cross-references that were external in FrameMaker as external in markup. However, if you export an *entire* FrameMaker book to markup, treatment of the individual FrameMaker documents that make up the book is more complicated: cross-references between FrameMaker documents in the book are considered internal cross-references in the markup—only those cross-references to documents not in the FrameMaker book are considered external.

For example, assume your FrameMaker book, manual.book, contains the files ch1.fm and ch2.fm and that there is an external cross-reference in ch1.fm to ch2.fm. When you export manual.book to markup, it creates one document instance and the external cross-reference from ch1.fm to ch2.fm becomes an internal cross-reference in this document instance. If you import this markup document back into FrameMaker (thus again creating multiple files), the cross-reference again becomes an external cross-reference. For information on exporting FrameMaker books, see Chapter 29, "Processing Multiple Files as Books."

On export to markup

When creating a DTD from an EDD, FrameMaker translates ID attributes used for cross-reference sources in the same way as any other attributes. FrameMaker translates a cross-reference element as an empty markup element of the same name. The element's attributes are exported as well. To simplify export, your cross-reference element should have an IDREF attribute.

When creating a DTD, for each cross-reference element, FrameMaker creates an additional impliable attribute, format, with the declared value CDATA. When exporting a FrameMaker document, the value of this attribute is set to the name of the cross-reference format used by the cross-reference. This attribute corresponds to the property cross-reference format in read/write rules. Furthermore, in XML DTDs, FrameMaker also creates an impliable CDATA attribute called srcfile. This attribute is used for external cross-references and cannot be modified by read/write rules.

When exporting a document, FrameMaker exports ID and IDREF attributes as it does any other attributes. Thus, if you have used an element for the source of your cross-reference and an IDREF attribute on the cross-reference element, the export process works simply to provide a natural markup representation.

If you define a cross-reference element without an IDREF attribute, then the software uses an internal mechanism for storing cross-reference information. It does not export this information to markup.

Whether or not the cross-reference element has an IDREF attribute, if the source of the cross-reference is not an element, then FrameMaker exports the cross-reference element as text. That is, the fact that an element was present in the FrameMaker document is lost; instead, the text content provided by the cross-reference format appears in the markup document.

If the cross-reference source is external to the file when exporting a single file or external to the book when exporting a book, then FrameMaker exports the cross-reference to SGML as text. It exports such a cross-reference to XML as a cross-reference element, setting its srcfile attribute to the URI of the file containing the source element with the ID of the source element as a fragment identifier. In other words, the value of the srcfile attribute is the URI of the source file followed by a # delimiter and the value of the ID attribute. See "Translating external cross-references to and from XML" on page 396 for examples.

A cross-reference source that isn't an element is exported in the same way as any other FrameMaker non-element marker. Its corresponding cross-reference exports as text in the way described above.

For more information on FrameMaker's representation of cross-references, see "Using UniqueID and IDReference attributes" on page 195 of this manual and see the FrameMaker and FrameMaker user's manuals.

On import to FrameMaker

When creating an EDD from a DTD, FrameMaker creates a cross-reference element when it encounters an element declaration with declared content EMPTY that has an attribute with the declared value IDREF or IDREFS. When it encounters any other element declaration with an attribute with declared value IDREF or IDREFS, it creates an element and treats the attribute as an ordinary attribute. When it encounters an attribute with the declared content ID, it creates a FrameMaker UniqueID attribute.

When importing a markup document, FrameMaker creates the appropriate attribute in FrameMaker if it encounters an attribute with the declared value ID. When it encounters an element with a single attribute with the declared value IDREF and the declared content EMPTY, it translates the element to a cross-reference element. If your markup document is invalid in that it does not include both the IDREF attribute and its corresponding ID attribute, this process results in an unresolved cross-reference. If your XML document also defines a srcfile attribute for such an element, FrameMaker interprets its value as that of the URI of the cross-reference source.

When importing a markup document, FrameMaker creates a cross-reference element when it encounters an element with a single attribute whose declared value is IDREFS and declared content is EMPTY. The software uses the first value in the IDREFS attribute as the ID of the cross-

reference. It saves the other values, to write out on export, but does nothing special with them. You must write a structure API client to change this behavior.

Other markup elements may be intended to represent cross-references. For example:

- SGML elements that have attributes with the declared values `NAMES`
- Elements with an `IDREFS` attribute
- Elements that have multiple attributes with the declared value `IDREF`
- Elements that are not `EMPTY` yet have attributes with the declared value

You can write a structure API client or an XSLT style sheet to handle these situations.

Modifications to the default translation

The following sections describe some of the possible modifications to the default translation of cross-references. You may require different rules or a structure API client, or you may use these rules in ways not discussed in these sections. For additional ways to modify the translation of cross-references, see the information at the end of each section.

For a summary of read/write rules relevant to translating cross-references, see Developer Reference, page 35: Cross-references. For information on writing structure API clients, see *Structure Import/Export API Programmer's Guide*.

Translating markup elements as FrameMaker cross-reference elements

You can identify which markup elements correspond to FrameMaker cross-reference elements and choose to use the same name or different names for the markup and FrameMaker elements. The rules for doing this apply on both import and export to create elements of the appropriate type. In general, you use a rule of this form:

```
element "gi" is fm cross-reference element fmtag;
```

where *gi* is a generic identifier and *fmtag*, if specified, indicates the name of the corresponding FrameMaker element. For example, to indicate that both of the markup elements `xref` and `link` correspond to FrameMaker cross-reference elements, you can use these rules:

```
element "xref" is fm cross-reference element "XRef";
element "link" is fm cross-reference element;
```

For information on the rules used in this example, see Developer Reference, page 56: element and Developer Reference, page 109: is fm cross-reference element.

Specifying a cross-reference element in Schema

In order for FrameMaker to translate a Schema element to a cross-reference upon import of an XML file associated with a Schema declaration, you must use element definitions and rules such as the following:

Schema element definition for a cross-reference

```
<xs:element name="CrossRef"><xs:complexType><xs:complexContent>
<xs:restriction base="xs:anyType">
   <xs:attribute name="Id" type="xs:ID"/>
   <xs:attribute name="myidref" type="xs:IDREF"/>
   <xs:attribute name="format" type="xs:string"/>
   <xs:attribute name="srcfile" type="xs:string"/>
</xs:restriction>
</xs:complexContent></xs:complexType></xs:element>
```

Read/write rule for a cross-reference

```
element "CrossRef"
{
is fm cross-reference element "CrossRef";
attribute "myidref" is fm property cross-reference id;
attribute "format" is fm property cross-reference format;
/*attribute "srcfile" is fm property cross-reference srcfile;*/
}
```

For information on the rules used in these examples, see Developer Reference, page 56: element, Developer Reference, page 109: is fm cross-reference element, and Developer Reference, page 125: is fm property value.

Renaming the markup attributes used with cross-references

(For details, see "On export to markup" on page 391) FrameMaker creates the `format` attribute for working with cross-references but you can choose a different attribute for the same purpose. You can do so either at the highest level to set a default or within an `element` rule for a specific markup element. Use the `is fm property` rule within an `attribute` rule either at the highest level or within an `element` rule. That is, use a rule of one of these forms:

```
attribute "attr"
   is fm property cross-reference format;

element "gi"
   attribute "attr"
      is fm property cross-reference format;
```

where *gi* is a generic identifier and *attr* is a markup attribute.

For example, to specify that all relevant markup elements use the `fmform` attribute to specify a FrameMaker cross-reference format, but that the markup `exref` element uses the `exform` attribute, you would use these rules:

```
attribute "fmform" is fm property cross-reference format;

element "exref" {
   is fm cross-reference element "CrossRef";
   attribute "exform" is fm property cross-reference format;
}
```

Instead of translating the cross-reference format as an attribute, you may choose to use the `fm property` rule to explicitly set the property value. You can use the `fm property` rule for this purpose.

You can use the `is fm property rule` to specify an attribute to use as a cross-reference Id. For example, if the attribute `linkend` stores the `IDREF` for a markup element, you can set that value to be the FrameMaker cross-reference ID property with the following rule:

```
attribute "linkend" is fm property cross-reference id;
```

On export, instead of writing the cross-reference Id to the `IDREF` attribute of the element, the software will write that value to the `linkend` attribute.

For information on the rules used in this example, see

- Developer Reference, page 56: element
- Developer Reference, page 46: attribute
- Developer Reference, page 116: is fm property
- Developer Reference, page 109: is fm cross-reference element.

Translating FrameMaker cross-reference elements to text in markup

You may not want your FrameMaker cross-reference elements to remain elements in the markup representation—you can choose to translate them to text instead. To do so, use this rule:

```
fm element "fmtag" unwrap;
```

where *fmtag* is the FrameMaker element tag for a cross-reference element.

For more information on these rules, see Developer Reference, page 77: fm element and Developer Reference, page 160: unwrap.

Maintaining attribute values with FrameMaker

FrameMaker can maintain values for `ID` and `IDREF` attributes so that your end users do not need to keep track of the values. If you want to let the software do so, you may choose also to prohibit your end users from changing the values of these attributes.

For information on how to maintain this control, see "Creating read-only attributes" on page 309.

Translating external cross-references to and from XML

The source of an internal cross-reference is completely identified by the value of its `ID` attribute. Hence, internal cross-references are specified with `IDREF` attributes. To identify the source of an external cross-reference, though, the file name of the containing document is also needed. When FrameMaker saves a document as SGML, it avoids the need to export a file name by converting external cross-references to text (see "On export to markup" on page 391).

When FrameMaker saves a structured document to XML, it preserves cross-reference elements for external cross-references, using the `srcfile` attribute to export the source's file name. Within FrameMaker, an external cross-reference links one FrameMaker document to another FrameMaker document. An exported cross-reference can refer either to the FrameMaker document that was the original cross-reference source, or to a corresponding XML document. For example, suppose the source of an external cross-reference is an element with its `ID attribute set to BAAHDJJE` that occurs in the file `c:\myproject\manual\intro.fm`. If the name of the cross-reference element is `xref` and the cross-reference format is not exported, by default FrameMaker exports the cross-reference as:

```
<xref srcfile="file:///C:/myproject/manual/intro.fm#BAAHDJJE">
```

If you save the file containing the cross-reference source to XML, you can choose instead to set the `srcfile` attribute to refer to the XML version of that file. You specify this option in the application definition in `structapps.fm` or another application definition file rather than with read/write rules.

25 Translating Variables and System Variable Elements

You use variables in FrameMaker documents to store information that may change at some later time, such as a product's name; information you know will change, such as the current date; or text that you must enter frequently. Variables make it easier for you to manage these changes.

In markup, you can use either elements or entities for similar purposes. Some of the material in this chapter is closely related to the handling of entities. For more information on entities, see Chapter 22, "Translating Entities and Processing Instructions."

In this chapter

This chapter starts by describing FrameMaker's default translation of variables. The chapter then describes modifications you can make to the default behavior. Some of these procedures are relevant when translating in both directions; others are relevant only in one direction.

This chapter contains these sections.

Default translation

Markup has no unique representation for variables. There are two types of FrameMaker variables:

- User-defined variables provide an easy way to store information that may change. For example, in an insurance policy document, you might represent the name of the insured person with the variable `Insured`, which FrameMaker replaces with the name of the insured person for a particular policy.
- System variables are used to insert system specific calculations, such as the current date or the current page number. You cannot modify system variables.

In addition, FrameMaker provides system variable elements. These elements are reflected in the structure of the document. You use them for the same purposes as system variables.

For more information on FrameMaker variables and system variable elements, see the FrameMaker user's manual.

On export to markup

FrameMaker translates a system variable element as an empty markup element. It does not record the definition of the corresponding variable, because that is formatting information for the element.

When writing a markup document, FrameMaker bases its treatment of non-element variables on the variable text. Unless instructed otherwise, it translates a user variable to a reference to an entity with the same name as the variable. If no entity definition with the given name exists, FrameMaker creates a new entity definition using the variable name and variable text. If an entity definition with the given name does exist, FrameMaker writes a message to the log file warning of a potential mismatch.

If the variable name is not a valid name in markup, FrameMaker uses `fmv1` as a default entity name. If an entity already exists by that name, FrameMaker increments the counter until an unused name is found, for example, `fmv2`, `fmv3`, and so forth.

FrameMaker determines the type of entity on the basis of special character formats within the variable text. If the variable text uses the `FmCdata` character format, FrameMaker exports the variables as a `CDATA` entity. If the variable text uses the `FmSdata` character format, FrameMaker exports it as an `SDATA` entity. In the absence of any relevant character format information and markup, FrameMaker exports the variable as a text entity. If the variable text contains markup, then FrameMaker exports the variable as a `CDATA` entity.

XML: The XML standard does not allow `SDATA` and `CDATA` entities. If your document contains variables that would translate to either of these entity types, FrameMaker exports them as text, and the use of the variable will be lost the next time you import the XML data.

To retain the use of variables, you should map FrameMaker variables to specific XML elements. (See "Translating markup elements as system variable elements" on page 402.) Also, you should never include markup in variables if you intend to export your document to XML.

SGML: FrameMaker reports special characters within `CDATA` entities as errors. For SGML text, external data, and `CDATA` entities, the entity text is the same as the variable text. For `SDATA` entities, FrameMaker uses the string `"FM variable"` as the parameter literal.

FrameMaker translates a non-element system variable as a reference to an entity in SGML with one of the following names:

System variable	Entity
Page Count	`fm.pgcnt`
Current Date (Long)	`fm.ldate`

System variable	Entity
Current Date (Short)	`fm.sdate`
Creation Date (Long)	`fm.lcdat`
Creation Date (Short)	`fm.scdat`
Modification Date (Long)	`fm.lmdat`
Modification Date (Short)	`fm.smdat`
Filename (Long)	`fm.lfnam`
Filename (Short)	`fm.sfnam`
Table Continuation	`fm.tcont`
Table Sheet	`fm.tsht`

For more information on FrameMaker's treatment of entities, see Chapter 22, "Translating Entities and Processing Instructions."

On import to FrameMaker

By default, FrameMaker does not create system variable elements when creating an EDD from a DTD. It creates user variables for entities of various sorts. For information on how FrameMaker translates entities by default as variables, see "On import to FrameMaker" on page 316.

Modifications to the default translation

The following sections describe some possible modifications to the default translation of FrameMaker variables and system variable elements. You may require different rules or a structure API client, or you may use these rules in ways not discussed in these sections. For additional ways to modify the translation of variables and system variable elements, see the cross-references at the end of each procedure and in Chapter 22, "Translating Entities and Processing Instructions."

For a summary of read/write rules relevant to translating variables and system variable elements, see Developer Reference, page 41: Variables and Developer Reference, page 36: Entities. For information on writing structure API clients, see the *Structure Import/Export API Programmer's Guide*.

Renaming or changing the type of entities when translating to variables

By default, FrameMaker converts non-element system variables and user variables to entity references. You can change the name of the entity to use for a variable with the `entity` rule. The format of this rule is:

```
entity "ename" is fm variable "fmvar";
```

where *ename* is a markup entity and *fmvar* is a FrameMaker variable. With this rule, FrameMaker creates a reference to an entity of the appropriate type, on the basis of character formats associated with the variable text. It also creates an entity declaration if one doesn't already appear in the internal DTD subset. For information on how character formats affect the translation of variables on export, see "On export to markup" on page 399.

You can change the type of entity to which a variable translates by changing the character format associated with the variable text in the variable definition.

For example, assume you have the FrameMaker variable `Start Tag Ex` with the variable text `<tag>` and that you want to translate it to the SGML `CDATA` entity `startex`. To do so, use this rule:

```
entity "startex" is fm variable "Start Tag Ex";
```

Be sure that `Start Tag Ex` uses the CDATA character format in its definition. That is, the definition of the variable must be:

```
<CDATA>\<tag\><Default ¶ Font>
```

If `startex` is not declared in the DTD, FrameMaker inserts the following entity declaration into the DTD subset when it encounters the first instance of the variable `Start Tag Ex` in the document being processed:

```
<!ENTITY startex CDATA "<tag>">
```

SGML: If you want the variable to become an `SDATA` entity instead, change the variable definition to:

```
<SDATA>\<tag\><Default ¶ Font>
```

In this case, if `startex` is not declared in the DTD, FrameMaker inserts this entity declaration:

```
<!ENTITY startex SDATA "<tag>">
```

If the DTD already has an appropriate entity declaration, FrameMaker retains that declaration.

In a similar circumstance, assume you map a text entity named `product` to a FrameMaker variable, `Product Name`. If you have the following text entity declaration:

```
<!ENTITY product "Not yet named">
```

then with the rule:

```
entity "product" is fm variable "Product Name";
```

FrameMaker produces a reference to the general entity, but doesn't create a new entity declaration.

For information on the rules used in these examples, see Developer Reference, page 61: entity and Developer Reference, page 139: is fm variable.

Translating markup elements as system variable elements

You can translate some markup elements as FrameMaker system variable elements. To do so, use the following rule:

```
element "gi" is fm system variable element ["fmtag"];
```

where *gi* is a generic identifier and the optional argument *fmtag* is a FrameMaker element tag corresponding to a system variable element. For example, to translate a markup element `date` as a system variable element of the same name, you could use this rule:

```
element "date" is fm system variable element;
```

To rename the element on import and export, you could use this rule:

```
element "date" is fm system variable element "TodaysDate";
```

This rule translates the markup element `date` as the FrameMaker system variable element `TodaysDate`. You cannot use a read/write rule to specify which system variable to associate with the element, because this association is considered formatting information. Instead, you must add appropriate format rules to the EDD. For example, you could have this definition in your EDD:

Element (System Variable): TodaysDate
System variable format rule
1. In all contexts.
   Use system variable: Current Date (Long)

In this case, `TodaysDate` always translates to the system variable Current Date (Long).

For information on these rules, see Developer Reference, page 56: element and Developer Reference, page 131: is fm system variable element.

Translating FrameMaker system variable elements to text in markup

You may not want your FrameMaker system variable elements to remain elements in the markup representation. You can choose to translate them to text in markup instead. To do so, use this rule:

```
fm element "fmtag" unwrap;
```

where *fmtag* is the FrameMaker element tag for a system variable element.

For more information on these rules, see Developer Reference, page 77: fm element and Developer Reference, page 160: unwrap.

Translating FrameMaker variables as SDATA entities

For SGML, you can translate FrameMaker variables as `SDATA` entities by using the `entity` rule or by manipulating parameter literals. For information on how to do so, see "Translating entities as FrameMaker variables" on page 325.

Discarding FrameMaker variables

FrameMaker always allows you to insert a variable in a document. To modify this behavior, you must use an FDK client. If you don't want FrameMaker to export some or all variables to markup, you can choose to have it discard all variables or particular variables.

To have FrameMaker discard variables, use this rule:

```
fm variable ["var1", . . ., "varn"] drop;
```

Each var_i is a variable. If you don't specify var_i in the rule, the rule applies to all variables not addressed explicitly by `entity` or other `fm variable` rules. It is an error if the same var_i appears in multiple `entity` or `fm variable` rules.

This rule always occurs in a highest-level rule, because it applies to all instances of the indicated variables. It does not apply to system variable elements.

For information on the rules used in this example, see Developer Reference, page 166: writer, Developer Reference, page 91: fm variable, and Developer Reference, page 53: drop.

26 Translating Markers

You use markers in FrameMaker documents to store various kinds of information you don't want visible to the document's audience. Although there's no analogous concept in markup, markers frequently correspond to various attributes or elements.

FrameMaker provides a variety of marker types. Two of these marker types have special default translations.

- DOC PI markers store information about some processing instructions, and DOC Entity Reference markers store information about entity references.
- DOC Comment markers are used to store comments when FrameMaker imports an XML document into a FrameMaker document. For information on this use of markers, see Chapter 22, "Translating Entities and Processing Instructions." You can change which marker type FrameMaker uses to store this information.
- Conditional Text markers indicate conditional text. For information on the treatment of conditional text on export, see Chapter 28, "Translating Conditional Text.".

In this chapter

This chapter describes the default translation of markers and modifications you can make to the default behavior. Some of these procedures are relevant when translating in both directions; others are relevant only in one direction.

This chapter contains these sections.

Default translation

Markup has no special representation for markers. FrameMaker allows you to represent markers either as marker elements or as non-element markers.

On export to markup

FrameMaker exports a marker element as an markup empty element of the same name, with two additional attributes, `text` and `type`. The value of the `text` attribute is the marker text; the value of the `type` attribute is the marker type.

> **Note: XML and SGML:** The XML specification states that closing delimiter string for a PI is `?>`. The SGML specification simply uses `>`. Unless otherwise stated, the examples of PIs in this section use the XML syntax.

FrameMaker exports non-element markers, other than those of Type `DOC PI`, as processing instructions of the following form:

```
<?FM MARKER [type] text?>
```

where *`type`* is the marker type and *`text`* is the marker text. For example, FrameMaker exports an Index marker with the text "translating, PIs" as:

```
<?FM MARKER [Index] translating, PIs?>
```

If a DOC PI marker has marker text of this form:

```
text
```

then FrameMaker outputs this processing instruction:

```
<?text?>
```

If a DOC Entity Reference marker has marker text of this form:

```
entname
```

where *`entname`* is the entity name, then FrameMaker outputs this entity reference:

```
&entname;
```

If a DOC Comment marker has marker text of this form:

```
text
```

then FrameMaker generates this comment:

```
<!-- text -->
```

You can use a rule to change the marker type whose text is treated in this manner.

FrameMaker uses the Cross-Ref marker type to mark the source of a non-element cross-reference. On export to markup, these markers are treated as any other non-element markers are. They are

not treated specially to indicate the source of a cross-reference. For information on exporting cross-references, see Chapter 25, "Translating Cross-References."

On import to FrameMaker

In the absence of read/write rules, FrameMaker cannot identify a markup element that corresponds to a marker. If you have elements you want to translate as markers, you must write rules.

However, if you start with an EDD instead of a DTD, the default DTD translates FrameMaker marker elements as markup elements with a declared content of `EMPTY`. If you have a markup document that uses this DTD, and your application specifies a FrameMaker template to use on import, then it translates the appropriate markup elements to marker elements when you import the document to FrameMaker.

Also, by default FrameMaker imports some entity references and processing instructions as non-element markers. For information on when this happens, see Chapter 22, "Translating Entities and Processing Instructions."

While importing a markup document, FrameMaker stores comments in a marker of type DOC Comment.

Modifications to the default translation

The following sections describe some modifications to the default translation of markers. You may require different rules or a structure API client, or you may use these rules in ways not discussed in these sections. For additional ways to modify the translation of markers, see the cross-references at the end of each procedure.

For a summary of read/write rules relevant to translating markers, see Developer Reference, page 38: Markers.

Translating markup elements as FrameMaker marker elements

You can identify which markup elements correspond to FrameMaker marker elements, and use the same name or different names for the FrameMaker and markup elements. This approach works for creating elements of the appropriate type on both import and export. In general, you use a rule of this form:

```
element "gi" is fm marker element ["fmtag"];
```

where *gi* is a generic identifier and *fmtag* is a FrameMaker element tag. If *fmtag* is specified, it is the name of the corresponding FrameMaker marker element; otherwise, the FrameMaker element name is the same as the generic identifier.

For information on the rules used in this example, see Developer Reference, page 56: element and Developer Reference, page 115: is fm marker element.

Specifying a marker element in Schema

In order for FrameMaker to translate a Schema element to a footnote or marker upon import of an XML file associated with a Schema declaration, you must use element definitions and rules such as the following:

Schema element definition for a footnote or marker

```
<xsd:element name="Footnote" type="xsd:string"/>
<xsd:element name="Marker" type="xsd:string"/>
```

Read/write rule for a footnote or marker

```
element "Footnote" is fm footnote element "Footnote";
element "Marker" is fm marker element "Marker";
```

For information on the rules used in these examples, see Developer Reference, page 56: element, Developer Reference, page 112: is fm footnote element, and Developer Reference, page 115: is fm marker element.

Writing marker text as element content instead of as an attribute

By default, on export FrameMaker writes the text of a marker element as the value of an attribute of an empty markup element. On import, it finds the marker text in that attribute. Instead, you can choose to have the marker text stored as the content of the corresponding markup element. Note that the markup element cannot be declared as an empty element.

To treat marker text for a specific FrameMaker marker element as content for the markup element, use the `marker text` rule as a subrule of the `element` rule:

```
element "gi" marker text is content;
```

where *gi* is a generic identifier.

For information on the rules used in these examples, see Developer Reference, page 56: element and Developer Reference, page 142: marker text is.

Using markup attributes and FrameMaker properties to identify markers

When translating FrameMaker marker elements to markup elements, by default FrameMaker uses the markup attributes `type` and `text`. These correspond, respectively, to the FrameMaker properties `marker type` and `marker text`. You can choose to have FrameMaker use different attribute names or to not use one or both of the attributes altogether.

To have FrameMaker not use one of these attributes, remove the attribute from the definition of the corresponding markup element in the DTD, or drop the attribute on import. When exporting a FrameMaker document, if the corresponding markup element does not have one of these attributes defined, the software does not write a value for the attribute.

To rename these properties or attributes, use the following rule:

```
element "gi" attribute "attr" is fm property prop;
```

where *gi* is a generic identifier, *attr* is a markup attribute and *prop* is one of `marker type` or `marker text`. For example, to have the markup element `index` become a marker element of type Index and get its text from the `term` attribute, you could use the following rule:

```
element "index" {
   is fm marker element;
   attribute "term" is fm property marker text;
}
```

and this element definition:

Element (Marker): Index
Initial marker type
1. In all contexts.
Use marker type: Index

With this rule and declaration, FrameMaker translates an `Index` element to a start-tag of the form:

```
<index term="Some index text">
```

For information on the rules used in this example, see:

- Developer Reference, page 56: element
- Developer Reference, page 115: is fm marker element
- Developer Reference, page 80: fm property
- Developer Reference, page 53: drop
- Developer Reference, page 46: attribute
- Developer Reference, page 116: is fm property

Discarding non-element FrameMaker markers

FrameMaker always allows you to insert a non-element marker in a document. Modifying this behavior requires an FDK client. However, if you don't want FrameMaker to export some or all non-element markers to markup, you can choose to have it discard either all non-element markers or non-element markers of certain types.

To discard non-element markers on export of a FrameMaker document, use this rule:

```
fm marker ["type1", . . ., "typen"] drop;
```

Each *type1* is a marker type. A particular *type1* can appear in only one rule. If there is a rule with no *type1* specified, it provides a default for marker types not explicitly listed in another rule.

You may choose to drop most non-element markers and retain only certain types as processing instructions. In this case, you could write rules to translate individual types as processing

instructions, followed by a rule to drop all non-element markers. For example, to drop all non-element markers except Index and Hypertext markers, use these rules:

```
fm marker "Index" is processing instruction;
fm marker "Hypertext" is processing instruction;
fm marker drop;
```

Since FrameMaker uses the first rule that matches in any given situation, the order of these rules is important. If the rules occurred in this order:

```
fm marker drop;
fm marker "Index" is processing instruction;
fm marker "Hypertext" is processing instruction;
```

then FrameMaker would drop all non-element markers, including Index and Hypertext markers.

For information on the rules used in this example, see Developer Reference, page 78: fm marker, Developer Reference, page 53: drop, and Developer Reference, page 140: is processing instruction. For information on writing FDK clients, see *FDK Programmer's Reference*, an online manual supplied with the Frame Developer's Kit.

27 Translating Conditional Text

Conditional text is a FrameMaker mechanism for specifying portions of a document that can be included or omitted as needed. For information on working with conditional text, see the *FrameMaker User Guide*.

> **Note: SGML** When FrameMaker exports a structured document to SGML, it exports visible conditional text, but does not indicate that the text was conditional. It does not export hidden conditional text.

FrameMaker can represent both condition settings and the application of condition tags in XML in a way that permits round-tripping of this information. The remainder of this chapter therefore applies only to XML.

In this chapter

This chapter starts by describing the default translation provided by FrameMaker. The chapter then describes how to modify the default behavior. It contains these sections.

Default translation

To preserve conditional text in XML, two types of information are needed:

- Condition settings, including the condition tags defined in the FrameMaker document as well as the condition indicators associated with each and whether the condition is shown or hidden
- Condition tags, if any, associated with each portion of the document's content

Neither type of information is stored in elements in the FrameMaker document. Condition settings are independent of the document's element structure and one or more condition tags can be applied to an element or to all or part of its content. FrameMaker therefore uses processing instructions to represent conditional text in XML.

Condition settings

A separate processing instruction at the beginning of the XML document defines each condition tag. This processing instruction has the form:

```
<?Fm Condition tag color style status?>
```

where

- *tag* is a condition tag
- *color* is the condition indicator color (AsIs if no color is specified)
- *style* is the condition indicator style, represented by one of the following keywords:

Keyword	Condition Indicator (as defined in the Style pop-up menu of the Edit Condition Tag dialog box)
NO_OVERRIDE	As Is
OVERLINE	Overline
STRIKETHROUGH	Strikethrough
SINGLE_UNDERLINE	Underline
DOUBLE_UNDERLINE	Double Underline
CHANGEBAR	Change Bar
NUMERIC_UNDERLINE	Numeric Underline
NMRIC_AND_CHNGBAR	Numeric Underline and Change Bar

- *status* is one of the keywords hide or show to indicate the show/hide status of the condition tag

For example, the following processing instructions define conditions tagged Summer and Winter:

```
<?Fm Condition Summer Blue OVERLINE hide?>
<?Fm Condition Winter Red SINGLE_UNDERLINE show?>
```

Conditional text

A conditional portion of an XML document's content is delimited by processing instructions that identify the applied condition tag. The processing instruction that precedes the conditional text has the form:

```
<?Fm Condstart tag?>
```

while the one following the conditional text has the form:

```
<?Fm Condend tag?>
```

where *tag* is the condition tag. If multiple condition tags apply to the same content, each tag has separate processing instructions.

For example, if a list of recipes uses condition tags to identify recipes with particular ingredients, the list can be customized for people with various food allergies. Such a document might contain markup such as:

```
<?Fm Condstart fruit?>
<?Fm Condstart blueberry?>
<recipe>Blueberry Muffins</recipe>
<?Fm Condend blueberry?>
<?Fm Condstart apple?><?Fm Condstart orange?>
<recipe>Apple Orange Fruit Cup</recipe>
<?FM Condend orange?><?Fm Condend apple?>
<?Fm Condend fruit?>
```

On export to markup

By default, FrameMaker exports processing instructions that define all condition settings. When exporting a book, FrameMaker exports the condition settings for each book component after the start-tag for the book component.

FrameMaker exports all conditional text, with surrounding processing instructions, regardless of whether the conditional text is shown or hidden.

When hidden conditional text is exported, the result may not be a valid XML document. Suppose, for example, that a course catalog begins with a `Title` element, and that only one `Title` is permitted. Class offerings for different times of year might be edited in one conditional document. The author might edit the titles for different versions as conditional text within a single `Title` element or as different conditional `Title` elements. If the author takes the first approach, a valid XML document might include the following fragment:

```
<Title>
<?Fm Condstart Summer?>Summer<?Fm Condend Summer?>
<?Fm Condstart Winter?>Winter<?Fm Condend Winter?>
Course Catalog
</Title>
```

However, the author might choose to enter different complete `Title` elements, making each of them conditional. In this case, the exported XML document might include the following:

```
<?Fm Condstart Summer?>
<Title>Summer Course Catalog</Title>
<?Fm Condend Summer?>
<?Fm Condstart Winter?>
<Title>Winter Course Catalog</Title>
<?Fm Condend Winter?>
```

Since this fragment contains two `Title` elements where only one is permitted, the XML document is not valid. If you expect such usage in your environment, you might want to permit multiple `Title` elements in your DTD. If you expect only one `Title` to be shown at a time, the

EDD can still permit only a single `Title`. The use of slightly different models in FrameMaker and XML allows FrameMaker documents to be validated prior to publishing to confirm that the show/hide settings produce a document that conforms to the permitted structure. At the same time, it supports export of hidden conditional text to valid XML.

On import to FrameMaker

When FrameMaker opens an XML document containing processing instructions for conditional text, it sets condition settings according to the encountered `<?Fm Condition?>` processing instructions. If multiple processing instructions refer to the same condition tag, FrameMaker uses the first definition. If opening the XML document creates a FrameMaker book, FrameMaker uses the first processing instruction for each condition tag after the start-tag for each book component. Thus, a condition tag can have different condition settings in different book components.

If you import an XML document into an existing FrameMaker document, condition settings in the FrameMaker document have priority over any defined in the imported XML document. In particular, when creating an XML text inset, FrameMaker ignores any `<?Fm Condition?>` processing instructions for existing condition tags but defines new condition tags for other `<?Fm Condition?>` processing instructions.

FrameMaker applies the condition tags specified in `<?Fm Condstart?>` and `<?Fm Condend?>` to the indicated content.

Modifications to the default translation

When FrameMaker imports an XML document, it interprets all conditional text processing instructions it encounters. For export, you can control:

- Whether hidden conditional text is exported
- Whether processing instructions delimit conditional text, that is, whether `<?Fm Condstart?>` and `<?Fm Condend?>` processing instructions are written

You control these options in the application definition in `structapps.fm` or another application definition file rather than with read/write rules. For information on the application definition see Developer Reference, page 13: Specifying conditional text output.

28 Processing Multiple Files as Books

FrameMaker provides a mechanism for grouping multiple FrameMaker documents into a single unit called a book. Each document remains a separate, complete FrameMaker document, but FrameMaker provides a set of facilities for working with those documents as a unit. For example, you can easily number pages consecutively throughout the book, generate a table of contents for the entire book, print the entire book with one command, or validate the element structure for the entire book.

Markup doesn't explicitly provide such a mechanism. However, markup documents are sometimes divided into external text entities so that a large document can be distributed over several files.

This use of text entities as include files is analogous to a FrameMaker book file, but markup allows more freedom in specifying the files that correspond to a book:

- A FrameMaker book can have only one level of documents. A markup document can have more than one level of nesting.
- A document in a FrameMaker book can either be unstructured or a single, complete element. An external text entity can include a partial element.

Because of this difference, if you start with a markup document that doesn't match the FrameMaker model for books, the entity structure will not correspond to the book structure. On the other hand, any valid FrameMaker book can be easily translated to markup with each book component exported as a text entity.

In this chapter

> **Note: XML and SGML:** The XML specification indicates that closing delimiter string for a PI is `?>`. Unless otherwise stated, the examples of PIs in this section use the XML syntax.

This chapter starts by describing the default translation for books. The chapter then describes your options for modifying the translation. It contains these sections.

Default translation

Books in FrameMaker and their counterparts in markup don't have special structure associated with them in the EDD or DTD. For this reason, creating an EDD or a DTD doesn't add book-specific information. All translation is done during conversion of individual FrameMaker books or markup documents.

On import to FrameMaker

FrameMaker does not attempt to automatically subdivide a markup document into FrameMaker documents in a book. However, you can use processing instructions or read/write rules to signal that a file is a book file and to signal the start of each new document in the book.

> **Note: XML and SGML:** The XML specification defines the PI closing delimiter as `?>`, while in SGML the closing delimiter is `>`. Also, for XML the colon is omitted. For example, following are two processing equivalent instructions—in SGML, `<?FM: book>`; in XML, `<?FM book?>`.
>
> This section uses the XML specification to illustrate PI syntax.

To instruct FrameMaker to generate a book, you can place the following processing instruction before the start-tag of the document element of a markup document:

```
<?FM book?>
```

The book processing instruction may occur before, within, or after the DTD. It is an error if it occurs elsewhere, if it is preceded by a document processing instruction (described next), or if it occurs more than once.

Within the markup document, you indicate the start of a new document in a FrameMaker book by placing a processing instruction immediately before the start-tag of the element that is the highest-level element in the FrameMaker document. The processing instruction has the form:

```
<?FM document "fname"?>
```

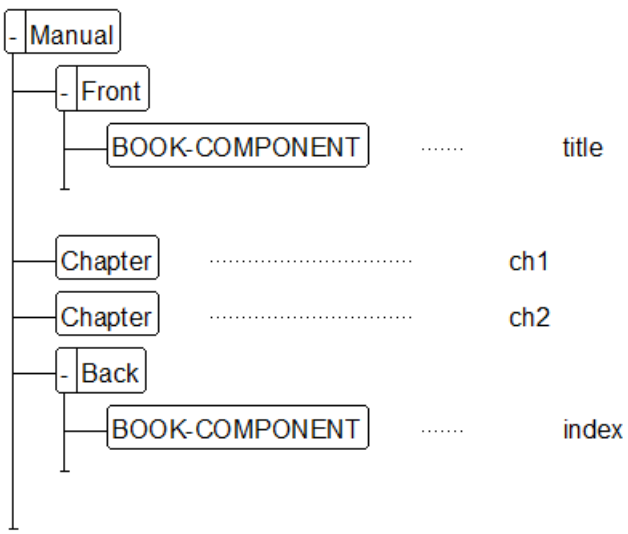
where *fname* is the name of the new document file. If *fname* is a relative pathname, it is relative to the directory for the book file. Any spaces within *fname* must be escaped by a backslash. If *fname* contains the processing instruction close delimiter string (specified as `>` in the SGML reference concrete syntax, and as `?>` in XML), the processing instruction must be entered through an entity. On export, FrameMaker generates *fname* from the name of the file it is processing.

You cannot omit the end-tag for an element that immediately precedes a processing instruction, even if markup minimization is allowed (in SGML). If you did so, the parser would place the omitted end-tag before the start-tag for the next element. Since this start-tag appears after the processing instruction, the end-tag generates an error in the log file.

For example, assume you have a book broken into four parts—the front matter, two chapters, and an index. In FrameMaker, each part is in a separate document. In SGML this situation might be represented as:

```
<!DOCTYPE manual SYSTEM "manual.dtd"
<?FM boo?>
<manual>
<?FM document "title"?>
<front>
. . .
</front>
<?FM document "ch1"?>
<chapter>
. . .
</chapter>
<?FM document "ch2"?>
<chapter>
. . .
</chapter>
<?FM document "index"?>
<back>
. . .
</back>
</manual>
```

In FrameMaker this SGML structure appears as:



On export to markup

FrameMaker treats documents in a FrameMaker book as external text entities. In addition to the file it creates for the entire book (the document entity), it creates a separate file for each structured document. It also creates separate files for unstructured documents as necessary and generates appropriate entity references for these additional files. FrameMaker also produces the processing instructions that allow it to recreate the original book structure.

In particular, FrameMaker names the entities corresponding to documents in a book `bkc1`, `bkc2`, and so on. If an entity name conflicts with the name of an existing entity, FrameMaker increments the counter and tries again. For example, if your DTD already defines an entity `bkc1`, FrameMaker tries instead to name the first book component entity `bkc2`.

Each book component entity declaration includes an external identifier containing a system identifier derived from the document name by dropping any extension and adding a new extension of the form `.e01`, `.e02`, `.e03`, and so forth. If there are more than 100 book components, the extension has the form `.001`, `.002`, `.003`, and so forth.

For example, assume you have the book file with the structure shown earlier. On export, FrameMaker generates this SGML document for that structure:

```
<!DOCTYPE manual . . . [
<!--Begin Document Specific Declarations-->
<!ENTITY bkc1 SYSTEM "title.e01">
<!ENTITY bkc2 SYSTEM "ch1.e02">
<!ENTITY bkc3 SYSTEM "ch2.e03">
<!ENTITY bkc4 SYSTEM "index.e04">
<!--End Document Specific Declarations-->
. . . other local entity declarations . . .]
]>
<?FM book?>
<manual>
<front>
&bkc1;
</front>
&bkc2;
&bkc3;
<back>
&bkc4;
</back>
</manual>
```

Each text entity starts with the appropriate document processing instruction. Thus, the entity `bkc2` has the form:

```
<?FM document ch1?>
<chapter>
. . .
</chapter>
```

When FrameMaker exports a book, it creates a new file for each structured document. Unstructured documents appear as text within an element.

In general, you can use a different EDD for each document in a FrameMaker book. However, if you export the book to markup, the result will be invalid unless the DTD provides a superset of the element declarations used in all the EDDs.

Modifications to the default translation

FrameMaker provides a small number of read/write rules specific to translating books. In addition, there are rules relevant to translating files within books. If you need to make other changes to the translation, you write a structure API client.

To specify the handling of unstructured documents, you specify the handling of the containing element.

- To discard a document, use the `drop` rule described in Developer Reference, page 53: drop.
- To import or export a document as an empty element, use the `drop content` rule described in Developer Reference, page 55: drop content.

For a summary of read/write rules relevant to translating books and book components, see Developer Reference, page 35: Books.

Using elements to identify book components on import

The default method to specify how a markup document should be broken into components of a FrameMaker book is with the processing instructions described above. However, you may be able to use rules for this purpose instead.

If you can identify elements in your DTD that correspond to where you want file breaks in FrameMaker, you can use this rule:

```
reader generate book
   put element "gi" in file ["fname"];
```

For example, if you want instances of the `chapter` and `section` elements to indicate new files, you can use these rules:

```
reader generate book {
   put element "section" in file "Sect.fm";
   put element "chapter" in file;
}
```

This will cause occurrences of `section` elements to create files named `Sect1.fm`, `Sect2.fm`, and so on and occurrences of `chapter` elements to create files named `chapter1.doc`, `chapter2.doc`, and so on.

FrameMaker does not create a new file if the element occurs inside an element that already created a new file. For example, if you use the `section` element both to indicate a new file and to indicate sections within a file, you can still use the rules above. A `section` element inside a `chapter` element or another `section` element does not create an additional file.

With this form of the rule, FrameMaker always creates a book for these elements. If you want FrameMaker to create a book only when it is processing documents with particular document elements, use this form of the rule:

```
reader generate book for doctype "dt1" [. . . "dtN"]
   put element "gi" in file ["fname"];
```

For example, if you want FrameMaker to create a book only if the document element is `reference` or `manual`, you can use this rule:

```
reader generate book for doctype "reference", "manual" {
   put element "section" in file "Sect";
   put element "chapter" in file;
}
```

With this rule, if you import this SGML document:

```
<!DOCTYPE reference
. . .
<reference>
<chapter>
<section>Intro
. . .
</section>
. . .
</chapter>
</reference>
```

FrameMaker creates two files: a book file for the `reference` element, and a document file for the `chapter` element. Since the `section` element occurs inside the `chapter` element, the

software doesn't create a separate document file for it. On the other hand, if you import this SGML document:

```
<!DOCTYPE chapter
. . .
<chapter>
<section>Intro
. . .
</section>
. . .
</chapter>
```

FrameMaker creates one file containing the entire structure; it does not create a separate book file.

For information on these rules, see Developer Reference, page 151: reader, Developer Reference, page 93: generate book, and Developer Reference, page 146: output book processing instructions.

Suppressing the creation of processing instructions for a book on export

By default, FrameMaker creates processing instructions for books and book components whenever you export a FrameMaker book to markup. This happens even if you use the `generate book` rule to cause FrameMaker not to need the processing instructions on import. To keep FrameMaker from writing any book or book component processing instructions, use the following rule:

```
writer do not output book processing instructions;
```

To confirm FrameMaker's default behavior of creating these processing instructions, use this rule:

```
writer output book processing instructions;
```

For information on these rules, see Developer Reference, page 166: writer and Developer Reference, page 146: output book processing instructions.

29 Additional XSL Transformation for XML

In addition to the default translation and the specific translations that you specify using read/write rules and the API, FrameMaker can apply transformations on both import and export of XML files that are specified by an XSL style sheet associated with an XML application.

In this chapter

This chapter describes the kinds of XSL transformations that are available on import and export of XML, and how you can control them. It contains these sections.

Overview of XSL translation for XML

If an XML document is associated with an XSL style sheet, FrameMaker can apply transformations defined in that style sheet when importing the XML document, or when exporting the document back to XML. These transformations are defined by the W3C XSLT standard; see http://www.w3.org/TR/xslt. FrameMaker supports the version 1.0 recommendations.

Upon import, XSL transformations are applied before the default read rules or any additional read rules you have defined. Upon export, XSL transformations are applied after the default or explicit write rules.

XSL transformations are performed on smartpaste, preprocessing, and postprocessing of the XML. That is, the result of applying an XSL transformation on import is a new file, which (if it is an XML file) is passed to the read/write rules. Similarly, the result of applying read/write rules on export is a new XML file, which, if it is valid, is passed to the XSLT processor.

> **Note: Entities** XPath does not provide any way of referencing entities in an XML file, so you cannot apply XSL transformations to entities.

Specifying an XSL file for import, export, and SmartPaste

You can specify an XSL file to be used for preprocessing an XML file on import in the XML file itself, or in the structure application defined in the `structapps.fm` file. If it is specified in both places, the value in the XML file takes precedence over the value in the structure application.

The XSL file to be used for postprocessing when exporting a file to XML, must be specified in the structure application.

The XSL file to be used for smartpaste when any content is copied and pasted into the structured file, must be specified in the structure application.

Specifying an XSL file in XML

To associate an XSL file with an XML file in the XML file itself, use the `xml-stylesheet` processing instruction (PI), and set the *ProcessStylesheePI* element in the `structapps.fm` file to `Enable`. (See Developer Reference, page 25: How the Stylesheets element affects XSL transformation.) When you import the XML into FrameMaker, the specified XSL file is used to preprocess the XML before read rules are applied.

If the `type` attribute of the `xml-stylesheet` instruction has the value `"text/css"`, the `href` attribute specifies a CSS style sheet. However, when the `type` attribute is either `"text/xml"` or `"text/xsl"`, the `href` attribute specifies the URI (path and filename) of an XSL file. For example:

```
<?xml-stylesheet type="text/xml" href="#style1"?>
<?xml-stylesheet type="text/xsl" href="style1.xsl"?>
```

The XSL can be embedded in the XML document, or it can be an external file.

Specifying an XSL file in a structure application

To specify an XSL file for either import or export in the structure application, include the element *XSLTPreferences* in the *Stylesheets* element which is a child of the *XMLApplication* element. (See Developer Reference, page 25: How the Stylesheets element affects XSL transformation.)

The *XSLTPreferences* element can have as its children as *SmartPaste*, *PreProcessing*, and/or a *PostProcessing* element. It can contain all subelements, but only one instance of each. Each *SmartPaste, PreProcessing*, and *PostProcessing* element can contain one *Stylesheet* element and one *Processor* element. The *Stylesheet* element references the XSL file that is to be used for *SmartPaste, PreProcessing, PostProcessing*. The *Processor* element references the processor (XALAN or SAXON) used for the transformation.

You can specify the path of the XSL file in the *Stylesheet* element in any of these ways:

- A URI.
- An absolute path.

- A path that uses the environment variable `$STRUCTDIR`.
- A relative path, which is resolved with respect to the path of the XML file being read or written.

You can either specify the preexisting XSLT processor (Xalan or Saxon) in the *Processor* element, or type the processor name that you want to use.

You can also specify a *StylesheetParameters* element as a child of *SmartPaste, PreProcessing,* or *PostProcessing*, in order to set parameters in the XSL at run time, before the transformation is performed. This element contains *ParameterName* and *ParameterExpression* pairs. Each pair specifies the name of a parameter used the XSL stylesheet, and an expression that constrains the value of that parameter for the subsequent transformation.

Applying XSL transformations

If XSL is associated with an XML structure application or an XML file, the transformations are applied independently of read/write rules.

SmartPaste When a user tries to copy-paste content from an external (unstructured) source into a structured file using smart paste, the content is modified according to the stylesheet before pasting.

Preprocessing on import When a user imports an XML document into FrameMaker, if an XSL file is specified, either in the XML file to be imported or in the *XMLApplication* element of the structure application, the tranformations defined in that XSL file are applied to the XML file before any other processing. The XML is not validated before the transformations are applied, even if the XML document specifies a DTD or Schema.

The result of the transformation can be an XML file, or some other kind of file, such as a MIF or text file.

- If the result of preprocessing is an XML file, FrameMaker imports that XML file into FrameMaker, applying all relevant default and specified read rules. The resulting opened document has the same title as the original XML document. If the user saves the document, it is saved by default to XML, and overwrites the original XML document.
- If the result of preprocessing is not an XML file, FrameMaker opens the resulting file independently of the structure application. The resulting opened document has the same title as the original XML document, but if the user saves it, FrameMaker asks for a filename to which to save it.

Postprocessing on export When a user exports a structured document to XML, FrameMaker applies all default and specified write rules and writes out an XML file. This file is then validated against the DTD or Schema specified in the structure application. If the validation fails, FrameMaker reports the error and does not apply any XSL transformations.

If the validation succeeds, FrameMaker looks for an XSL file specification.

- It looks first in the validated XML file. This can contain an XSL file specification if, for example, the document was imported from an XML document containing an *xml-stylesheet* PI, and *Retain Stylesheet Information* is enabled in the XML application.
- If there is no XSL file specification in the XML, FrameMaker looks for a *PostProcessing* element in the XML application.

When it finds a valid XSL file specification, FrameMaker applies the XSL transformations to the validated XML, and saves the result to the same XML output file.

XSLT errors

FrameMaker notifies the user if it encounters any of the following errors while attempting to apply XSL tranformations:

- FrameMaker cannot find or read the referenced XSL file.
- The XSLT processor fails, because, for example, the XSL file is not valid, is not formed in accordance with the W3C XSLT specification, or contains errors in XPath expressions.
- The result of XSLT preprocessing is a type of file that FrameMaker cannot open.

30 Developing Markup Publishing Applications

FrameMaker provides a robust and flexible set of tools for creating markup editing and publishing applications. All general-purpose markup systems require application-specific information. You can use FrameMaker to develop such an application in a set of steps, each of which is straightforward.

The major parts of an application correspond to components in which the information is specified, as follows:

Application Component	File
Element and attribute definitions	EDD (can be derived automatically from a DTD or an XML schema)
Format rules	EDD
Page layouts	Template
Markup representation of elements and attributes	Read/write rules and structure API client
Markup structure transformation	Structure API client or XSLT for XML

The structure application bundles this information so that it can be easily accessed. Application development involves creating these files along with data analysis, documentation, and maintenance activities.

This chapter provides an overview of the development process. It contains these sections.

- "Markup applications," next, examines the features and components of markup applications.
- "Developing Adobe FrameMaker structure applications" on page 430 summarizes the process of developing a structure application, covering the steps common to many editing and publishing tools, and those unique to FrameMaker.
- "Technical steps in FrameMaker application development" on page 435 enumerates the tasks involved in developing a FrameMaker application and defines the files containing modules of the application that the application developer maintains.

Markup applications

A markup application is an application that manages and stores critical business information in a markup format. Here, markup refers to either XML (Extensible Markup Language) or SGML (Standardized General Markup Language).

There are two important aspects of markup that make it suitable for streamlining the creation, maintenance, and delivery of critical business information:

- Open standards
- Formal structure

If you can take advantage of these aspects, you will see improvements in your publishing process, and increased accuracy in the information you deliver. Because of open standards, you can share your information with more people using a wider variety of software applications. If you take advantage of the formal structure in markup, you can develop increasingly sophisticated ways to automate your processes.

Part of the formality of markup is that it separates content from appearance. The way markup separates a document's content from its appearance and intended processing makes it a natural format for single-source documents. However, the ability to use text in different ways requires that each use be defined. In markup terminology, such a definition is called an *application*.

More precisely, the SGML standard (ISO 8879, Standard Generalized Markup Language) defines a text processing application to be "a related set of processes performed on documents of related types." In particular, the standard states that an SGML application consists of "rules that apply SGML to a text processing application including a formal specification of the markup constructs used in the application. It can also include a definition of processing." This definition applies equally XML

Whether you are interested in XML or SGML, a publishing system that uses this technology is a general *markup application*. A markup application is the sum total of the tools in your XML or SGML publishing system. You work with these tools to author, process, and deliver your information in a variety of formats that can include print, PDF, HTML, or business transactions.

Understanding markup applications

You can see the need for markup applications by comparing general-purpose markup tools to traditional database packages. Just as database software is used to maintain and access collections of numeric data and fixed-length text strings, a markup system is used to maintain and access document databases. Both types of systems typify powerful, flexible products that must be configured before their power can be used effectively. In particular, both tools require the user to define the data to be processed as well as how that data will be entered and accessed.

Database	Markup
Depends on a schema to define record layout	Uses a type definition (DTD or XML Schema) to define structural elements and attributes
Input forms facilitate data entry	Configured XML or SGML text editor facilitates data entry
Reports present customized summaries and other views of information	Composed documents present information to end user

The *DTD* and accompanying input and processing conventions comprise the rules that define a markup application.

While markup applications need not include a visual rendering of documents—consider a voice synthesizer, word count, or linguistic analysis tool—many of them do involve publishing. Publishing is the process of distributing information to the consumer. Traditional publishing involves preparation of written materials such as books, magazines, and brochures. Computerized information processing facilitates the publishing of information in additional media, such as the Internet, mobile applications or CD-ROM, and also provides the possibility of interfacing with other applications, such as database distribution.

Applying formatting to markup

Often, the same information is published in multiple forms. In such cases, its appearance may change according to the purpose and capabilities of each medium. The appropriate rendering enhances the reader's comprehension of the text. Since markup prepares information for multiple presentations without making assumptions about its rendering, additional information is needed to control each rendering. This formatting information, although pertinent to an application, is outside the scope of the markup itself.

Consider, for instance, a repair manual stored in markup. It may include procedures such as the following:

```
<taskmodule skill="adv">
<heading>Tuning a 5450A Widget</heading>
<warning type="1">Do not hit a widget with a hammer. Doing so
could cause explosive decompression.</warning>
<background>The 5450A Widget needs tuning on a <emph>monthly
basis</emph> to maintain optimum performance. The procedure
should take less than 15 minutes.</background>
<procedure>
<step time="1">Remove the tuning knob cover.</step>
<step time="5">Use the calibration tool to adjust the setting to
initial specifications.</step>
</procedure>
</taskmodule>
```

The markup can be passed through a rendering application to present the mechanics with a formatted version:

Tuning a 5450A Widget

This procedure requires advanced certification.

Warning: Do not hit a widget with a hammer. Doing so could cause explosive decompression.

The 5450A Widget needs tuning on a *monthly basis* to maintain optimum performance. The procedure should take less than 15 minutes.

1. Remove the tuning knob cover.
2. Use the calibration tool to adjust the setting to initial specifications.

The application that prepares the formatted version applies rules indicating how the elements of the markup example are to appear. This example applies these rules:

- Put the `heading` element in a large, bold font.
- Generate the sentence, "This procedure requires advanced certification" from the value of the `skill` attribute.
- Insert the italic header "Warning:" before the text of the `warning` element.
- Number the steps in the `procedure`.

Notice that the values of the `time` attributes do not appear in the formatted text. The rules that drive a different application—a scheduling program, for instance—might use the `time` attribute to allocate the mechanic's time, for example. Or it might assemble the entire maintenance procedure (possibly thousands of such maintenance cards), and add up the times for each step to estimate the maintenance schedule.

Markup editing and publishing applications

Two important classes of markup applications involve creating (editing) and publishing documents. Editing tools are used to create markup documents; publishing tools produce formatted results from existing markup documents.

It is easiest to work with an editor if you have some visual indication of the document's structure, so markup text editors such as Adobe FrameMaker also require some formatting specifications. The editor can provides rules to determine, for example:

- How closely documents under development must conform to their type specification
- Options for listing available elements for the user
- Whether to automatically insert elements when the user creates a new document or inserts an element with required sub-elements
- Whether to prompt for attribute values as soon as the user creates a new element

Some tools address either the editing or the publishing application. FrameMaker is a single tool that addresses both. Although you can use FrameMaker for either activity alone, its strengths include the ability to create new structured documents (or edit existing ones) that are ready for publishing without additional processing. This pairing allows a single set of format rules to support both tasks. Thus, less effort is required to configure FrameMaker than to configure separate editing and publishing tools.

Developing Adobe FrameMaker structure applications

Developing a FrameMaker application shares many steps with developing a markup application for any other editing or publishing tool. This section summarizes the process, covering the steps common to many editing and publishing tools, and those unique to FrameMaker.

FrameMaker editing philosophy

Formatting, or visual clues—font variation, indentation, and numbering—help readers understand and use written information. FrameMaker is a *WYSIWYG* editor. While most *WYSIWYG* editors require authors to indicate how each part of the document is formatted, FrameMaker can use the rules in the markup application to format document components automatically in a consistent manner.

This combination of *WYSIWYG* techniques with the structured principles of markup is unique to FrameMaker. The goal in FrameMaker, is to streamline the process of creating and publishing documents through the use of markup. While authors manipulate elements and their attributes to create native markup documents, they work in a *WYSIWYG* fashion instead of directly with markup syntax.

Since the markup document model underlies edited material, the markup form of the document can be produced at any time during the *WYSIWYG* editing process. This process relies on the underlying element structure and hence has no need for the inaccurate filtering schemes that plague tools for generating markup from word processing formats.

Tasks in FrameMaker application development

The major tasks involved in the development of a FrameMaker application are:

- Defining the elements that can be used in a document and the contexts in which each is permitted. This task is analogous to developing a DTD. In fact, the element definitions can be automatically derived from a DTD if one exists.
- Determining which elements correspond to special objects, such as tables, graphics, and cross-references.
- Developing format rules that define the appearance of a structural element in a particular context.

- Providing page-layout information such as running footers and headers, margins, and so on—the foundation of every *WYSIWYG* document.
- Establishing a correspondence between the markup and FrameMaker representations of a document. For most elements, this correspondence is straightforward and automatic, but FrameMaker allows for some variations. For example, terse markup names can be mapped into longer, more descriptive FrameMaker names, and variant representations of tables, graphics, and other entities can be chosen.

Each of these tasks requires planning and design before implementation, as well as testing afterward.

The analysis phase

The specific features of a FrameMaker application largely depend on the tasks it will perform:

- Will users create new content, or will the application simply format existing markup documents?
- Whether or not they require further editing, are existing markup documents to be brought into the system? What about documents that are not XML or SGML (unstructured FrameMaker documents or, perhaps, the output of a word processor)? Will such legacy documents be imported on an ongoing basis or only at the beginning of the project?
- Will finished documents be saved as markup? Will authors themselves output markup, or will they pass completed projects to a production group that performs this post-processing step?
- Does the application involve a database or document management tool? Will portions of the documents be generated automatically? Are there calculations to be performed?
- Will there be a single DTD or a family of related DTDs used for different tasks (for example, a reference DTD used for interchange with a variant authoring DTD)?

Unless a project is based on one or more existing DTDs, one of the first outputs of the analysis phase is definition of the document structures to be used. The definition process usually involves inspecting numerous examples of typical documents. Even when there is a DTD to start from, the analysis phase is necessary to produce at least tentative answers to questions such as the previously listed ones.

Defining structure

A primary goal of the analysis phase is defining the structures that the end user will manipulate within FrameMaker. The bulk of this effort is often the creation of a DTD. When the project begins with an existing DTD, the DTD provides a foundation for the structure definition. FrameMaker, in fact, can use the DTD directly. Nevertheless, a pre-existing DTD does not eliminate the need for analysis and definition. As Eve Maler and Jeanne El Andaloussi explain in *Developing SGML DTDs: From Text to Model to Markup*, many projects use several related DTDs. If the existing DTD is intended for interchange, the editing environment can often be made more productive by creating an editing DTD. Several examples from the widely known DocBook DTD used for

computer software and hardware documentation illustrate the types of changes that might be made:

- Changing some element names, attribute names, or attribute values to reflect established terminology within the organization.

  For example, DocBook uses the generic identifier `CiteTitle` for cited titles. If authors are accustomed to tagging such titles as `Book`, an application might continue to use the element name `Book` during editing, but automatically convert `Book` to and from `CiteTitle` when reading or writing the markup form of the document.

- Omitting unnecessary elements and attributes.

  DocBook defines about 300 elements. Many organizations use only a small fraction of these. There is no need to make the remainder available in an interactive application. FrameMaker displays a catalog of the elements that are valid at a given point in a document. Removing unnecessary elements from the authoring DTD means the catalog displays only the valid elements that an organization might actually use in a particular context and avoids overwhelming the user with a much larger set of valid DocBook elements.

- Providing alternative structures.

  For example, DocBook provides separate element types—`Sect1`, `Sect2`, and so on—for different levels of sections and subsections. Defining a single `Section` element to be used at all levels (with software determining the context and applying an appropriate heading style) simplifies the task of rearranging material and hence provides a better environment for authors who may need to reorganize a document. Again, `Sections` can be automatically translated to and from the appropriate `Sect1`, `Sect2`, or other DocBook section element.

- Simplifying complex structures that are not needed by an organization.

  DocBook, for instance, provides a very rich structure for reporting error messages. If only a few of the many possibilities will actually be used, you can edit the DTD to eliminate unnecessary ones.

In addition to providing and editing the DTD, the analysis and definition of structure includes planning for the use of tables and graphics. Many DTDs provide elements that are clearly intended for such inherently visual objects. In other cases, however, such a representation of an element results from the formatting rules of the application. Consider again, for instance, the repair procedure on page 429. The formatted version did not display timing information for the procedure steps. This alternative version formats the procedure in a three-column table, showing the duration of each step:

Tuning a 5450A Widget

This procedure requires advanced certification.

Do not hit a widget with a hammer. Doing so could cause explosive decompression.

Step	Duration	Description
1	1 minute	Remove the tuning knob cover.
2	5 minutes	Use the calibration tool to adjust the setting to initial specifications.

Thus, defining structure involves defining the family of DTDs to be used in the project as well as deciding in general terms how the various elements will be used.

The design phase

The analysis phase of the project evolves into a design phase with two major goals: defining the desired results as well as the best way to accomplish those results. Defining the desired results includes the graphic design of the completed documents: page layouts as well as visual characteristics to be assigned to each structural element. Even when there is a rich body of sample documents and a tradition of consistent use throughout an organization, reexamination and systematic inspection may reveal unexpected inconsistencies and suggest new approaches. As a result, analysis frequently results in changes to existing processes.

Defining the best way to accomplish the desired results involves planning how to use FrameMaker—and any other relevant software—to meet the project goals. While this step includes planning how each structural element will be formatted in various contexts, it also must account for user interface aspects. Unfortunately, this process is frequently slighted in markup projects. As a result, SGML and to a lesser extent XML has acquired an undeserved reputation in some circles for being complex and difficult to use. In fact, poorly designed markup applications are often at fault.

Many things can be done to make markup easier and more appealing to the user. For example, FrameMaker menus can be customized, both by adding new application-specific commands defined through the FDK and by simplifying the menus by removing access to unneeded capabilities. In this part of the design phase, the project team must consider questions such as the following:

- What is the expected sequence of tasks that users will perform?
- Are there repeated steps that can be automated through the FDK?
- Will users work with complete documents or with fragments?
- Will all documents use the same formatting templates?

Implementing the application

Once the technical goals of the project have been determined, implementation begins. "Technical steps in FrameMaker application development" on page 435 describes the steps involved in implementing a FrameMaker application.

Testing

Although writing a DTD or a formatting specification does not use a traditional programming language, it is essentially a programming process. Therefore, even when there is no need for FDK clients, creating an markup application (for FrameMaker or any other tool) is a software development effort and, as such, requires testing. Two types of tests are needed: realistic tests of actual documents and tests of artificial documents constructed specifically to check as much of the application as possible.

All necessary processes must be tested: formatting, markup import, and markup export. Test documents must include FrameMaker documents to save as markup, and markup documents to open in FrameMaker. Testing must also include any processing of legacy documents, including the use of the FrameMaker conversion utility. Finally, once the application is used in production, continued testing should incorporate some actual production documents.

Training and support

Despite the intuitive nature of structured documents, end users cannot be expected to understand the intended use of different element types and attributes simply by working with them. They must be provided with training in the form of documentation, classes, or sample documents. Application-specific training can be combined with training on FrameMaker and other tools to be used in the project.

Provision must also be made for ongoing support. As they use the application, end users will occasionally have questions or encounter bugs. They may need to use structures that have not already been defined.

Maintenance

While the application development effort decreases over time, some maintenance should always be expected. In addition to needing bugs fixed, applications may be changed to accommodate new formatting, to encompass additional documents, and to track updates to the DTD.

The implementation team

So far, the different phases in the development of an markup application have been enumerated. Implementing those phases requires a team of people with different areas of expertise.

All large projects begin with an analysis phase. Participants must include both end users and developers, who will work on the eventual implementation. For an XML or SGML project, the end users include.

- Authors, editors, and graphic designers (production formatters) who will use the editing and publishing application
- Subject-matter experts familiar with the content requirements of the documents to be processed.

Developers include individuals with expertise in

- Markup
- FrameMaker
- XSLT
- Any other markup tools to be used
- Additional software tools, such as document management systems and database packages

Additional participants in the analysis may contribute significantly to the project definition even if they will neither develop nor use the completed application. For example, the organization's Web advocate may be able to identify some requirements that do not affect the rest of the team.

The skill sets required within the implementation team include document design, markup knowledge, setting up FrameMaker formatting templates, and setting up formatting rules that control automatic application of the desired graphic design to structured documents. If the FDK is used, programming skills are also needed. Of course, technical writing skills in documenting the finished application are a valuable contribution to the completed effort.

Technical steps in FrameMaker application development

This section enumerates the tasks involved in developing a FrameMaker application and defines the files containing modules of the application that the application developer maintains.

Element definition documents (EDD)

At the heart of every FrameMaker application is an element definition document (*EDD*). An EDD provides three types of information:

- The definitions of the elements that can occur in a document, including their allowable content and attributes. These definitions can be automatically extracted from a DTD or Schema.
- The formatting rules that define the visual characteristics of various document components.
- Other information governing behavior of elements, including rules for automatically inserting several elements in response to a single user action, preparing for the use of document fragments, and so forth.

An EDD is itself a structured document, and you use the structured editing features in creating and editing the EDD. You do not need not remember where to insert element definitions, context specifications, or format rules, because the Structure View and Element Catalog guide you in creating these constructs correctly.

The EDD formats the structure elements for readability: Different fields are clearly labeled, and spacing, indentation, and different fonts emphasize the organization. You can enhance the accessibility of information by grouping element definitions into sections and explaining them with extensive comments. These characteristics are illustrated by the following fragment:

Element Definition Document (EDD) for Reports
This EDD defines the structure rules for a report.

Element (Container): Report
Valid as the highest-level element.
General rule: (Title, Abstract, Contents, Chapter+, Appendix*)

Report and Chapter Structure
A report consists of Chapters and Appendices, preceded by a Title, Abstract, and Table of Contents. The Chapters and Appendices may be divided into Sections. Each Chapter, Appendix, and Section has a Head (or title).

Title of the entire report (the Head element is used for Chapter, Appendix, and Section titles).

Element (Container): Title
Valid as the highest-level element.
General rule: (<TEXT>)
Text format rules
1. **In all contexts.**
Basic properties
Alignment: Center
Line spacing
Height: 12pt
Line spacing is fixed.
Default font properties
Weight: Bold
Size: 36pt

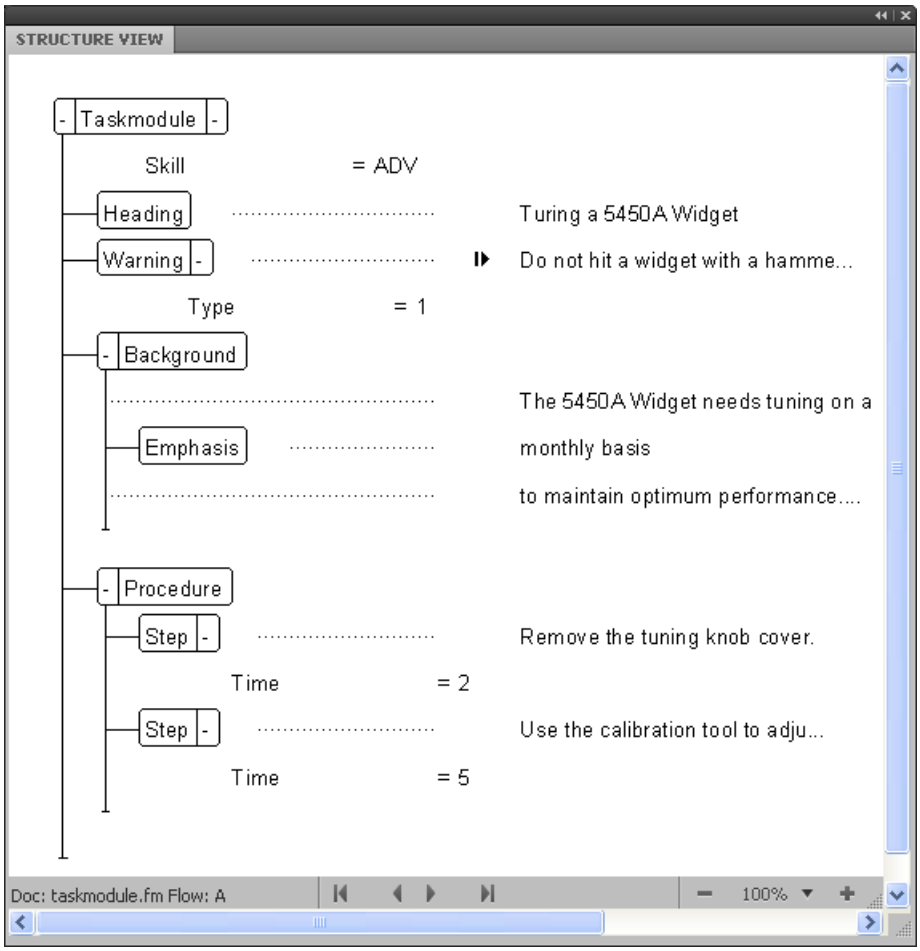
Structured templates

End users do not use EDDs directly. Instead, the definitions made in an EDD are extracted and stored in a template—that is, a FrameMaker document used as a starting point for creating other documents. Developers provide end users with a template that incorporates the structure and format rules from a particular EDD, as well as page layout information and formatting catalogs that define paragraph, character, cross-reference, and table styles. It may also contain sample text and usage instructions.

In some applications, information can be formatted in several ways. For example, the same text may need to be formatted for books with two page sizes. FrameMaker can accommodate such requirements. If an EDD's format rules refer to paragraph and character styles in the template's catalogs, new catalogs can be imported from another template to change the document's appearance. New page layouts, table styles, and cross-reference styles can also be imported. If the format rules incorporate specific formatting parameters, the appearance can be changed by importing a new EDD with different format rules.

Moving data between markup and FrameMaker

FrameMaker maintains the native element and attribute structure of markup in a *WYSIWYG* environment. The FrameMaker user can easily inspect this element structure in the Structure View. For the maintenance procedure example, the Structure View appears as follows:



Since both the FrameMaker rendition and the markup text file include content and element structure, data can move between the two forms automatically. The user interface is

straightforward. The FrameMaker **File > Open** command recognizes XML and SGML documents. When a user opens one, FrameMaker automatically converts the document instance to a structured *WYSIWYG* document and applies the format rules of the appropriate markup application. To write a structured *WYSIWYG* document to XML the user simply selects the **File > Save**, or **File > Save As XML...** command. For SGML, the user specifies SGML when executing the **File > Save As...** command.

Just as the form of a structured document parallels the form of a markup document instance, an EDD parallels a DTD or Schema. FrameMaker can also move definitions of possible structures between the two forms. Thus, if a new project is based on an existing XML DTD, FrameMaker can automatically create an EDD from the DTD. For example, given DTD declarations such as

```
<!ELEMENT step (#PCDATA)>
<!ATTLIST step time NUMBER #IMPLIED>
```

FrameMaker builds the following element definition:

Element (Container): Step
General rule: <TEXT>
Attribute list
1. **Name:** Time **Integer** **Optional**

Since there is no formatting information in the DTD, no format rules are included in the automatically generated EDD. You can manually edit this information into the EDD, or import a CSS style sheet into the EDD. You only need to do this once, even if the DTD is revised.

It is common for a DTD to undergo several revisions during its life cycle. When you receive an updated DTD, FrameMaker can automatically update an existing EDD to reflect the revision, and the update process preserves existing formatting information and comments in the EDD. The update incorporates changes to content models and attribute definitions, inserts new element types and removes discarded ones, then generates a short report summarizing the changes so that the application developer can review them.

If a project is not founded on an established DTD or Schema, the application developer can start implementation by creating an EDD. Once the EDD is finished, FrameMaker can automatically create the corresponding DTD.

Using XML Schema to create a DTD or EDD

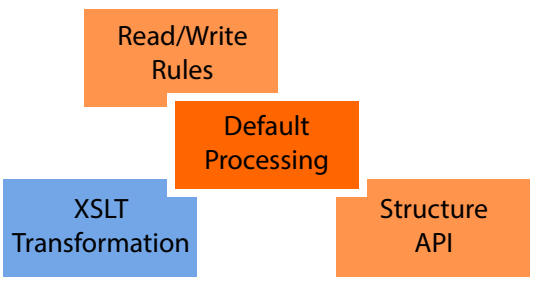
FrameMaker can create a DTD from the Schema, and the EDD from that DTD. This happens automatically when you import an XML file that uses Schema and does not refer to a DTD. You can also import a Schema directly, use it to create a DTD and EDD, and create a template or reference the resulting DTD in your XML documents. For example workflows, see Developer Reference, page 201: XML Schema to DTD Mapping.

Customizing markup import/export

You might want to customize the correspondence between a document's markup and FrameMaker representations for several reasons; for example:

- To base the FrameMaker application on an authoring version of an interchange DTD
- To integrate FrameMaker with database or document management tools
- To calculate portions of a document or display predefined text when certain elements or attribute values occur or particular entities are used
- To interpret certain elements, such as graphics or tables, as special objects

To support such requirements, FrameMaker offers three strategies for specifying how abstract markup structures are represented in the *WYSIWYG* publishing environment.

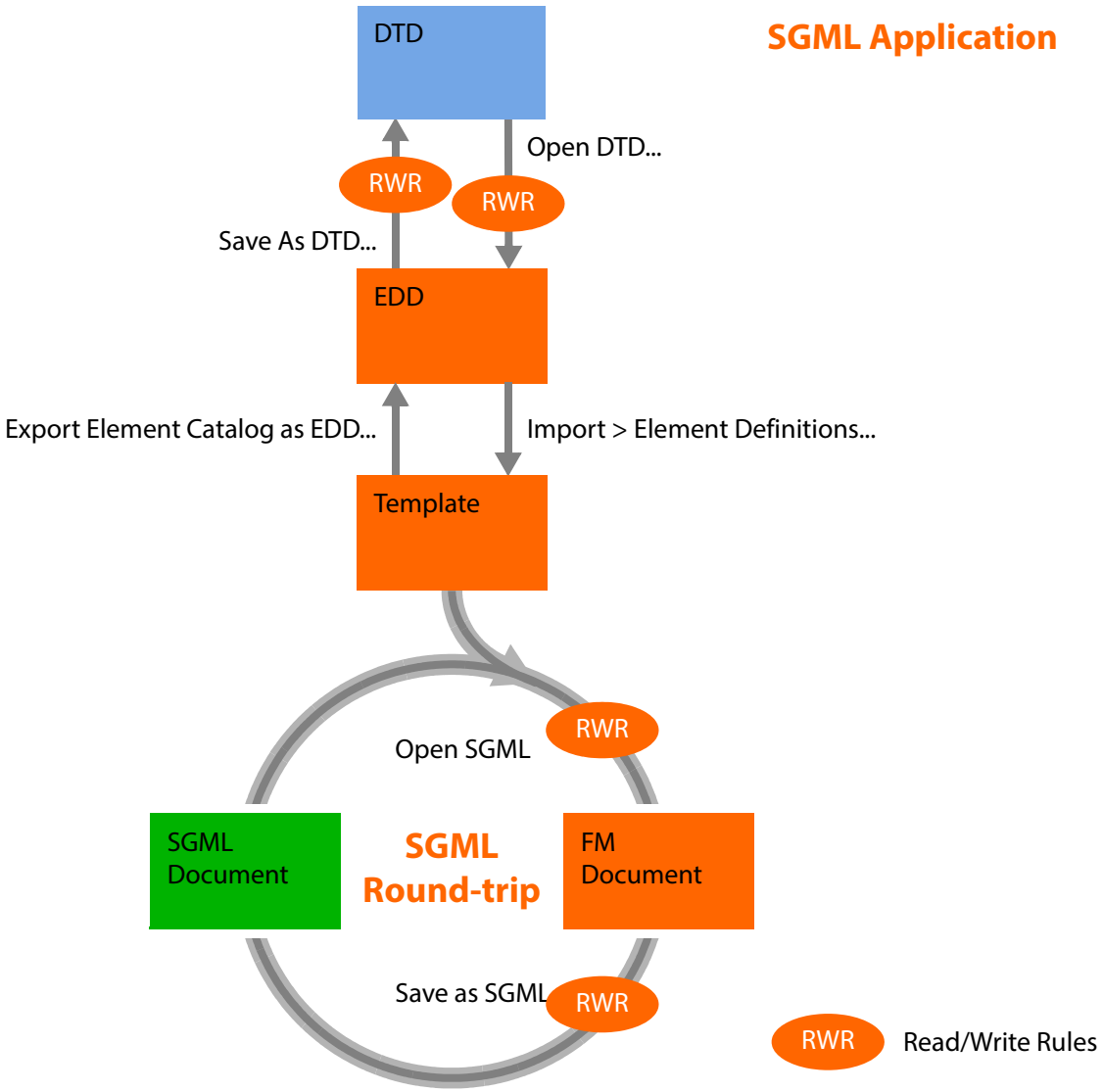


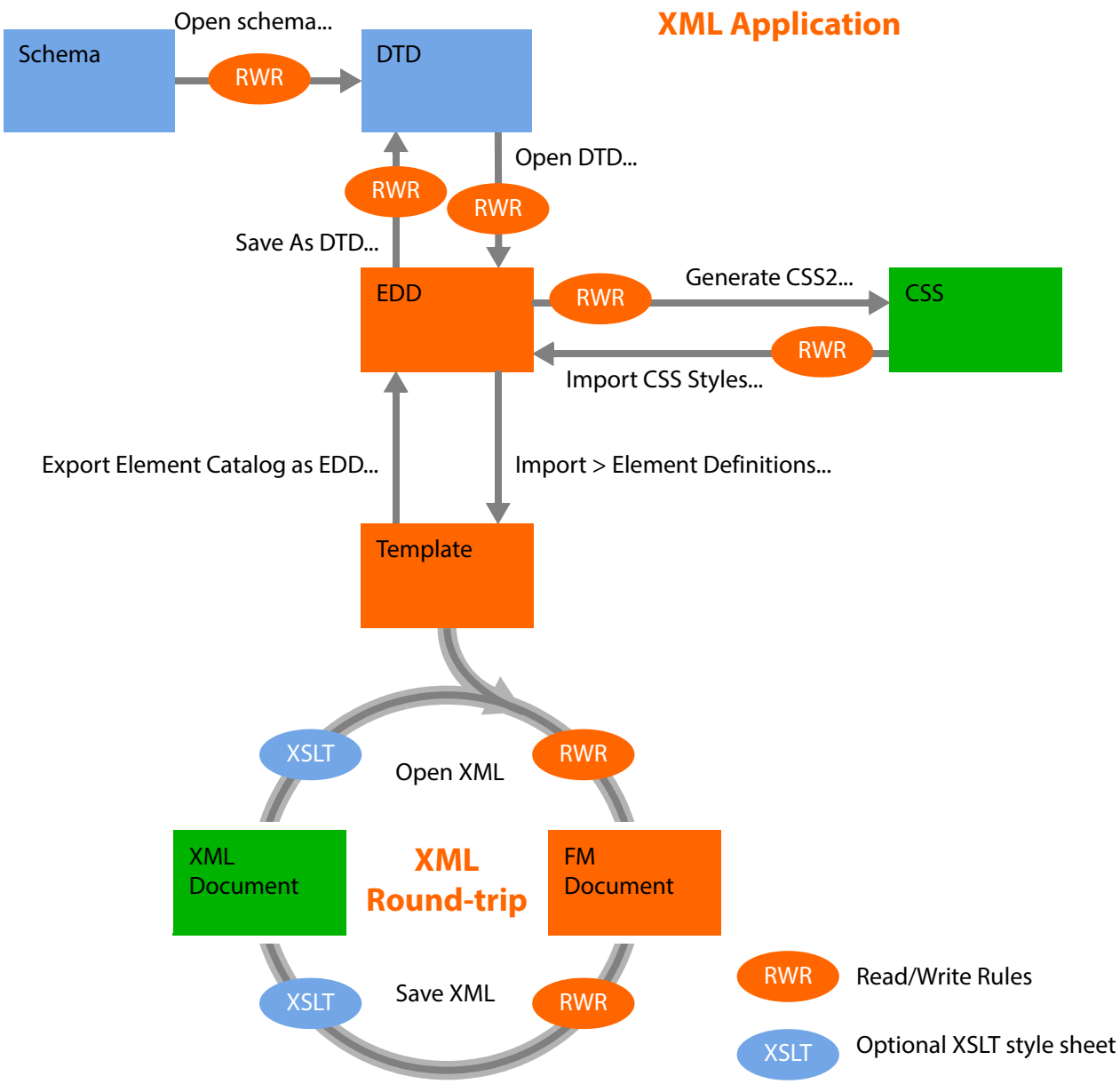
Most projects combine two or more of these approaches:

- By default, FrameMaker automatically translates a markup element into a FrameMaker element with the same name (and translates a FrameMaker element to a markup element with the same name); similar processing occurs for attributes. For many elements, the default processing is adequate.
- Some customizing can be accomplished through *read/write rules*. FrameMaker provides read/write rules for customizing that is common to many applications. For example, there are read/write rules for mapping markup elements into specific FrameMaker element objects such as table parts or graphics. The developer creates read/write rules in a separate file.
- For cases in which read/write rules are insufficient, the FDK includes a C function library called the Structure API, which enables more extensive customizing. The Structure API can be used, for instance, to extract part of a document's text from a database using attribute values as database keys.

- XSLT may be used to transform an XML structure into a structure that is more appropriate for editing or formatted output with FrameMaker. Export transformation converts the XML back into its interchange structure.

Read/write rules can be used with all four possible conversions—markup document to FrameMaker document, FrameMaker document to markup document, DTD to EDD, and EDD to DTD. These conversions are shaded in the following diagrams, which also show the FrameMaker commands that invoke them and their interaction with other data files.





Structure application files

Moving documents and type definitions between their markup and FrameMaker forms involves a number of files. The developer bundles the names of the files needed for a particular application—as well as other information—into a *structured application definition document*. Definitions for all markup applications available on a particular computer system are grouped in an application file. The FrameMaker configuration file, `structapps.fm`, is a structured

application definition document that FrameMaker reads whenever it starts. The following example is a fragment of a typical structure application file:

Application Definition Version <*version*>

Application name: Maintenance
- **DTD:** task.dtd
- **Read/write rules:** task.rw
- **SGML declaration:** namelen.big
- **DOCTYPE:** taskmodule

Application name: Report
- **Template:** report.tpl
- **Entity locations**
  - **Entity catalog file:** catalog

Because FrameMaker automatically reads the `structapps.fm` file, writers and editors do not need to know about the structure application file. To import or export markup documents, however, they may need to know the names of available applications—but even this information can be stored in a FrameMaker template or automatically selected from the document type name of an markup document. For example, given the preceding structure application file example, when reading an SGML document that begins:

```
<!DOCTYPE taskmodule SYSTEM "task.dtd">
```

FrameMaker uses the Maintenance application. Because that application specifies read/write rules, the indicated read/write rules control the import. Furthermore, the Maintenance application is stored in the resulting structured document so that the same read/write rules are used if the document is later exported back to markup.

The information in an application definition can include:

- One or more document type names that trigger use of the application on import.
- The name of the file containing the DTD to be used for export. This field is not needed for applications that only require import, since imported markup documents always include a document type declaration. When it does appear, the "DTD for export" field actually contains the name of a file containing DTD declarations (sometimes called an external DTD subset) suitable for use in document type declarations such as the one shown in the preceding example for the Maintenance application.
- Character encoding.
- The handling of conditional text for export to XML.
- Files containing external entities (identified by entity name or public identifier).
- An entity catalog.

- Search paths for subfiles of read/write rules and external entities.
- External cross reference handling for XML.
- Whether namespaces are enabled in an XML application
- The name of the file containing the application's read/write rules.
- An XML schema reference
- Identification of a structure API client to use during import or export.
- The handling of CSS style sheets for import and export.
- The names of XSLT style sheets for import pre-process and export post-process for XML transformation.
- The name of a file containing an SGML declaration to be used for export when no SGML declaration appears at the beginning of imported SGML documents.
- The name of a template to be used to create new documents during import. The template incorporates the element definitions imported from an EDD as well as page layout information and formatting catalogs. A template is needed only for applications that include markup import.
- XML Encoding for export or display

Legacy documents

In general, the conversion of unstructured documents to structured documents (or to markup documents) cannot be completely automated. FrameMaker offers a utility that adds structure to unstructured FrameMaker documents. This is done by mapping paragraph and character styles as well as other objects—such as footnotes, cross-references, markers, and tables—to elements. Some manual polishing of the resulting documents is expected; the amount of editing depends on the discipline with which formatting catalogs were used in the original and on whether a unique set of elements corresponds to each formatting tag. Since FrameMaker can read many common word processing formats, this utility can be used to structure word processor documents as well. Once any errors in the resulting structured document have been repaired, it can be exported to markup. Thus, FrameMaker provides a path for converting word processor documents to markup.

The rules for mapping between styles and elements are specified in a conversion table. Techniques for defining the mapping are provided in Developer Reference, Chapter 4, Conversion Tables for Adding Structure to Documents

Application delivery

Once an application has been completed, it must be delivered to the end users. The developer can either install the completed application on the end users' system or network, or provide instructions for doing so. At a minimum, end users need copies of all templates and FDK clients.

If end users will be structuring existing unstructured documents, they will also need copies of all conversion tables used. Finally, if end users will be exporting or importing markup documents, their structure application files must be modified for the new application. They need access to the read/write rules, entity catalog, DTD/Schema for export, XSLT style sheets, and (for SGML structure applications only) an SGML declaration.

Typical application development scenarios

Although creation of each component of a FrameMaker application is straightforward, novice developers may need some guidance as to the order in which to create these components. The strategy for creating an application can vary considerably with the requirements of a particular project. The following scenarios begin after the analysis and design phases have been completed. In all three, the developer implements a few elements at a time and stops to test the growing application frequently.

Starting from existing markup documents

In the first scenario, there is an existing DTD and sample markup documents. You select a short sample document and start development by supporting the element types that occur in it. Later, you add more complicated documents. Application development proceeds as follows:

1. Create a new file of read/write rules and enters rules that reflect any decisions made during the analysis phase of the project. In particular, enter rules to
   - –Identify elements used for graphics, tables and table components, and cross-references
   - –Discard elements and attributes that the organization will not use even if they are defined in the DTD
   - –Rename elements and attributes to reflect the organization's terminology
   - –Replace terse, abbreviated markup names with longer, more readable FrameMaker versions
   - –Specify FrameMaker names with multiple capital letters. For instance, you might map the element generic identifier `BLOCKQUOTE` to the FrameMaker element tag `BlockQuote`. Such rules will not be needed with SGML applications if the SGML declaration does not force general names to uppercase (that is, if case is significant in the SGML names).
2. Define a new markup application in the structure application file that invokes the read/write rules. If appropriate, the application definition specifies an XML or SGML application and maps any public identifiers used in the DTD to system filenames.
3. Using the new application, create an EDD from the DTD with the **StructureTools > Open DTD** command. The resulting element declarations may be grouped into sections, and some added comments in the process.
4. Open the selected sample document. The resulting FrameMaker document serves as the basis of a template. In this document, define page layouts, specifying the number of columns on the page, the width of any side-heads, page numbers, and so forth.

5.Return to the EDD and begin to define formatting rules. In general, formatting specifications are unnecessary for higher level elements, such as those for chapters and sections. Concentrate on elements that contain text and their immediate ancestors, such as elements for list items, notes, cautions, emphasized phrases, and cited titles.

6.After providing format rules for a few elements, test them by using the **File > Import > Element Definitions** command to import the element definitions from the EDD into the sample document. This command reports any formatting catalog entries mentioned in format rules but not defined in the sample FrameMaker document. Define all reported formats and glance through the sample document to verify that the formatted elements appear as intended.

   You can then return to the EDD, correcting any errors in the format rules and adding some more before testing them again, until the sample document is completely formatted. At that point you might want to test some other documents. After creating a base for further development, continue systematically through the DTD, making and testing context-sensitive format rules for all necessary elements.

7.You might need to add more read/write rules. If you change additional element tags, the original names must be replaced by the new versions throughout the EDD, in the definitions of the changed elements as well as in definitions that refer to those elements. To update the names in the EDD, use the **StructureTools > Import DTD** command, which reprocesses the DTD according to the current read/write rules and updates the existing EDD. It writes a report listing the changes made.

8.To create a template from the sample document, you can delete all content and save the result.

9.Use the template to test the editing environment by adding automatic insertion rules to the EDD, specifying, for example, that when the end user inserts a `List` element, FrameMaker should automatically insert an `Item` element within it. Consider whether an FDK client can provide input accelerators, such as automating frequent sequences of editing steps. You might also want to customize the FrameMaker menus for end users, perhaps removing developer-oriented commands or (to prevent the end user from overriding automatic format rules) removing formatting commands.

10.For markup import, add the template to the application definition in the structure application file, and add read/write rules for special characters and other entities. Once again, open the markup version of the sample document, test the result, and modify the application until the results are correct.

11.If read/write rules are insufficient to import any elements, use the structure API to develop a client with the requisite functionality.

12.If markup export is also required, test it and add other read/write rules or structure API functions, if necessary.

13.Once formatting, import, export, and editing functions have been tested, deliver the completed application, along with documentation, to the end user.

Working with Schema

You can import an XML document that references a Schema file, and you can specify a Schema file in your structure application, to use for validating a document upon export to XML.

1.For a specific XML document, you can include the path of the schema file in the XML using attributes - `noNamespaceSchemaLocation` or `schemaLocation` depending on whether your schema includes a target namespace or not.

2.To specify a Schema file for use in exporting XML, modify the `structapps.fm` file. Use the Schema element as part of the `XMLApplication` to provide the Schema file path for export.

3.Open the XML in Frame using a structured application. Edit it.

4.Save the XML using a structured application. The Schema element in the structapps.fm file is output in the file and validation is performed against it.

In this workflow, a DTD is generated automatically as an intermediary file from the Schema given in the XML document, and you do not modify it. However, you can also use a Schema file to generate an EDD directly, or you can modify the DTD and reference it from the XML document. When an XML document references both a Schema and a DTD, FrameMaker imports it using the DTD, although it still validates against the Schema.

Creating a DTD or EDD from Schema

You can create a new EDD from a Schema definition, or import the elements from a Schema definition into an existing EDD. FrameMaker converts the Schema definition to DTD first, and then creates or imports elements to an EDD. To do this, use these commands in the **StructureTools** menu:

- Open Schema: This command converts a specified Schema to DTD, and creates a new EDD from the DTD.
- Import Schema: This command converts a specified Schema to DTD, and imports elements from the DTD into an existing EDD.

Each command allows you to specify the Schema file, then asks you where to save the resulting DTD file. To create an EDD from Schema:

1.In Structured FrameMaker, click **Structure Tools > Open Schema**. Choose a Schema file.

2.Choose a path for the DTD to be output.

3.Examine the resulting DTD and make any modification you wish.

4.Create an EDD from the generated DTD.

5.Use this EDD to create a template that can be included in the Structured Application

6.Provide your DTD path along with the Schema Location in the input XML. This will make sure that FrameMaker works correctly with your template. Validation of input and output XML is still performed against the Schema.

Building a new application

Another typical use of FrameMaker involves developing the document type definition from scratch. In this scenario, FrameMaker is used to write documents that are then exported to markup. Again, development work relies on a thorough data analysis.

1. Instead of starting with a DTD, begin to develop an EDD. Guided Editing makes the mechanics of editing the EDD simple. You can choose to define both content rules and attribute definitions for one element at a time, or to complete all content rules before defining any attributes. You can also define content rules for numerous elements before adding any format rules.
2. Create a template and then use the **File > Import > Element Definitions...** command to interpret the EDD and store the resulting element catalog in the template.
3. Use the template to test sample documents. As in the scenario of the previous section, you may find it easier to keep track of the work by repeating the edit-import-test cycle a few elements at a time.
4. When the EDD is complete, use the **StructureTools > Save As DTD** command to create an equivalent DTD.
5. Create a structure application that uses the generated DTD for export.
6. Test the export of the sample document and add read/write rules and a structure API client as necessary. If you do not want to use the FrameMaker default representation for special object elements, edit the corresponding element and attribute definition list declarations, in addition to providing the associated rules.
7. You can edit the generated DTD by allowing for tag omission and providing appropriate short reference mapping, short reference use, and entity declarations. Since FrameMaker exports all comments in the EDD to the generated DTD, review comments carefully. Retain comments pertaining to the structures being defined, but discard comments specific to FrameMaker formatting, since they do not necessarily apply to all uses of the DTD.
8. If the application requires markup import as well as export, test conversion in the import direction as well, modifying the application to correct any errors.
9. Finally, deliver the application to end users and prepare for the project's maintenance phase.

Working with legacy documents

The final scenario assumes there are existing unstructured FrameMaker documents. The developer can take either of two general approaches: constructing a conversion table to structure the legacy documents and using the resulting structure to start an EDD; or creating an EDD based on an analysis of the legacy documents and then creating a conversion table to reflect the element definitions in the EDD. The following steps outline the first approach:

1. Select a typical unstructured document and use the **StructureTools > Generate Conversion Table** command to create a new conversion table based on the format tags of the

unstructured document. This command puts a row in the table for each type of FrameMaker object and format tag used in the sample document. The initial conversion table might appear as follows:

Wrap this object or objects	In this element
P:MainTitle	MainTitle
ChapterTitle	P:ChapterTitle
P:Body	Body
C:Emphasis	Emphasis
G:	GRAPHIC
F:flow	FOOTNOTE
T:Table	Table

2. Edit the conversion table, changing entries in the second column, adding entries in the third column, and adding new rows for higher level structures.

3. Use the conversion table to add structure to the sample document, ignoring the lack of formatting in the resulting structured document but correcting as many structural errors as possible by editing the conversion table and reapplying it.

4. With the **StructureTools > Export Element Catalog as EDD** command, extract an EDD that has an element definition for each element type that appears in the sample document. Although these element definitions define the content of the elements very generically—each is allowed to contain text as well as any defined element—the EDD provides a skeleton that you can further develop.

5. The exported EDD lists the element definitions alphabetically, according to their tags. You can rearrange them into a logical order if desired—easily moving them around using drag-and-drop in the Structure View—and then replace the content rules with more restrictive ones and add format rules.

6. The sample document (and other automatically structured documents) can be reformatted by using the **File > Import > Element Definitions** command from the EDD into the generated structured document

7. Continue to finish the application, as in the previous scenario. However, since page layouts and formatting catalogs were defined in the original unstructured document, the template development effort does not need to be repeated.

Index

A

B

C

D

E

F

G

H

I

K

L

M

T

U

V

W

X

Legal notices

For legal notices, visit the Legal Notices page.