Adobe® FrameMaker® 6.0

FrameMaker+SGML Developer's Guide
Online Manual

Adobe, the Adobe logo, Acrobat, Acrobat Reader, Adobe Type Manager, ATM, Display PostScript, Distiller, Exchange, Frame, FrameMaker, FrameViewer, InstantView, and PostScript are trademarks of Adobe Systems Incorporated. Apple, PowerBook, QuickTime, Mac, Macintosh and Power Macintosh are trademarks of Apple Computer, Inc., registered in the United States and other countries. HP-UX is a registered trademark of Hewlett-Packard Company. Microsoft, MS-DOS, Windows, and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Sun and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. Unix is a registered trademark and X Window System is a trademark of The Open Group. All other trademarks are property of their respective owners. © 2000 Adobe Systems Incorporated. All rights reserved.

Contents

Before You Begin xiii

Part I Developing a FrameMaker+SGML application 1

Chapter 1 *SGML Application Basics 3*

Where to begin 3

SGML application scenarios 3

- Translating in one or two directions? 3
- Can you simplify when translating in only one direction? 4
- Do you have an existing DTD or EDD? 4

SGML application development 4

- You need a starting point: an EDD or DTD 6
- FrameMaker+SGML automatically translates between DTDs and EDDs 6
- You need to provide formatting information in FrameMaker+SGML 6
- You might need to change the default translation 6
- How you modify the translation 7
- What your end users do 7

Chapter 2 *The SGML and FrameMaker+SGML Models 9*

Structure descriptions 9

- FrameMaker+SGML EDDs 9
- SGML DTDs 9

Elements 10

- FrameMaker+SGML element types 10
- SGML elements 11
- Element declarations and definitions 11

Attributes 12

Entities 13

Documents 14

- SGML documents 14
- FrameMaker+SGML documents 14

Multiple-file documents 15

Format rules 15

Graphics 15

Equations 16

Tables 16

Cross-references 17

Marked sections and conditional text 17

Processing instructions 18

SGML features with no counterparts 18

Unsupported optional SGML features 19

Chapter 3 *Creating an SGML Application 21*

The development process 22

- Task 1. Producing an initial EDD and DTD 22

Task 2. Getting sample documents 25
Task 3. Creating read/write rules 26
Task 4. Finishing your application 29
For more information 31
Pieces of an SGML application 32
Application definition file 32
SGML external DTD subset 32
SGML declaration 32
FrameMaker+SGML template 33
SGML read/write rules document 33
Entity catalogs 33
Documentation 33
Creating a FrameMaker+SGML template 34
Cross-reference formats 34
Variables 36
Special text flows to format generated lists and indexes 37
HTML mapping for export 37

Chapter 4 *Working with Special Files 41*

Location of SGML files 41
Application definition file 42
Editing an application definition file 42
Contents of sgmlapps.fm 43
Defining an application 44
Providing default information 46
Specifying a document element 47
Specifying a read/write rules document 47
Specifying a DTD 48
Specifying a FrameMaker+SGML template 48
Specifying an SGML declaration 48
Specifying entities 49
Specifying entities through an entity catalog 49
Specifying the location of individual entities 51
Specifying names for external entity files 51
Specifying public identifiers 53
Specifying a search path for external entity files 53
Specifying a search path for including files in rules documents 55
Specifying an SGML API client 56
Specifying the character encoding for SGML files 56
Limiting the length of a log file 57
Log files 58
Generating log files 58
Messages in a log file 58
Using hypertext links 59
Setting the length of a log file 59
Other special files 59

Part II Working with an EDD 61

Chapter 5 *Developing an Element Definition Document (EDD) 63*

In this chapter 63
Overview of the development process 64
Creating or updating an EDD from a DTD 65
About the DTD 65
Read/write rules and the new EDD 65
Creating an EDD from a DTD 66
What happens during translation 66
Updating an EDD from a DTD 67
Log files for a translated DTD 67
Starting an EDD without using a DTD 68
Creating a new EDD 68
Exporting an Element Catalog to a new EDD 68

The Element Catalog in an EDD 69
High-level elements 69
All elements in the catalog 70
Defining preliminary settings in an EDD 77
Specifying whether to create formats automatically 77
Specifying whether to transfer HTML mapping tables 77
Setting an SGML application 77
Organizing and commenting an EDD 78
Writing element definitions 79
About element tags 80
Guidelines for writing element definitions 80
Defining a container, table or footnote element 81
Defining a Rubi group element 85
Defining an object element 86
Keyboard shortcuts for working in an EDD 89
Editing structure 89
Moving around the structure 90
Creating an Element Catalog in a template 90
Importing element definitions 91
Log files for imported element definitions 91
Debugging element definitions 92
Saving an EDD as a DTD for export 92
Read/write rules and the new DTD 93
Creating a DTD from an EDD 93
What happens during translation 93
SGML declarations 94
Log files for a translated EDD 94
Sample documents and EDDs 95

Chapter 6 *Structure Rules for Containers, Tables, and Footnotes 97*

In this chapter 97
Overview of EDD structure rules 98
Writing an EDD general rule 99
Syntax of a general rule for EDD elements 99
Restrictions on general rules for tables 102
Default general rules for EDD elements 103
Specifying validity at the highest level in a flow 104
Adding inclusions and exclusions 104
Inclusions 104
Exclusions 105
How content rules translate to SGML 106
Inserting descendants automatically in containers 106
Inserting table parts automatically in tables 108
Initial structure pattern 108
Default initial structure 110
Inserting Rubi elements automatically in Rubi groups 111
Initial structure pattern 111
Debugging structure rules 112

Chapter 7 *Text Format Rules for Containers, Tables, and Footnotes 113*

In this chapter 113
Overview of text format rules 114
How elements inherit formatting information 115
The general case 115
Inheritance in a table or footnote 117
Inheritance in a document within a book 118
Specifying an element paragraph format 119
Writing context-dependent format rules 120
All-contexts rules 121
Context-specific rules 121
Level rules 125
Nested format rules 128
Multiple format rules 128
Context labels 130

Defining the formatting changes in a rule 131
Paragraph formatting 131
Text range formatting 131
No additional formatting 132
Specifications for individual format properties 133
Basic properties 135
Font properties 137
Pagination properties 139
Numbering properties 140
Advanced properties 141
Table Cell properties 142
Asian Text Spacing properties 143
Writing first and last format rules 143
How first and last rules are applied 144
A first or last rule with an autonumber 145
Defining prefixes and suffixes 145
How prefix and suffix format rules are applied 146
A prefix or suffix for a text range 146
A prefix or suffix for a paragraph 147
A prefix or suffix for a sequence of paragraphs 147
A prefix or suffix for a text range or a paragraph 148
Attributes in a prefix or suffix rule 149
When to use an autonumber, prefix or suffix, or first or last rule 150
Defining a format change list 151
Setting minimum and maximum limits on properties 152
Debugging text format rules 154

Chapter 8 *Attribute Definitions 157*

In this chapter 157
Some uses for attributes 157
How an end user works with attributes 158
Writing attribute definitions for an element 159
Attribute name 160
Attribute type 160
Specification for a required or optional value 161
Hidden and Read-only attributes 162
List of values for Choice attributes 163
Range of values for numeric attributes 163
Default value 164
Using UniqueID and IDReference attributes 164
UniqueID attributes 166
IDReference attributes 168
Using attributes to format elements 169
Using attributes to provide a prefix or suffix 170

Chapter 9 *Object Format Rules 173*

In this chapter 173
Overview of object format rules 174
Context specifications for object format rules 175
All-contexts rules 175
Context-specific rules 175
Setting a table format 178
Specifying a graphic content type 179
Setting a marker type 180
Setting a cross-reference format 182
Setting an equation size 182
Specifying a system variable 183
Debugging object format rules 185

Part III Translating between SGML and

FrameMaker+SGML 187

Chapter 10 *Introduction to Translating between SGML and FrameMaker+SGML 189*

In this chapter 189

What you can do with SGML read/write rules 189

What you can do with SGML API clients 190

A detailed example 191

- DTD fragment 191
- Document instance 192
- EDD fragment 192
- Formatting and read/write rules 193
- FrameMaker+SGML document 194

Chapter 11 *SGML Read/Write Rules and Their Syntax 197*

In this chapter 197

The rules document 197

Rule order 199

Rule syntax 199

Case conventions 200

Strings and constants 200

- String syntax 200
- Constant syntax 201
- Variables in strings 201

Comments 202

Include files 202

Reserved element names 203

Commands for working with a rules document 203

Chapter 12 *Translating Elements and Their Attributes 205*

In this chapter 205

Default translation 206

- Translating model groups and general rules 206
- Translating attributes 207
- Naming elements and attributes 209
- Inclusions and exclusions 210
- Line breaks and record ends 211

Modifications to the default translation 211

- Renaming elements 211
- Renaming attributes 212
- Renaming attribute values 213
- Translating an SGML element to a footnote element 213
- Translating an SGML element to a Rubi group element 214
- Changing the declared content of an SGML element associated with a text-only element 215
- Retaining content but not structure of an element 216
- Retaining structure but not content of an element 216
- Formatting an element as a boxed set of paragraphs 217
- Suppressing the display of an element's content 217
- Discarding an SGML or FrameMaker+SGML element 218
- Discarding an SGML or FrameMaker+SGML attribute 218
- Specifying a default value for an attribute 219
- Changing an attribute's type or declared value 220
- Creating read-only attributes 221
- Using SGML attributes to specify FrameMaker+SGML formatting information 222

Chapter 13 *Translating Entities and Processing Instructions 225*

In this chapter 225

Default translation 226

On export to SGML 226

On import to FrameMaker+SGML 228

Modifications to the default translation 234

Specifying the location of entity declarations 235

Renaming entities that become variables 235

Translating SDATA entity references on import and export 235

Translating SDATA entities as FrameMaker+SGML variables 236

Translating SDATA entities as special characters in FrameMaker+SGML 237

Translating SDATA entities as FrameMaker+SGML text insets 238

Translating SDATA entities as FrameMaker+SGML reference elements 240

Translating external SGML text entities as text insets 241

Translating internal SGML text entities as text insets 241

Changing the structure and formatting of a text inset on import 243

Discarding external data entity references 244

Translating ISO public entities 244

Facilitating entry of special characters that translate as SGML entities 244

Creating book components from general entities 245

Discarding unknown processing instructions 245

Using entities for storing graphics or equations 245

Chapter 14 *Translating Tables 247*

In this chapter 247

Default translation 248

On import to FrameMaker+SGML 248

On export to SGML 251

Modifications to the default translation 252

Formatting properties for tables 252

Identifying and renaming table parts 256

Representing FrameMaker+SGML table properties as SGML attributes 257

Representing FrameMaker+SGML table properties implicitly in SGML 258

Adding format rules that use CALS attributes (CALS only) 259

Working with colspecs and spanspecs (CALS only) 260

Specifying which part of a table a row or cell occurs in 260

Specifying which column a table cell occurs in 261

Omitting explicit representation of table parts 262

Creating parts of a table even when those parts have no content 264

Specifying the ruling style for a table 266

Exporting table widths proportionally 267

Creating vertical straddles 267

Using a table to format an element as a boxed set of paragraphs 270

Creating tables inside other tables 272

Rotating tables on the page 272

Chapter 15 *Translating Graphics and Equations 273*

In this chapter 273

Default translation 274

Supported graphic file formats 274

General import and export of graphic elements 275

On export to SGML 276

On import to FrameMaker+SGML 282

Modifications to the default translation 284

Identifying and renaming graphic and equation elements 284

Exporting graphic and equation elements 285
Representing the internal structure of equations 286
Renaming SGML attributes that correspond to graphic properties 286
Omitting representation of graphic properties in SGML 288
Omitting optional elements and attributes from the default DTD declarations 288
Specifying the data content notation on export 289
Changing the name of the graphic file on export 290
Changing the file format of the graphic file on export 291
Specifying the entity name on export 294
Changing how FrameMaker+SGML writes out the size of a graphic 295

Chapter 16 *Translating Cross-References 297*

In this chapter 297
Default translation 297
On export to SGML 298
On import to FrameMaker+SGML 299
Modifications to the default translation 300
Translating SGML elements as FrameMaker+SGML cross-reference elements 300
Renaming the SGML attributes used with cross-references 300
Translating FrameMaker+SGML cross-reference elements to text in SGML 301
Maintaining attribute values with FrameMaker+SGML 301

Chapter 17 *Translating Variables and System Variable Elements 303*

In this chapter 303
Default translation 303
On export to SGML 304
On import to FrameMaker+SGML 305
Modifications to the default translation 305
Renaming or changing the type of entities when translating to variables 305
Translating SGML elements as system variable elements 306
Translating FrameMaker+SGML system variable elements to text in SGML 307
Translating FrameMaker+SGML variables as SDATA entities 307
Discarding FrameMaker+SGML variables 307

Chapter 18 *Translating Markers 309*

In this chapter 309
Default translation 309
On export to SGML 310
On import to FrameMaker+SGML 310
Modifications to the default translation 311
Translating SGML elements as FrameMaker+SGML marker elements 311
Writing SGML marker text as element content instead of as an attribute 311
Using SGML attributes and FrameMaker+SGML properties to identify markers 312
Discarding non-element FrameMaker+SGML markers 313

Chapter 19 *Processing Multiple Files as Books 315*

In this chapter 315
Default translation 316
On import to FrameMaker+SGML 316
On export to SGML 318
Modifications to the default translation 319
Using elements to identify book components on import 319

Suppressing the creation of processing instructions for a book on export 321

Chapter 20 *Read/Write Rules Summary 323*

Chapter 21 *Read/Write Rules Reference 333*

anchored frame 333

attribute 335

character map 337

convert referenced graphics 340

do not include dtd 341

do not include sgml declaration 341

do not output book processing instructions 341

drop 342

drop content 344

element 345

end vertical straddle 348

entity 349

entity name is 352

equation 355

export dpi is 357

export to file 359

external data entity reference 362

external dtd 363

facet 364

fm attribute 366

fm element 367

fm marker 368

fm property 370

fm variable 372

fmsgml version 373

generate book 373

implied value is 377

include dtd 379

include sgml declaration 380

insert table part element 381

is fm attribute 385

is fm char 387

is fm colspec 389

is fm cross-reference element 389

is fm element 391

is fm equation element 392

is fm footnote element 393

is fm graphic element 394

is fm marker element 395

is fm property 396

is fm property value 398

is fm reference element 402

is fm rubi element 404

is fm rubi group element 405

is fm spanspec 406

is fm system variable element 407

is fm table element 408

is fm table part element 409

is fm text inset 411

is fm value 413

is fm variable 414

is processing instruction 415

line break 417

marker text is 418

notation is 420

output book processing instructions 422

preserve fm element definition 422

preserve line breaks 424

processing instruction 425

proportional width resolution is 426

put element 427

reader 427

reformat as plain text 429

reformat using target document catalogs 429

retain source document formatting 430

specify size in 431

start new row 433

start vertical straddle 434

table ruling style is 435

unwrap 436

use processing instructions 438

use proportional widths 438

value 439

value is 441

write sgml document 441

write sgml document instance only 441

writer 442

Appendix A *Conversion Tables for Adding Structure to Documents 445*

How a conversion table works 445

Setting up a conversion table 446

- Generating an initial conversion table 447
- Setting up a conversion table from scratch 448
- Updating a conversion table 448

Adding or modifying rules in a conversion table 448

- About tags in a conversion table 449
- Identifying a document object to wrap 450
- Identifying an element to wrap 451
- Identifying a sequence to wrap 452
- Providing an attribute for an element 453
- Using a qualifier with an element 454

Handling special cases 455

- Promoting an anchored object 455
- Flagging format overrides 456
- Wrapping untagged formatted text 457
- Nesting object elements 457
- Building table structure from paragraph format tags 458

Testing and correcting a conversion table 459

Appendix B *The CALS Table Model 461*

FrameMaker+SGML properties that DO NOT have corresponding CALS attributes 461

Element and attribute definition list declarations 463

Element structure 465

Attribute structure 465

- Inheriting attribute values 466
- Orient attribute 466
- Straddling attributes 466

Appendix C *SGML Read/Write Rules for CALS Table Model 467*

Appendix D *SGML Declaration 471*

Text of the default SGML declaration 471

SGML concrete syntax variants 473

Unsupported optional SGML features 474

Appendix E *Character Set Mapping 475*

Appendix F *ISO Public Entities 483*

What you need to use ISO public entities 484

- Entity declaration files 485

Entity read/write rules files 485

What happens with the declarations and rules 488

Appendix G *SGML Batch Utilities for UNIX 493*

Importing SGML documents in batch mode 493

Exporting documents as SGML in batch mode 495

Appendix H *Using the XML Export Feature 497*

Terms used in this appendix 498

Restrictions 499

XML syntax 499

Empty elements 499

CSS Style sheets 499

XML character encoding 503

XML read/write rules 504

Handling invalid element names 505

Default attribute value 505

Appendix I *Developing SGML Publishing Applications 507*

Implementing an SGML application in the FrameMaker+SGML publishing environment 507

Overview of FrameMaker+SGML Application Development 507

Technical Steps in FrameMaker+SGML Application Development 516

Typical Application Development Scenarios 525

Conclusions 529

Glossary 531

Index 539

Before You Begin

This manual is for developers of structured FrameMaker+SGML templates and of SGML applications. It is not for end users who work with structured documents that use such templates and applications.

Developing structured FrameMaker+SGML templates

End users of FrameMaker+SGML can read, edit, format, and write structured documents with FrameMaker+SGML. For each document, they need a template that contains a catalog of the elements that make up the structure and describes the formatting of those elements in various contexts. In support of these end users, you create the catalog and accompanying structured template.

Developing SGML applications

End users can also read and write SGML (Standard Generalized Markup Language) documents with FrameMaker+SGML. When the software reads an SGML document, the document appears as a formatted structured document. When the software writes a formatted structured FrameMaker+SGML document, the document can appear as an SGML document.

For the end user, this process of translation between FrameMaker+SGML documents and SGML documents is transparent and automatic. However, for most SGML document types a small part of the software that manages the translation—an SGML application—has to be specifically developed. If the end users that you support use FrameMaker+SGML to read and write SGML documents, you create this application. The application primarily consists of a structured template, special rules described in this manual, and for some situations, an SGML API client 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 SGML application:

- Structured document creation
- SGML concepts and syntax, including how to work with a *document type declaration*
- FrameMaker+SGML end-user concepts and syntax
- FrameMaker+SGML template design

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

- C programming
- FDK API usage

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

Using FrameMaker+SGML documentation

FrameMaker+SGML comes with a complete set of end-user and developer documentation with which you should be familiar. If you use the Frame Developer's Kit in creating your SGML application, you'll also need to be familiar with the FDK documentation set.

Using this manual

This manual is divided into three major parts and a series of appendixes. If you're creating an SGML application, you'll find information you need in all three parts. If, however, you're not working with SGML but are creating a structured template, you'll need only Part II. The parts are as follows:

- Part I, "Developing a FrameMaker+SGML application"

  Part I is for developers of SGML applications. It has introductory information, an overview of the steps in creating an SGML application, a comparison of SGML and FrameMaker+SGML concepts, and details of putting together the pieces of an application into a whole.

- Part II, "Working with an EDD"

  Part II is for developers of a FrameMaker+SGML structured template. It contains information on how you use an element definition document (EDD) to define elements and determine their formatting for your documents. You use this part in conjunction with chapters in the FrameMaker user's manual that describe other aspects of template creation.

- Part III, "Translating between SGML and FrameMaker+SGML"

  Part III is for developers of SGML applications. FrameMaker+SGML's default translation between SGML documents and FrameMaker+SGML documents follows a model. This part describes the model and the rules you use to modify the default translation.

- Appendixes

  The appendixes include information such as how to add structure to unstructured FrameMaker+SGML documents and how to work with ISO public entities. There is also a glossary of terms.

If you're creating an SGML application, we encourage you to read the first three chapters of Part I before reading any of the rest of the manual. For your purposes, these chapters are a prerequisite to the rest of the manual.

Each of the three parts of this manual has a short introduction in which you can find information about specific chapters in that part.

Using other FrameMaker+SGML manuals

The *FrameMaker User Guide* and the *FrameMaker+SGML User Guide* make up the primary end-user documentation for this product. Together they explain how to use the FrameMaker+SGML authoring environment for both structured and unstructured documents. They also explain how to create templates for your documents.

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

Using FDK manuals

If you create an SGML API client for your 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+SGML 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 *SGML API Programmer's Guide*.)
- *FDK Programmer's Reference* is a reference for the functions and objects described in the *FDK Programmer's Guide*.
- *SGML API Programmer's Guide* explains how to use the FDK to make advanced modifications to the software's default behavior for translation between SGML documents and FrameMaker+SGML 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 Developing a FrameMaker+SGML application

Part I provides basic information for developing SGML applications, manual including:

- Chapter 1, “SGML Application Basics”

  Describes situations that require an SGML application. Also contains a high-level description of application creation.

- Chapter 2, “The SGML and FrameMaker+SGML Models”

  Compares relevant SGML and FrameMaker+SGML concepts. You should read this chapter even if you are already familiar with both SGML and FrameMaker+SGML, since translation between the two is its own distinct topic. The chapter deals with counterpart constructs in the two representations, and also with the more difficult case for translation purposes, constructs in one that have no real counterpart in the other.

- Chapter 3, “Creating an SGML Application”

  Describes typical application creation workflow. Also discusses the types of files used in your final SGML application.

- Chapter 4, “Working with Special Files”

  Tells where to find special files used by FrameMaker+SGML with SGML documents. Also explains creation of the file that defines the pieces of your SGML application.

1 SGML Application Basics

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

Where to begin

If your end users do not need SGML, but use FrameMaker+SGML only because they want the benefits inherent in using a structured authoring environment, you only need to create an *element definition document (EDD)* and a structured template for them. In this case, the material you need is contained in Part II, “Working with an EDD.”

If your end users need to read or write SGML documents, you will probably need to develop an SGML application to modify the default translation between SGML documents and FrameMaker+SGML documents. In that case, this entire manual is of use to you. The rest of Part I provides general information on creating an SGML application. Part III, “Translating between SGML and FrameMaker+SGML,” provides details of the default translation between SGML and FrameMaker+SGML and explains how you can change this default behavior with SGML read/write rules.

SGML application scenarios

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

Translating in one or two directions?

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

If end users always work in one environment and documents developed in one representation are simply filtered to the second representation, you have to prepare an application to translate documents only in one direction. For example, your company may already use FrameMaker+SGML or FrameMaker and now has a requirement for SGML delivery. In this situation, your application only needs to be able to export FrameMaker+SGML documents using an appropriate document type definition (DTD). It doesn’t have to be able to import SGML documents into FrameMaker+SGML.

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

Or perhaps your end users work with FrameMaker+SGML to create their documents, but they collaborate with other writers who use a different authoring environment for SGML documents. In this situation your application needs both to read SGML documents into FrameMaker+SGML and to write FrameMaker+SGML to SGML, 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+SGML documents to deliver in SGML, they developed documents using the full power of FrameMaker+SGML. They may well have used such FrameMaker+SGML features as markers to facilitate document creation. But some of these features may not be relevant in an SGML representation, so you can choose to omit them from the SGML document. If you won't be re-importing the SGML documents back into FrameMaker+SGML, the resultant loss of information is not important. Therefore, you don't have to retain the use of those features in your SGML application.

Do you have an existing DTD or EDD?

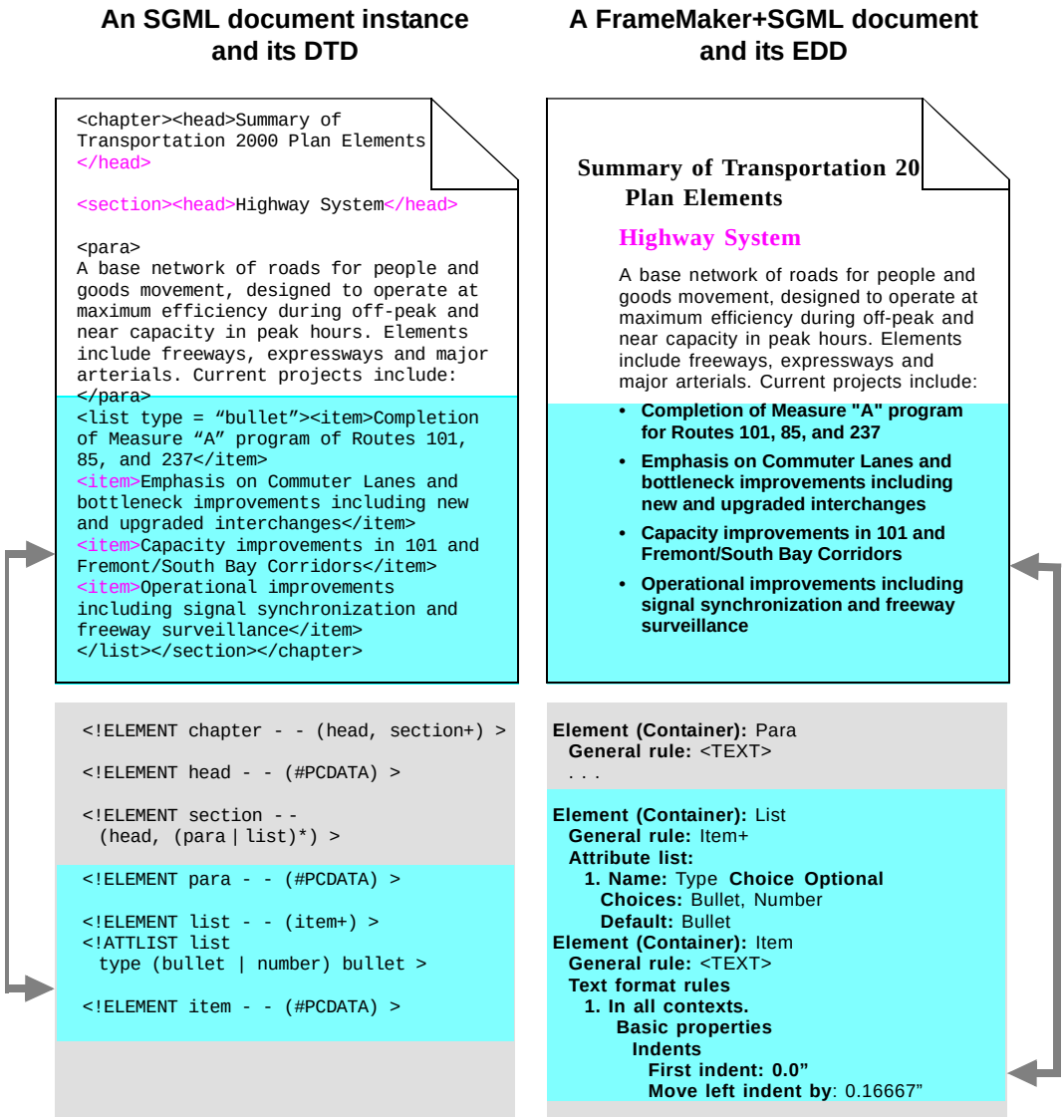
Another factor influencing the design of your application is the degree to which the document models on the SGML and FrameMaker+SGML sides are already established. Is your starting point an established EDD or DTD—for instance, one of the Department of Defense *CALS* DTDs? Will you be provided with both an established EDD and an existing body of documents that use that EDD? Or has neither definition document as yet been written?

The most common situation is to start with a document model in one representation or the other. In this case, your application must deal with the special problems of translation that the established DTD or EDD presents. For example, if your end users already have a library of documents using a particular DTD, 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 will be easier.

SGML application development

SGML and FrameMaker+SGML have many similarities. To describe the hierarchical nature of documents, both use special methods to define the components that can occur in a

document instance and the relationship among them. In SGML, these constructs are grouped in a DTD, and in FrameMaker+SGML they are grouped in an EDD. Within the content of particular document instances, both use descriptive tags to identify structural components. The tags associated with the document's contents correlate to the model of the document found in the DTD or the EDD, as in the illustration below. The lists in the documents in the top half of the illustration correspond to the DTD and EDD models of lists in the bottom half of the illustration.



Despite the similarities in their basic approach to the creation of structured documents, SGML and FrameMaker+SGML differ in their methods of representing the contents of

documents and in the aspects of documents they represent. In this respect, SGML and FrameMaker+SGML 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+SGML and SGML, the software needs instructions to help it translate between the two languages.

You need a starting point: an EDD or DTD

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 an SGML DTD or a FrameMaker+SGML EDD.

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

FrameMaker+SGML automatically translates between DTDs and EDDs

FrameMaker+SGML automatically translates both the SGML and FrameMaker+SGML definition documents and individual document instances into their counterparts in the other representation. If you start with a DTD, the software creates a default version of the corresponding EDD. Similarly, if you start with an EDD, FrameMaker+SGML creates a default version of the DTD.

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

You need to provide formatting information in FrameMaker+SGML

You create a structured template to provide appropriate formatting for your documents. Since the SGML DTD typically provides no formatting information, this task is independent of whether or not the default translation is correct for your DTD.

You may create multiple structured templates for the same SGML DTD. For example, your end users can use the same SGML source documents for different purposes, such as a printed manual and an online help system. They may want the same SGML document formatted differently in the two situations.

You might need to change the default translation

The structure and formatting of SGML documents varies widely, so most of your job in creating an SGML application is to change the default translation FrameMaker+SGML uses. You do so by providing information the software needs to recognize and process particular

constructs. If FrameMaker+SGML automatically translates all the components of a document just as you want, you don't need to provide extra information.

One of the differences between SGML and FrameMaker+SGML is that FrameMaker+SGML, but not SGML, has explicit representations for items such as tables or graphics. (With SGML, an individual DTD defines a representation appropriate for its particular use.) For this reason, FrameMaker+SGML has to assume certain DTD representations of these items for its default translation to include the items, and 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+SGML translates documents, you start by using special *SGML read/write rules*. The default translation as modified by rules is sufficient to represent many variations, and most of your effort in creating an SGML application consists of specifying these rules.

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

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

What your end users do

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

2 The SGML and FrameMaker+SGML Models

SGML and FrameMaker+SGML use models for structured documents that are sometimes similar and sometimes substantially different. This chapter describes the similarities and differences between the two models that you need to understand before you can create an SGML application.

Structure descriptions

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

FrameMaker+SGML EDDs

In FrameMaker+SGML, 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+SGML document that stores information about 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 information on creating EDDs, see Part II, "Working with an EDD."

SGML DTDs

In SGML, 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, an SGML 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. SGML has nothing to say about the semantics of the process, that is, the meaning of those pieces.

Before its document instance, an SGML 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 SGML document that uses the DTD, the SGML standard allows 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 ] >
```

where *name* is the document type name and *extid* is the external identifier of an entity that consists of a list of declarations. 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.

Elements

In both SGML and FrameMaker+SGML, 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+SGML distinguishes among several specific element types, corresponding to special constructs such as tables or cross-references. SGML does not make such a distinction.

FrameMaker+SGML element types

A large proportion of the elements in a structured FrameMaker+SGML document are defined to include text, child elements, or both. FrameMaker+SGML 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

- graphics
- tables
- table parts (such as table footings)

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

FrameMaker+SGML 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+SGML 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 5, "Developing an Element Definition Document (EDD)." For information on translating between FrameMaker+SGML and SGML elements of various types, see Part III, "Translating between SGML and FrameMaker+SGML."

SGML elements

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

Element declarations and definitions

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

Element content

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

A FrameMaker+SGML 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). An SGML element's content model can include either the reserved name `ANY` ("anything") or a combination of child elements. In some element definitions in both SGML and FrameMaker+SGML, the content rule also specifies what content is required or optional and the required order of the content.

Inclusions and 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.

Other information

Finally, SGML declarations and FrameMaker+SGML definitions each provide some information not relevant to the other. For example, an SGML element declaration specifies whether or not the element's start-tag or end-tag can be omitted. In FrameMaker+SGML, because it is a WYSIWYG tool, the user does not explicitly enter markup such as start-tags and end-tags. Therefore, the omission of a start-tag or end-tag has no meaning in FrameMaker+SGML.

As another example, a FrameMaker+SGML element definition can have *format rules*. But because SGML does not address document formatting, format rules have no direct counterpart in SGML and are discarded when exporting to SGML.

For information on creating element definitions, see Chapter 6, "Structure Rules for Containers, Tables, and Footnotes."

Attributes

FrameMaker+SGML and SGML 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. Another common use of attributes is to specify formatting information.

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

By default, FrameMaker+SGML translates most attributes in one representation as attributes of the same name in the other representation. However, you may decide to supply rules to change this behavior. Some attributes in SGML represent information best represented by other means in FrameMaker+SGML. 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+SGML attribute.

For information on defining attributes in an EDD, see Chapter 8, “Attribute Definitions,” in this manual. For information on translating between FrameMaker+SGML and SGML attributes, see Chapter 12, “Translating Elements and Their Attributes.”

Entities

An SGML *entity* is a collection of characters you reference as a unit. Entities are used for many purposes in SGML. 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+SGML provides several features for handling situations for which you use entities in SGML.

Entities in SGML 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+SGML doesn’t have a single construct called an entity, it provides the functionality of many SGML entities through various mechanisms. Entity functions include the following:

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

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

Graphics In SGML, you often store graphics in separate files and then include them in the document with general entity name attributes. In FrameMaker+SGML, 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+SGML these might be either variables or characters in a particular font.

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

For information on creating variables, text insets, and books, see the *FrameMaker User Guide*. For information about structured books, see the *FrameMaker+SGML User Guide*.

For information on translating entities to various FrameMaker+SGML constructs, see Chapter 13, "Translating Entities and Processing Instructions," Chapter 15, "Translating Graphics and Equations,"Chapter 17, "Translating Variables and System Variable Elements." and Chapter 19, "Processing Multiple Files as Books."

Documents

SGML and FrameMaker+SGML use the term *document* differently. An SGML document has a particular set of parts in a particular order and can be spread across multiple physical files. A FrameMaker+SGML document is simply a single file in FrameMaker+SGML format.

SGML documents

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, SGML applications can use the *reference concrete syntax* defined in the SGML standard.

The prolog and document instance set allowed with FrameMaker+SGML have the simplest form defined in the standard. For a document used with FrameMaker+SGML, 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 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+SGML documents

Since FrameMaker+SGML is a WYSIWYG tool, a FrameMaker+SGML document is organized differently than an SGML document. A FrameMaker+SGML 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 SGML and FrameMaker+SGML allow a single conceptual document to be divided into multiple physical files.

FrameMaker+SGML 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+SGML document and can stand on its own.

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

For information on creating FrameMaker+SGML books, see the FrameMaker and FrameMaker+SGML user guides. For information on creating books from SGML text entities, see Chapter 19, "Processing Multiple Files as Books."

Format rules

SGML has no standard mechanism for representing the formatting of a document. Some DTDs use attributes for some formatting information.

FrameMaker+SGML provides format rules that allow an EDD to store context-sensitive formatting information. When you edit a document and insert an element with format rules, FrameMaker+SGML applies the appropriate format to the element's content on the basis of surrounding structure and attribute values. That is, FrameMaker+SGML 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+SGML 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 7, "Text Format Rules for Containers, Tables, and Footnotes," and Chapter 9, "Object Format Rules."

Graphics

SGML doesn't have a standard mechanism for representing graphics. There are several common methods in use, in each of which an entity holds the graphic object itself. That

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

FrameMaker+SGML provides tools for creating a graphic. Alternatively, your users can import an external graphic, either by copying it directly into your FrameMaker+SGML 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+SGML recognizes a graphic in several file formats, such as MIF or Sun raster format. Because FrameMaker+SGML 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 15, "Translating Graphics and Equations."

Equations

As with graphics, SGML has no standard mechanism for representing equations, while FrameMaker+SGML has a complete tool for working with them. Once created, however, equations in FrameMaker+SGML have characteristics very similar to graphics. For this reason, FrameMaker+SGML treats equations in essentially the same way as graphics for SGML import and export, and this manual discusses them together.

For information on creating graphics and equations in FrameMaker+SGML, see the *FrameMaker User Guide*.

For information on translating equations, see Chapter 15, "Translating Graphics and Equations."

Tables

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

SGML 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* table model, which is similar to the table description supported by FrameMaker+SGML. 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+SGML 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+SGML, see the *FrameMaker User Guide*. For information on defining tables and table parts in an EDD, see Chapter 6,

"Structure Rules for Containers, Tables, and Footnotes." For information on translating tables, see Chapter 14, "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 SGML standard 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+SGML can use this model for cross-references within a FrameMaker+SGML document.

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

- In the `SGML ID/IDREF` model, both the cross-reference and its source are elements. In FrameMaker+SGML the source of the cross-reference can be an element but can also be a paragraph or a spot in a paragraph.
- In the `SGML ID/IDREF` model, attributes contain the information connecting a cross-reference to its source. In FrameMaker+SGML you can also store the information in markers instead.
- The `ID/IDREF` mechanism is natural in SGML for *internal cross-references*, those in which the source and the cross-reference are in the same document. However, it cannot be used with *external cross-references*, those in which the source and the cross-reference are in different documents. FrameMaker+SGML provides a single mechanism for both internal and external cross-references.
- FrameMaker+SGML and SGML have different views of what constitutes an internal or external cross-reference. In FrameMaker+SGML, a cross-reference to a different file is always an external cross-reference. In SGML, cross-references to different entities in a single document are always internal cross-references. So cross-references between components in a FrameMaker+SGML book are considered external, but cross-references between the SGML text entities that correspond to those components are internal, since the entire book translates to a single SGML document.
- SGML allows a single attribute value to link several sources. FrameMaker+SGML requires a separate cross-reference for each citation.

For information on creating cross-references in FrameMaker+SGML documents, see the *FrameMaker User Guide*. For information on translating cross-references to SGML, see Chapter 16, "Translating Cross-References."

Marked sections and conditional text

Both SGML and FrameMaker+SGML 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+SGML, it is *conditional text*. The details of these two mechanisms

differ significantly. For this reason, when translating between FrameMaker+SGML and SGML, the software does not attempt to preserve the fact that information was tagged as conditional text in a FrameMaker+SGML 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+SGML 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+SGML encounters conditional text that is hidden when writing a FrameMaker+SGML 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 an SGML API client to change the exported document instance to reflect condition tags.

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

Processing instructions

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

For information on handling processing instructions, see Chapter 13, "Translating Entities and Processing Instructions," and Chapter 19, "Processing Multiple Files as Books."

SGML features with no counterparts

SGML features with no FrameMaker+SGML counterparts include:

- Internal parameter entity declarations or references
- Notation declarations
- Short reference mapping or usage
- Markup minimization
- Content references

FrameMaker+SGML 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+SGML's processing can't distinguish whether or not the start-tags were omitted.

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

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+SGML:

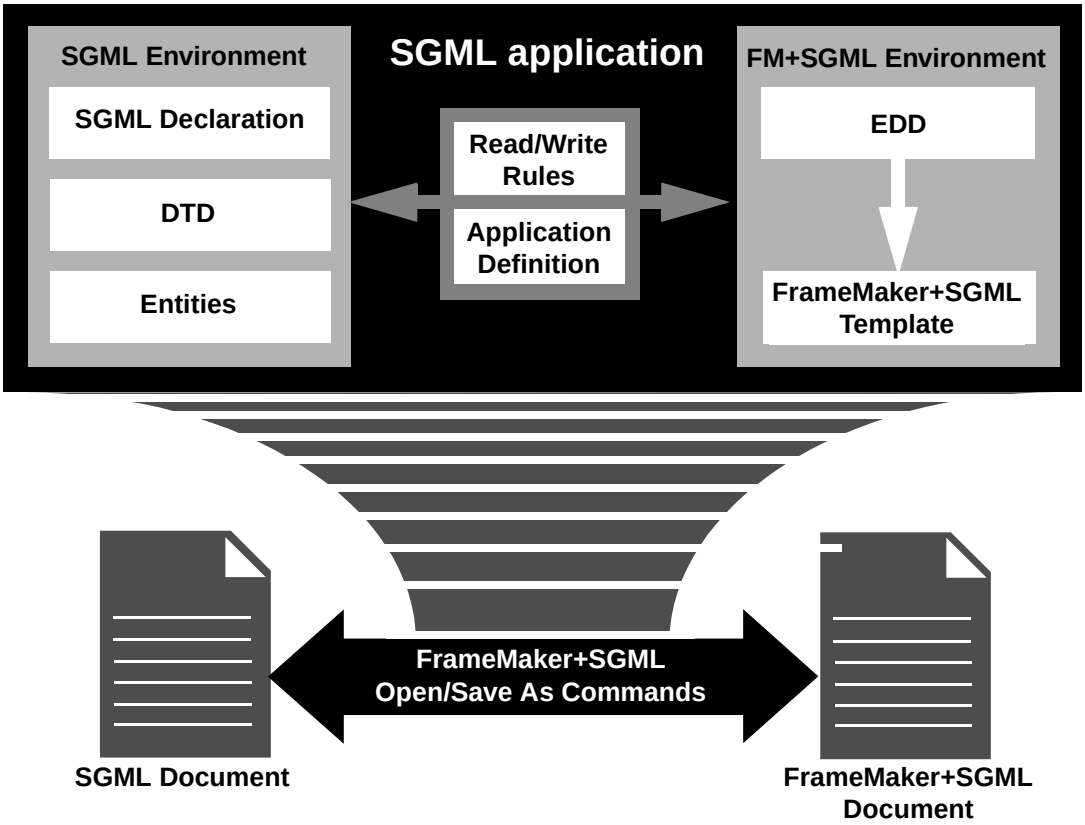
- `DATATAG`
- `RANK`
- `LINK`
- `SUBDOC`
- `CONCUR`

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

3 Creating an SGML Application

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

The diagram shows your completed SGML application at work. Information in the application funnels into the Open and Save As commands in FrameMaker+SGML, adapting their default translation behavior to your SGML and FrameMaker+SGML environments. Your SGML application has information (such as a DTD) specific to the SGML representation of documents, other information (such as an EDD) specific to the FrameMaker+SGML representation of documents, and rules and definition to bridge this information. The application pieces are described in more detail in "Pieces of an SGML application" on page 32.



The development process

As an SGML application developer, your primary task is to modify the default translation behavior of FrameMaker+SGML to fit the circumstances of your application. This section gives an overview of the steps used to create an SGML 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.

2. Get sample documents to test your application.

   If you don't have sample SGML documents and FrameMaker+SGML 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 SGML read/write rules to modify how the software translates between SGML and FrameMaker+SGML.

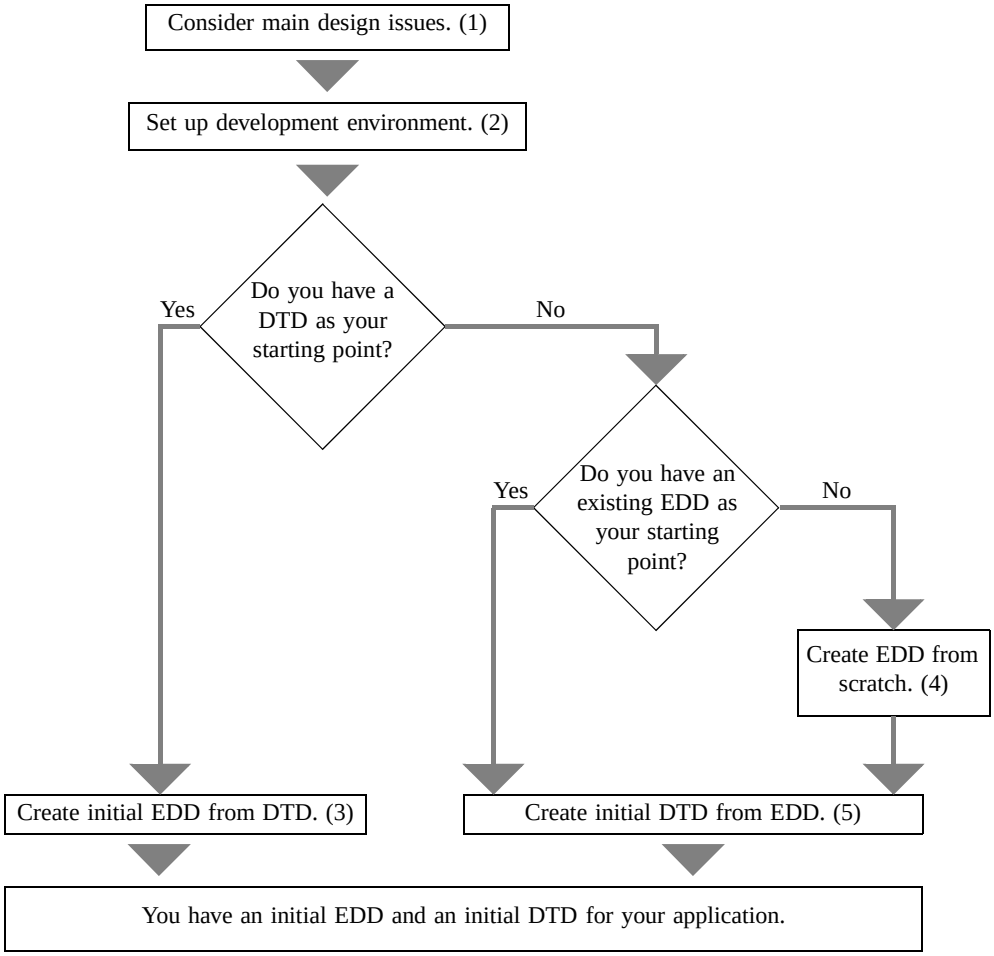
4. Use the FDK and FrameMaker+SGML tools to finish your application.

   Once you've done as much as you can with rules, you may have to write an *SGML API client* to further modify the software's behavior. Consider other modifications you want, such as customizing the FrameMaker+SGML 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 31.

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+SGML and SGML versions of the same document
- the environment (SGML or FrameMaker+SGML) in which documents will be created and delivered
- whether end users work with only one SGML application or multiple applications

(2) Set up the development environment.
You or your system administrator must install FrameMaker+SGML.

Once FrameMaker+SGML 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 SGML 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 an EDD, DTD, and 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 File>Developer Tools>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 SGML constructs into FrameMaker+SGML constructs.

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

- SGML elements to FrameMaker+SGML elements of the same name
- SGML attributes to FrameMaker+SGML attributes, assuming that attributes contain extra information about their associated elements
- entities to various constructs such as variables

The software produces a log file if it encounters any problems while processing your 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 an SGML application. Because of the richer semantics available in an EDD, you should first create an EDD and its associated FrameMaker+SGML 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+SGML 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 File>Developer Tools>Save As DTD to create an initial DTD to see how the software translates your EDD with no help from your application.

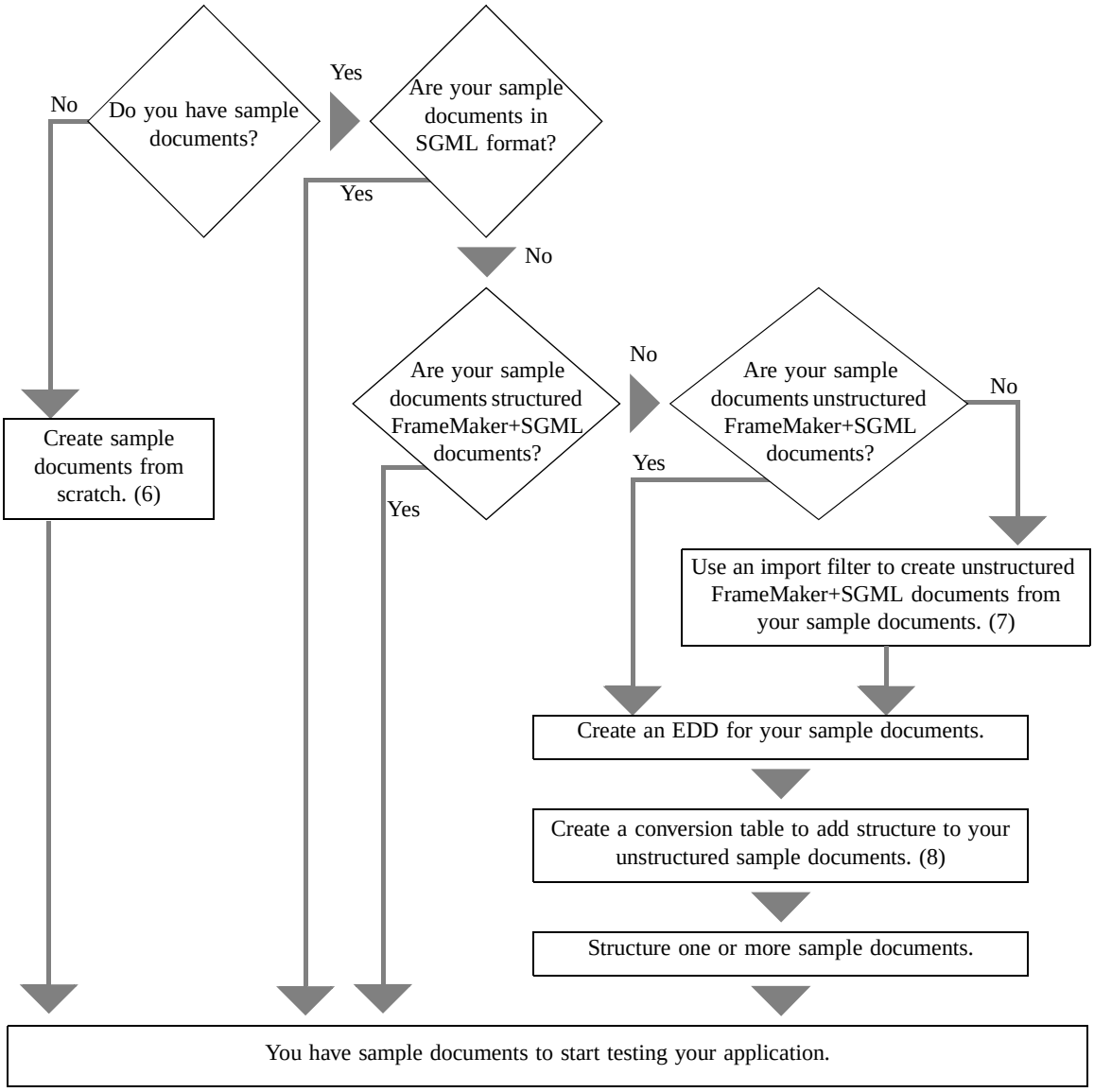
The DTD has element and attribute definition list declarations corresponding to the element and attribute definitions in the EDD. The software reads through the EDD, processing the elements and their attributes one at a time. For each definition, the software

- creates an SGML 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+SGML 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 SGML documents, you probably won't yet have FrameMaker+SGML documents. If you have existing unstructured FrameMaker+SGML 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 SGML documents. If you're starting with a preexisting EDD, create structured FrameMaker+SGML documents.

(7) Use document import filters to get FrameMaker+SGML documents.
If your sample documents are in a file format other than SGML or FrameMaker+SGML, you should convert them to unstructured FrameMaker+SGML documents and then use the software's tools for structuring unstructured documents.

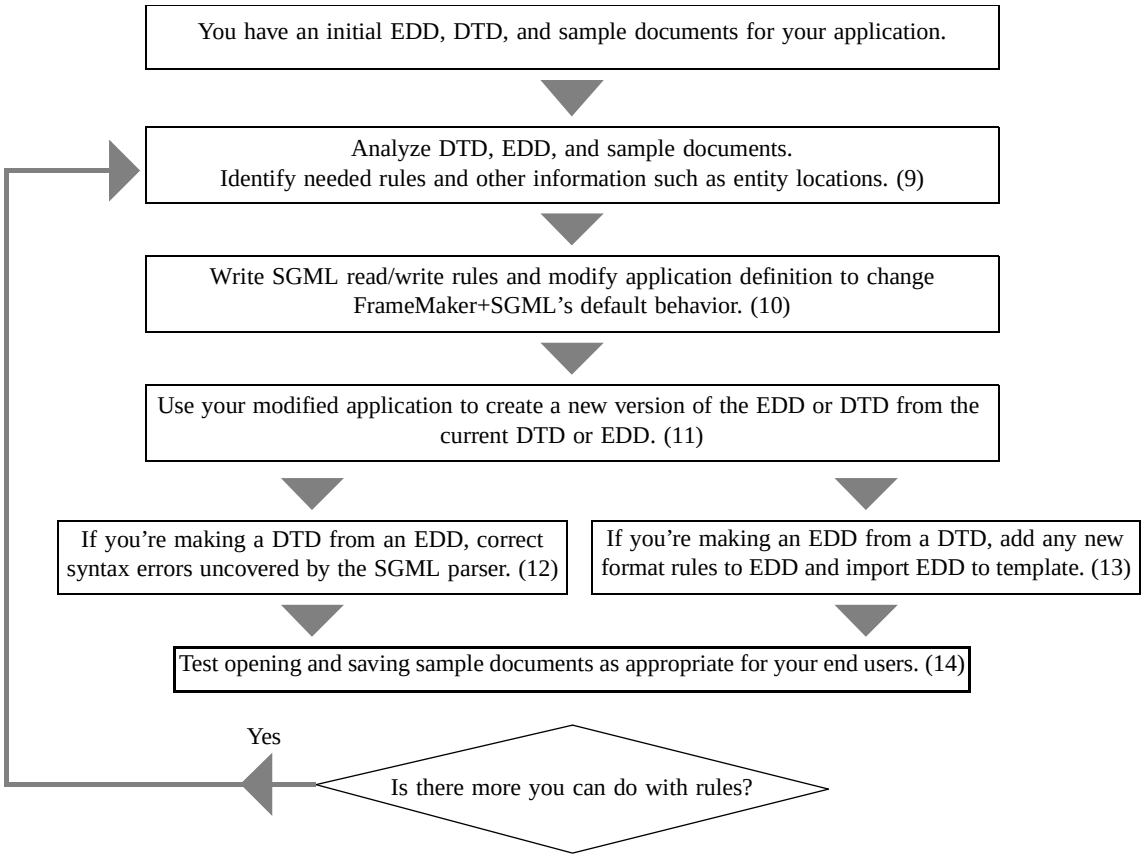
(8) Structure unstructured FrameMaker+SGML documents if necessary.
If you've gotten to this point, you have unstructured FrameMaker+SGML documents to use as your sample documents. FrameMaker+SGML provides tools to aid you in structuring unstructured documents.

You create a conversion table that describes a set of unstructured documents and then use FrameMaker+SGML 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+SGML's conversion tools, you should use the resulting EDD as one of the starting points of your SGML 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+SGML translates between them. This portion of application development is an iterative process. This 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+SGML 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.

   Determine the level of minimization provided by the DTD. *Markup minimization* is irrelevant in FrameMaker+SGML'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 an SGML API client to do so.

Determine the purpose of elements with a declared content of `EMPTY`. These will translate to the appropriate FrameMaker+SGML 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+SGML document.

3. Examine entity declarations.

   Determine how entities are used and which become text, variables, graphics, book files, or special characters in FrameMaker+SGML. The SGML standard defines several sets of character entities. Check whether your DTD refers to any of these character sets.

4. Examine notation declarations.

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

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

   FrameMaker+SGML assumes SGML 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 an SGML API client. 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 an SGML 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+SGML 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 SGML parser.
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.

The result of this flexibility is that correctly interpreting the markup of a particular DTD or document instance can be quite difficult. SGML systems use a separate program or section of code, called the *SGML parser*, to interpret markup in a standard way.

In particular, FrameMaker+SGML uses a parser for this purpose. When it processes a DTD or an SGML 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 rules, but others may required you to change the quantity and capacity limits defined in the SGML declaration.

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+SGML template to include appropriate format rules.

FrameMaker+SGML 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.)

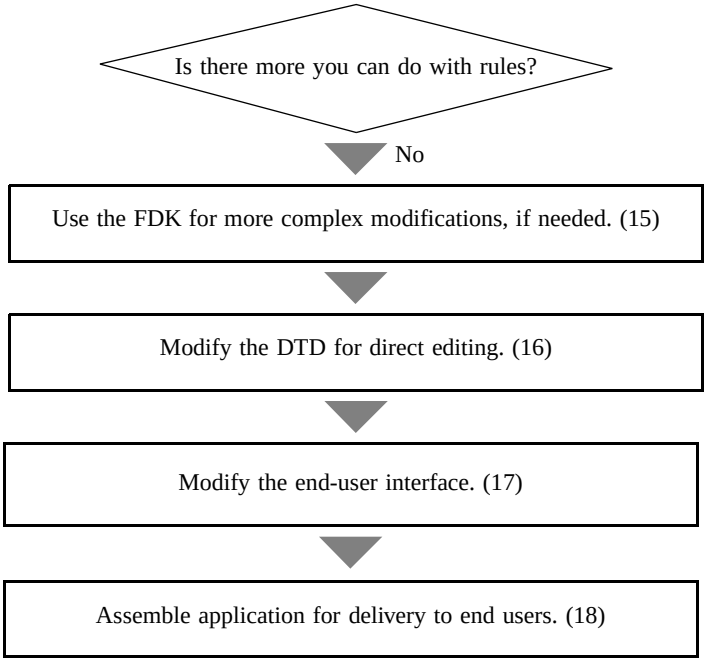
(14) Test with sample documents.
You should test your application at various stages of its development and you should test your SGML read/write rules thoroughly.

If end users will be opening SGML documents in FrameMaker+SGML, invoke the software with sample documents that test your modifications in that direction. If users will be saving FrameMaker+SGML documents as SGML, 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+SGML 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+SGML'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 the FDK for more complex modifications.
In some situations, SGML read/write rules are not sufficient to describe the appropriate translations between SGML and FrameMaker+SGML. For example, your 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 an SGML API client.

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

(16) Modify the DTD for direct editing.
If you've created a DTD from an EDD and your end users will be editing SGML documents directly, you may want to make modifications to the DTD for ease of use. For example, you may choose to change some element definitions to allow tag omission. Furthermore, you may wish to add short reference mapping or to use additional declarations in the DTD to make it easier for your end users.

(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 the Developer Tools submenu from the File menu. This submenu contains only developer commands. 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+SGML	*Installing* FrameMaker+SGML
	Creating an application definition	"Application definition file" on page 42
3, 4	Creating an EDD	Part II, "Working with an EDD"
	Working with log files	"Log files" on page 58
7	Using document filters available with FrameMaker+SGML	*Using Filters*
8	Creating an EDD and FrameMaker+SGML template	Part II, "Working with an EDD"
	Creating an EDD for an existing set of unstructured documents	Appendix A, "Conversion Tables for Adding Structure to Documents"
9-14	How FrameMaker+SGML translates SGML documents and how you can modify that with rules	Part III, "Translating between SGML and FrameMaker+SGML"
	Writing format rules	Chapter 7, "Text Format Rules for Containers, Tables, and Footnotes," and Chapter 9, "Object Format Rules"
	Importing an EDD into a FrameMaker+SGML template	Chapter 5, "Developing an Element Definition Document (EDD)"
15	Using the FDK while creating an SGML application	*SGML API Programmer's Guide*
17	Modifying the user interface	*Customizing FrameMaker Products* (for Macintosh or WIndows®) or *Changing Setup Files* (for UNIX®)
18	Assembling your application	"Pieces of an SGML application," next

Pieces of an SGML application

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

The following sections describe the possible parts of an SGML application and your options for assembling those parts into a working application. Although you can include an SGML 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+SGML with these paths and other information by placing them in the `sgmlapps.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 SGML applications, put information about all of them in a single `sgmlapps.fm` file. FrameMaker+SGML reads this file on startup, so your end users can choose any applications defined in this file. You need to deliver this `sgmlapps.fm` file to your end users.

For information on the `sgmlapps.fm` file, see "Application definition file" on page 42.

SGML external DTD subset

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

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

SGML declaration

An SGML 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 application nor a particular SGML document specifies a declaration, FrameMaker+SGML uses the SGML declaration described in Appendix D, "SGML Declaration."

When writing an SGML document, if you do not specify an SGML declaration in the application definition, the software uses the declaration defined in that appendix.

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

FrameMaker+SGML template

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

As you create your application, you can work with a FrameMaker+SGML EDD in a separate file. However, FrameMaker+SGML does not use a separate EDD file to find information on structure when opening an SGML document; it gets the information directly from the FrameMaker+SGML template. At some point during the creation of the FrameMaker+SGML 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.

SGML read/write rules document

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

Because an SGML 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 13, "Translating Entities and Processing Instructions," Chapter 4, "Working with Special Files," and Appendix F, "ISO Public Entities."

Documentation

As part of your SGML 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 SGML applications. Refer end users to the FrameMaker and FrameMaker+SGML user guides. You can remove this manual from your end users' installation directory. However, the Online Manuals menu in FrameMaker+SGML Help will still have an item for it. If an end user clicks on that menu item after you've removed the manual, an alert appears on the user's screen.

Creating a FrameMaker+SGML template

You should deliver your FrameMaker+SGML template to end users as a separate file. The template is a FrameMaker+SGML 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 90

For information on creating a FrameMaker+SGML template, see the FrameMaker and FrameMaker+SGML user guides. For information on creating the EDD for a template, see Chapter 5, "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+SGML elements for HTML export. For general information about cross-reference formats, variable definitions, generated lists and indexes, and HTML export, see the *FrameMaker User Guide*.

Cross-reference formats

Use the following building blocks to create cross-reference formats that refer to FrameMaker+SGML 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

Building block	Meaning
<$elemparanumonly>	The autonumber counters of the source element's first paragraph (or of the paragraph containing the source element), including any characters between the counters
<$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+SGML 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+SGML 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+SGML. 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+SGML generates and formats lists and indexes, see the FrameMaker user's manual.

HTML mapping for export

The *FrameMaker User Guide* includes instructions for converting a FrameMaker document to *HTML*. Converting FrameMaker+SGML 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 77

Mapping FrameMaker+SGML elements to HTML elements

You specify how to map FrameMaker+SGML 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+SGML 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+SGML elements and attributes in the column titled FrameMaker+SGML 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+SGML elements to HTML headings in the Mapping table. The hierarchy is determined by the context of the FrameMaker+SGML element, just as context can determine the formatting of a FrameMaker+SGML 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+SGML list element, and then you map FrameMaker+SGML 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+SGML 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+SGML 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)

4 Working with Special Files

This chapter discusses the special files you need to create an SGML 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 SGML files

FrameMaker+SGML provides a location for putting all of the special files associated with SGML applications. Some files are provided with the software. Others are files you create and store at this location.

If you use FrameMaker+SGML on a UNIX platform, when the software needs one of these files it looks in several places. It searches directories in this order:

- `curdir/fminit/uilanguage/sgml` where *`curdir`* is the directory from which you start FrameMaker+SGML and *`uilanguage`* indicates a particular user-interface language, such as `usenglish` or `ukenglish`
- `fminit/uilanguage/sgml` in the user's home directory
- `$FMHOME/fminit/uilanguage/sgml`, where `$FMHOME` is the directory in which FrameMaker+SGML is installed

If you use FrameMaker+SGML on a Macintosh or Windows platform, when the software needs one of these files it looks in a directory named `sgml` in the home directory for FrameMaker+SGML. On a Windows platform, you can specify an alternate sgml directory in the fmsgml.ini file. For information about the fmsgml.ini file, see the FrameMaker+SGML user's manual.

On all platforms, you can use the variable `$SGMLDIR` to refer to the `sgml` directory. The rest of this chapter follows this convention.

When you install FrameMaker+SGML, the `sgml` directory contains the following files and subdirectories:

`default.rw`	The default SGML read/write rules file. FrameMaker+SGML 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 Developer Tools>New Read/Write Rules File command on the File menu.
`docbook/`	The DocBook starter kit files. For information on this directory, see the online manual *Using the DocBook Starter Kit*.

`entfmts`	Default formats used for ISO public entities. For information on this file, see Appendix F, “ISO Public Entities.”
`isoents/`	Other ISO public entity files. For information on the contents of this directory, see Appendix F, “ISO Public Entities.”
`sgmlapps.fm`	The default version of the application definition file. You edit this file so that it contains the definitions of any applications you create.

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

Application definition file

You need to deliver an `sgmlapps.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 `sgmlapps.fm` file.

The `sgmlapps.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.

> **Important:** Reinstalling FrameMaker+SGML can overwrite the current `sgmlapps.fm` file with the default version that ships with the product. Be sure your end users understand this. To keep your site-specific SGML applications file intact, they should make a copy of this file before reinstalling, and then use the copy to overwrite the fresh version of `sgmlapps.fm`.

FrameMaker+SGML can associate a particular document with an SGML application in several ways:

- An EDD can explicitly associate itself with an SGML application. If it does, all FrameMaker+SGML documents that use that EDD are automatically associated with the application.
- An SGML application can name one or more SGML document elements. If it does, all SGML documents that use one of those document elements are automatically associated with the application.
- If neither of the above occurs, FrameMaker+SGML asks the end user to pick an SGML application to use with a document.

Editing an application definition file

The `sgmlapps.fm` file is a structured FrameMaker+SGML document. You edit this file by choosing Developer Tools>Edit SGML Application File from the File menu and using standard FrameMaker+SGML editing techniques. FrameMaker+SGML reads this 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 Developer Tools>Reread SGML Application File command.

The Reread SGML Application command rereads the current sgmlapps.fm file that is open and has current focus. Its purpose is to update or replace the current list of SGML applications stored in memory. This is true for all platforms. When this command is issued, the previous list of SGML applications stored is replaced or updated with the application list of the sgmlapps.fm file the command was issued on.

On UNIX, when you edit the `sgmlapps.fm` file, you may choose to write it to a directory other than the one in which FrameMaker+SGML found it. When you then use the Reread SGML Application File command, the software again looks in the three directories mentioned in "Location of SGML files" on page 41.

For example, assume you install the US English version of FrameMaker+SGML. The installation process places the default `sgmlapps.fm` in the directory `$FMHOME/fminit/usenglish/sgml`. Assume you do not yet have an `sgmlapps.fm` file in any other directory and your home directory is `/usr/vpg`. If you use the Edit SGML Application File command, it opens the file `$FMHOME/fminit/`*`uilanguage`*`/sgml`. You can then save it to `/usr/vpg/fminit/usenglish/sgml/sgmlapps.fm`. If you subsequently use the Reread SGML Application File command, FrameMaker+SGML rereads the current `sgmlapps.fm` file you have open. If the `sgmlapps.fm` file is from your home directory, then the application file in your home directory is read, not the one in FrameMaker+SGML's installation directory.

Contents of sgmlapps.fm

The highest-level element in an `sgmlapps.fm` file is `SGMLSetup`. That element's first child must be `Version`, to indicate the product version. The `Version` element is followed by zero or more `SGMLApplication` elements, each of which defines the pieces of an SGML application. Finally, there can be an optional `Defaults` element, which specifies information used unless overridden for a particular application.

The following table lists all elements allowed in `sgmlapps.fm` and identifies the sections that discuss each of those elements.

Element	Discussed in
`ApplicationName`	"Defining an application" on page 44
`CharacterEncoding`	"Specifying the character encoding for SGML files" on page 56
`Defaults`	"Providing default information" on page 46
`DOCTYPE`	"Specifying a document element" on page 47
`DTD`	"Specifying a DTD" on page 48
`Entities`	"Specifying entities" on page 49
`Entity`	"Specifying the location of individual entities" on page 51

Element	Discussed in
EntityCatalogFile	“Specifying entities through an entity catalog” on page 49
EntityName	“Specifying the location of individual entities” on page 51
EntitySearchPaths	“Specifying a search path for external entity files” on page 53
FileName	“Specifying the location of individual entities” on page 51
	“Specifying public identifiers” on page 53
FileNamePattern	“Specifying names for external entity files” on page 51
FrameDefaultAPIClient	“Specifying an SGML API client” on page 56
MaxErrorMessages	“Limiting the length of a log file” on page 57
Path	“Specifying a search path for external entity files” on page 53
	“Specifying a search path for including files in rules documents” on page 55
Public	“Specifying public identifiers” on page 53
PublicId	“Specifying public identifiers” on page 53
ReadWriteRules	“Specifying a read/write rules document” on page 47
RulesSearchPaths	“Specifying a search path for including files in rules documents” on page 55
SGMLApplication	“Defining an application” on page 44
SGMLDeclaration	“Specifying an SGML declaration” on page 48
Template	“Specifying a FrameMaker+SGML template” on page 48
UseAPIClient	“Specifying an SGML API client” on page 56
UseDefaultAPIClient	“Specifying an SGML API client” on page 56
Graphic	“How FrameMaker+SGML searches filename patterns” on page 52
Mapping	“Specifying entities through an entity catalog” on page 49
Notation	“Specifying names for external entity files” on page 51
FmTranslator	“Specifying an SGML API client” on page 56
XMLCharacterEncoding	“XML character encoding” on page 503
XMLWriteRules	“XML read/write rules” on page 504

Defining an application

FrameMaker+SGML collects all information pertaining to conversion of a related set of documents into an SGMLApplication element. This element has one required child element and several optional child elements.

The first child of a parent `SGMLApplication` element must be `ApplicationName` and gives the name of the application. It looks like:

Application Name: *name*

where *name* is a name to identify your application in the Set SGML Application and Use SGML Application dialog boxes. You cannot use the same name for multiple SGML applications.

If present, the optional child elements can occur in any order and can include the following elements, discussed in the named sections:

Element	Discussed in
`DOCTYPE`	"Specifying a document element" on page 47
`ReadWriteRules`	"Specifying a read/write rules document" on page 47
`DTD`	"Specifying a DTD" on page 48
`Template`	"Specifying a FrameMaker+SGML template" on page 48
`SGMLDeclaration`	"Specifying an SGML declaration" on page 48
`Entities`	"Specifying entities" on page 49
`RulesSearchPaths`	"Specifying a search path for including files in rules documents" on page 55
`UseAPIClient`	"Specifying an SGML API client" on page 56
`UseDefaultAPIClient`	"Specifying an SGML API client" on page 56
`CharacterEncoding`	"Specifying the character encoding for SGML files" on page 56
`XMLCharacterEncoding`	"XML character encoding" on page 503
`XMLWriteRules`	"XML read/write rules" on page 504

Several of these elements provide pathnames. If a relative pathname is given, the software looks for the file in several places (entities and read/write rules files only; hence `RulesSearchPaths` and `EntitySearchPaths` elements):

- The directory containing the file being processed. For example, if you're opening a DTD, the software first searches the directory in which it found the DTD only. If the pathname is absolute, it looks there. If it can't find it via the specified path, the SGML log reports an error and the operation is aborted.
- `$SGMLDIR` (for information on what directory this is, see "Location of SGML files" on page 41)
- The directory from which you started FrameMaker+SGML

The following is a typical application; note that these pathnames are for Unix systems:

Application name: ReportNumeric
DOCTYPE: Report
Read/write rules: reports/rules/repnumer.rul
DTD: reports/repnumer.dtd
Template: reports/repnumer
Rules search paths
1: $SGMLDIR/reports/rules
Use default API client.

If an application definition includes any of these elements, the value in the application definition overrides any value for that element in the `Defaults` element. The sections following the next section describe these elements in detail.

Providing default information

Some of the information you provide for individual applications may be common to all your applications. For such information you can specify defaults that are used whenever an application doesn't provide its own version of the information. You use the `Defaults` element to provide such information.

If present, the optional child elements of `Defaults` can occur in any order (with the exception of the `Graphics` element, it must be the last child) and can include the following elements, discussed in the named sections:

Element	Discussed in
`ReadWriteRules`	"Specifying a read/write rules document" on page 47
`DTD`	"Specifying a DTD" on page 48
`Template`	"Specifying a FrameMaker+SGML template" on page 48
`SGMLDeclaration`	"Specifying an SGML declaration" on page 48
`Entities`	"Specifying entities" on page 49
`RulesSearchPaths`	"Specifying a search path for including files in rules documents" on page 55
`FrameDefaultAPIClient`	"Specifying an SGML API client" on page 56
`UseAPIClient`	"Specifying an SGML API client" on page 56
`MaxErrorMessages`	"Limiting the length of a log file" on page 57
`CharacterEncoding`	"Specifying the character encoding for SGML files" on page 56
`Graphics`	"How FrameMaker+SGML searches filename patterns" on page 52
`XMLCharacterEncoding`	"XML character encoding" on page 503
`XMLWriteRules`	"XML read/write rules" on page 504

Specifying a document element

The `doctype` element specifies the generic identifier of the document element in SGML documents used with this application. If you open an SGML document with the matching document element specified in the DOCTYPE declaration, FrameMaker+SGML uses this application when translating the document. The element looks like:

DOCTYPE: *`doctype`*

where *`doctype`* identifies a document element.

For example,

DOCTYPE: chapter

matches an SGML document with the following declaration:

```
<!DOCTYPE chapter ...>
```

If more than one application defined in the `sgmlapps.fm` file specifies the same document element, and the end user opens a file with that document element, the software gives the user a choice of which of these applications to use. If the user opens an SGML document for which no application specifies its document element, the software gives the user the choice of all defined applications.

You can use more than one `DOCTYPE` element for an application, if that application is applicable to multiple document elements. For example, if the `Book` application applies when the document element is either `chapter` or `appendix`, you can use this definition:

Application name: Book
  DOCTYPE: chapter
  DOCTYPE: appendix
  . . .

The `DOCTYPE` element can be a child of a parent `SGMLApplication` element only.

Specifying a read/write rules document

The `ReadWriteRules` element specifies the SGML read/write rules document associated with the application. It looks like:

ReadWriteRules: *`rules`*

where *`rules`* is the pathname of a FrameMaker+SGML read/write rules document.

You can have only one `ReadWriteRules` element for each application. It can be a child of both parent `SGMLApplication` and parent `Defaults` elements.

Specifying a DTD

The `DTD` element specifies a file containing the external DTD subset the software uses when importing and exporting an SGML document. It looks like:

DTD: *`dtd`*

where *`dtd`* is the pathname of a file containing a document type declaration subset.

Note that the file you specify with the `DTD` element must be an external DTD subset. It cannot be a complete DTD. That is, the file cannot have the form:

```
<!DOCTYPE book [
   <!element book . . .>
   . . .
]>
```

Instead, it should simply have the form:

```
<!element book . . .>
. . .
```

For more information on external DTD subsets, see "SGML DTDs" on page 9.

You can have only one `DTD` element for each application. It can be a child of both parent `SGMLApplication` and parent `Defaults` elements.

Specifying a FrameMaker+SGML template

The `Template` element specifies the location of the FrameMaker+SGML template. It looks like:

Template: *`template`*

where *`template`* is the pathname of a FrameMaker+SGML template.

The software uses this template to create new FrameMaker+SGML documents from SGML documents, which may be single documents resulting from the `Open` or `Import` command or documents in a book created through the `Open` command.

If this element is not present, the software creates new portrait documents as needed. When you import an SGML document into an existing document, the software uses the import template only for reference elements.

You can have only one `Template` element for each application. It can be a child of both parent `SGMLApplication` and parent `Defaults` elements.

Specifying an SGML declaration

The `SGMLDeclaration` element specifies the location of a file containing a valid SGML declaration. It looks like:

SGMLDeclaration: *`declaration`*

where *declaration* is the pathname of the SGML declaration file.

You can have only one SGMLDeclaration element for each application. It can be a child of both parent SGMLApplication and parent Defaults elements.

Specifying entities

To specify the location of various entities, you use the Entities element. The possible child elements of a parent Entities element are:

Element	Discussed in
EntityCatalogFile	"Specifying entities through an entity catalog" on page 49
Entity	"Specifying the location of individual entities" on page 51
FileNamePattern	"Specifying names for external entity files" on page 51
Public	"Specifying public identifiers" on page 53
EntitySearchPaths	"Specifying a search path for external entity files" on page 53

If you use the EntityCatalogFile element, you cannot use any of the elements Entity, FilenamePattern, or Public.

You can have only one Entities element for each application, although that Entities element can have more than one of some of its child elements. The Entities element can be a child of both parent SGMLApplication and parent Defaults elements.

Specifying entities through an entity catalog

The EntityCatalogFile element specifies a file containing mappings of an entity's public identifier or entity name to a filename. It looks like:

Entity locations
Entity catalog file: *fname*

where *fname* is the filename of the entity catalog. Entity catalogs and their specified format are described later.

You can specify multiple EntityCatalogFile elements in a single Entities element. If you use this element, you cannot use any of the Entity, FilenamePattern, or Public elements.

You can use the EntityCatalogFile element both in the Entities element of the Defaults element and in an SGMLApplication element to specify information for a particular application. When searching for an external entity, FrameMaker+SGML searches the application's entity catalogs before searching those in default EntityCatalogFile elements.

If you have an EntityCatalogFile element in an application definition, the software ignores Entity, FilenamePattern, and Public elements in the Defaults element.

Why use entity catalogs

Technical Resolution 9401:1994 published by SGML Open relates to entity management issues affecting how SGML documents work with each other:

- Interpreting external identifiers in entity declarations so that an SGML document can be processed by different tools on a single computer system
- Moving SGML documents to different computers in a way that preserves the association of external identifiers in entity declarations with the correct files or other storage objects

The technical resolution uses *entity catalogs* and an interchange packaging scheme to address these issues. FrameMaker+SGML supports such entity catalogs with the `EntityCatalogFile` element.

Entity catalog format

Each entry in the entity catalog file associates a filename with information about an external entity that appears in an SGML document. For example, the following are catalog entries that associate a public identifier with a filename:

```
PUBLIC "ISO 8879-1986//ENTITIES Added Latin 1//EN" "isolat1.ent"
PUBLIC "-//USA/AAP//DTD BK-1//EN" "aapbook.dtd"
```

In addition to entries mapping public identifiers to filenames, an entry can associate an entity name with a filename:

```
ENTITY "chips" "graphics\chips.tif"
```

A single catalog can contain both types of entry.

If the specified filename in a catalog entry is a relative pathname, the path is relative to the location of the catalog entry file.

For a complete description of the syntax of a catalog entry, see *Technical Resolution 9401:1994 Entity Management* published by SGML Open.

How FrameMaker+SGML searches entity catalogs

There may be multiple catalog files for a single application. When trying to locate a particular external entity, FrameMaker+SGML searches the files one at a time until it finds the entry it is looking for. In each file, the software first searches for an entity using the external entity's public identifier. If the software finds the identifier, it uses the associated filename to locate the entity. If it does not find the public identifier, the software searches the file looking for the entity name. If it does not find the entity name either, the software continues searching in the next catalog file.

In some circumstances, a system identifier specified in an external entity declaration may not be valid. If so, FrameMaker+SGML uses public identifier and entity name mappings.

Specifying the location of individual entities

Instead of using an entity catalog to associate entities with files, you can use the `Entity` element as a child of a parent `Entities` element. This element allows you to directly associate a filename with an individual entity. It looks like:

Entity locations
Entity name: *ename*
Filename: *fname*

where *ename* is the name of an entity and *fname* is a filename.

You can specify multiple child `Entity` elements for a single `Entities` element. You use the `FilenamePattern` and `EntitySearchPaths` elements to help the software find these files.

The `Entity` element can be a child of a parent `Entities` element in the `Defaults` element to set default entity information and of a parent `SGMLApplication` element to specify information for a particular application. When searching for an external entity, the software searches the application's entity locations before searching those in the default `Entity` elements.

Specifying names for external entity files

One or more `FilenamePattern` elements can appear as a child of a parent `Entities` element to tell the software how to locate an external entity.

A `FilenamePattern` element does not apply to an entity for which there is an `Entity` element. Otherwise, it applies to all external entities except those with an external identifier that includes a public identifier but no system identifier. The `FilenamePattern` looks like:

Entity locations:
FilenamePattern: *pattern*

where *pattern* is a string representing a device-dependent filename. The three variables that can appear within *pattern* are interpreted as follows:

Variable	Interpretation
`$(System)`	The system identifier from the entity declaration
`$(Notation)`	The notation name from the entity declaration of an external data entity
`$(Entity)`	The entity name

Case is not significant in variable names, although it may be significant in the values of the variables. If a variable is undefined in a particular context, that variable evaluates to the empty string.

There can be multiple child `FilenamePattern` elements in a parent `Entities` element. The software assumes the last pattern in the `Entities` element is:

FilenamePattern: $(System)

Thus, if no `FilenamePattern` elements appear or even if no `Entities` element appears, the software assumes system identifiers are complete pathnames and will check search paths to locate the file.

How FrameMaker+SGML searches filename patterns

When locating an external entity, FrameMaker+SGML tests the value of the *`pattern`* arguments in successive `FilenamePattern` elements that have the same parent `Entities` element, in the order they occur, until it finds the name of an existing file. As it tests each *`pattern`*, it substitutes relevant information from the entity's declaration for variables in *`pattern`*.

You can use the `FilenamePattern` element both in the `Entities` element of the `Defaults` element and in an `SGMLApplication` element to specify information for a particular application. When searching for an external entity, FrameMaker+SGML tests all the filename patterns specified for the application before it tests those in default `FilenamePattern` elements.

Example

Suppose the `Entities` element looks like:

Entity locations:
FilenamePattern: $(System).sgm
FilenamePattern: $(System).$(Notation)

and the SGML document contains:

```
<!ENTITY intro SYSTEM "introduction">
<!ENTITY chips SYSTEM "chipsfile" NDATA cgm>
. . .
&intro;
. . .
<graphic entity=chips>
```

When processing the reference to `intro`, the software searches for a file called `introduction.sgm`. If that file is not found, it searches for a file called `introduction.` (this filename has a trailing period) because there is no notation name in the entity declaration. It is an error if neither of these files exists.

When processing the `entity` attribute of the `graphic` element, FrameMaker+SGML searches for a file named `chipsfile.sgm`. If one is not found, it then looks for

`chipsfile.CGM`, assuming that the `NAMECASE GENERAL` parameter of the associated SGML declaration is `NAMECASE GENERAL YES`.

> **Important:** The `NAMECASE GENERAL` parameter of the SGML declaration determines the case-sensitivity of notation names. The value of this parameter in the reference concrete syntax is `NAMECASE GENERAL YES`. With this declaration, the SGML parser forces notation names to uppercase. Because of this, and because the UNIX file system is case-sensitive, on a UNIX system, if you have the default SGML declaration and you use the `$(notation)` variable with the filename element, that portion of the name must be uppercase.

Specifying public identifiers

The `Public` element of an `Entities` element tells the software how to process an external identifier that has a public identifier but no system identifier. It looks like:

Entity locations:
Public ID: *pid*
Filename: *fname*

where *pid* is a public identifier and *fname* is the name of a file to be associated with the entity using the public identifier.

You can give multiple `Public` elements in the same parent `Entities` element. If you want to give multiple filenames to search for a particular public identifier, you can specify the same public identifier in multiple `Public` elements.

You can use the `Public` element both in the `Entities` element of the `Defaults` element and in an `Entities` element of an `SGMLApplication` element to specify information for a particular application. If a `Public` element occurs as a child of an `SGMLApplication` element, that identifier is used in preference to one occurring as a child of the `Defaults` element.

Specifying a search path for external entity files

The `EntitySearchPaths` child of a parent `Entities` element tells the software what directories to search for the files indicated by `Entity`, `FilenamePattern`, and `Public` elements. It looks like:

Entity locations:
EntitySearchPaths
1: $directory_1$
. . .
N: $directory_n$

where each *directory*$_i$ is a device-dependent directory name. The three variables and their abbreviations that can be used to specify a directory are as follows:

Variable	Abbreviation	Interpretation
`$HOME`	~	The user's home directory
`$SRCDIR`	.	The directory containing the SGML document entity being processed
`$SGMLDIR`		The `sgml` directory in use (for information on what directory this is, see "Location of SGML files" on page 41)

Each *directory*$_i$ value can be an absolute pathname or relative to `$SRCDIR`.

How FrameMaker+SGML searches for entity files

To locate an external entity, FrameMaker+SGML searches the specified directories in the order listed. You can use the `EntitySearchPaths` element both in the `Entities` element of the `Defaults` element and in an `SGMLApplication` element. When searching for an external entity, FrameMaker+SGML searches the directories named in the `EntitySearchPaths` element for the application before it searches those in a default `EntitySearchPaths` element.

An `Entities` element can contain only one `EntitySearchPaths` element. The software assumes the `EntitySearchPaths` element ends this way:

EntitySearchPaths
. . .
N: $SRCDIR

Thus, if there is no `EntitySearchPaths` element, the software assumes all SGML files are in the same directory.

Example

Assume the `Defaults` element is defined as follows:

Defaults
Entity locations:
FilenamePattern: $(System).sgm
FilenamePattern: $(System).$(Notation)
EntitySearchPaths
1: $HOME
2: $SRCDIR

and the SGML document contains:

```
<!ENTITY intro SYSTEM "introduction">
<!ENTITY chips SYSTEM "chipsfile" NDATA cgm>
. . .
&intro;
. . .
<graphic entity=chips>
```

When processing the reference to `intro`, the software looks for the files:

```
$HOME/introduction.sgm
$SRCDIR/introduction.sgm
$HOME/introduction.
$SRCDIR/introduction.
```

until it finds one of those files. When processing the `graphic` element, the software searches in order for:

```
$HOME/chipsfile.sgm
$SRCDIR/chipsfile.sgm
$HOME/chipsfile.CGM
$SRCDIR/chipsfile.CGM
```

Specifying a search path for including files in rules documents

The `RulesSearchPaths` element is analogous to the `EntitySearchPaths` element, but it pertains to additional files you include in an SGML read/write rules document rather than to external entities referenced within an SGML document. Its `Path` child elements indicate individual directories. It looks like:

RulesSearchPaths:
1: $directory_1$
. . .
N: $directory_n$

where each $directory_i$ is a device-dependent directory name. The three variables and their abbreviations that can be used to specify a directory are as follows:

Variable	Abbreviation	Interpretation
`$HOME`	~	The user's home directory
`$RULESDIR`	.	The directory containing the current SGML read/write rules document (for Macintosh and Unix, only)
`$SGMLDIR`		The `sgml` directory in use (for information on what directory this is, see "Location of SGML files" on page 41)

Each $directory_i$ value can be an absolute pathname or relative to `$RULESDIR`.

How FrameMaker+SGML searches for rules files

Only one `RulesSearchPaths` element can occur as the child of a single parent `SGMLApplication` element or parent `Defaults` element. When searching for a file you include in an SGML read/write rules document, FrameMaker+SGML searches the directories named in the `RulesSearchPaths` element for the application before it searches those in the `RulesSearchPaths` element of the `Defaults` element.

The software assumes `RulesSearchPaths` ends in this way:

RulesSearchpaths:

. . .

N: $RULESDIR

Thus, if there is no `RulesSearchPaths` element, the software assumes all files you include in the SGML read/write rules document are in the same directory. This applies to the `XMLWriteRules` element, as well as the `ReadWriteRules` element.

Specifying an SGML API client

In an application definition, the `UseDefaultAPIClient` element tells the software that your application does not use a special client for SGML translation. In the defaults section, the `FrameDefaultAPIClient` element serves the same purpose. The default client is named `FmTranslator`.

If you do need an SGML API client, use the `UseAPIClient` element in either context. For information on creating SGML API clients for an SGML application, see the online manual *SGML API Programmer's Guide*.

Specifying the character encoding for SGML files

The `CharacterEncoding` element tells the software which encoding to use for the SGML text. Typically, this is only important on non-Western systems, or in SGML applications that encounter SGML files using double-byte text. It can contain one of the following child elements: `ISOLatin1`, `ASCII`, `ANSI`, `MacASCII`, `ShiftJIS`, `KSC8EUC`, `GB8EUC`, `CNSEUC`, `Big5`, `JIS8EUC`. The `CharacterEncoding` element looks like this:

CharacterEncoding:
ShiftJIS

On a non-Western system, the text for an SGML file can contain double-byte text. This text can be in any one of a number of different text encodings. For example, Japanese text on a Unix system is typically in EUC, while on the Macintosh it is typically in Shift-JIS.

FrameMaker+SGML can interpret SGML files that contain double-byte text in *#PCDATA*, *RCDATA*, and *CDATA*. The software expects all other text to be within the 7-bit ASCII range (which is supported by all Asian fonts). This means that document content can be in double-byte encodings, but the markup must be in the ASCII range. Typically, for example, the only text in a DTD that will contain double-byte characters would be text used to specify attribute values.

To import and export SGML that contains double-byte text, you should specify the character encoding to use, either as a default for all applications, or for a specific SGML application. For a given SGML application, there can only be one encoding. If you don't specify an encoding for your application, FrameMaker+SGML determines the encoding to use by considering the current default user interface language and the current operating system; for the current language, it uses the operating system's default encoding. The default encodings for the supported operating systems are:

	Windows 95/NT	**Macintosh**	**Unix**
Roman languages	ANSI	Macintosh ASCII	ISOLatin-1
Japanese	Shift-JIS	Shift-JIS	JIS8 EUC
Simplified Chinese	GB8 EUC	GB8 EUC	GB8 EUC
Traditional Chinese	Big5	Big5	CNS EUC
Korean	KSC8 EUC	KSC8 EUC	KSC8 EUC

It is possible to have an Asian language for the user interface, but the content of the document files is in Roman fonts. In this case, any exported Roman text that falls outside of the ASCII range will be garbled. For this reason, we recommend that you specify an encoding for any application that might be used on a non-Western system.

The template for your application must use fonts that support the language implied by the encoding you specify. Otherwise, the text will appear garbled when imported into the template. You can fix this problem after the fact by specifying different fonts to use in the resulting files.

Limiting the length of a log file

The `MaxErrorMessages` child element of the `Defaults` element allows you to limit the length of SGML error reports. It looks like:

MaxErrorMessages: *n*

where *n* is the desired limit. If *n* is less than 10, the software resets it to 10. This must be the last child of the parent `Defaults` element.

By default, FrameMaker+SGML does not write more than 150 messages (error messages and warnings) to a single log file.

Messages pertaining to opening and closing book components are not included in this limit. Messages generated through your own SGML API client are also not counted, although if you wish, you can count them using your own code.

In documents that generate large numbers of messages, the 151st message is replaced with a note that additional messages have been suppressed.

Note that processing continues, even though further messages are not reported. This message limit is reset for every file processed and for each component of a book.

Log files

FrameMaker+SGML log files give you information used to identify and correct errors in your application.

Generating log files

FrameMaker+SGML 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. The one exception to this is that the UNIX batch utilities always produce a log file so that you know what files were processed.

FrameMaker+SGML generates messages for conditions such as:

- SGML syntax errors
- SGML read/write rule syntax errors
- Missing or otherwise unavailable files
- Missing entities
- Inconsistencies between SGML read/write rules and the relevant EDD or DTD

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 an SGML 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+SGML updates an existing EDD, it reports an error for a FrameMaker+SGML 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+SGML encounters a syntax error in a rules document, it stops processing.

To aid in debugging your SGML application, FrameMaker+SGML provides a facility for locating certain problems. Choose Check SGML Read/Write Rules from the Developer Tools menu. FrameMaker+SGML 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+SGML 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+SGML document, such as a rules document or a FrameMaker+SGML 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+SGML 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+SGML limits the number of messages it sends to the log file for a given document. For information on how to change that number, see "Limiting the length of a log file" on page 57.

Other special files

There are several other file types you must work with in creating an SGML application. These files are discussed in other chapters. For information on creating an EDD, see Part II, "Working with an EDD." For information on creating SGML read/write rules files, see Chapter 11, "SGML Read/Write Rules and Their Syntax." For information on ISO public entity set files, see Appendix F, "ISO Public Entities."

Part II *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 SGML application, you should be familiar with Part I, "Developing a FrameMaker+SGML application," before using the material in this part.

The chapters in this part are:

- Chapter 5, "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. The chapter also has a list of all elements in an EDDs Element Catalog, with links to sections where the elements are covered in the syntax chapters.

- Chapter 6, "Structure Rules for Containers, Tables, and Footnotes"
  Chapter 8, "Attribute Definitions"
  Chapter 7, "Text Format Rules for Containers, Tables, and Footnotes"
  Chapter 9, "Object Format Rules"

  The four syntax chapters describe structure rules, attribute definitions, text format rules, and object format rules, and show examples of these constructs in element definitions.

5 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+SGML 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 exporting from, then importing into a new document.

You can start a new EDD in two ways. If you have an SGML *document type definition* (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 SGML, you can create an EDD and write its definitions entirely in FrameMaker+SGML.

An EDD is a regular structured FrameMaker+SGML 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 *FrameMaker+SGML User Guide*.

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. In the outline below, click a topic to go to its page.

A first look at the steps for developing an EDD:

- "Overview of the development process" on page 64

How to start or update an EDD:

- "Creating or updating an EDD from a DTD" on page 65
- "Starting an EDD without using a DTD" on page 68

Summary of the elements you work with in an EDD:

- "The Element Catalog in an EDD" on page 69

How to define elements and supply other information about them:

- "Defining preliminary settings in an EDD" on page 77
- "Organizing and commenting an EDD" on page 78
- "Writing element definitions" on page 79
- "Keyboard shortcuts for working in an EDD" on page 89

What to do when you're finished developing an EDD:

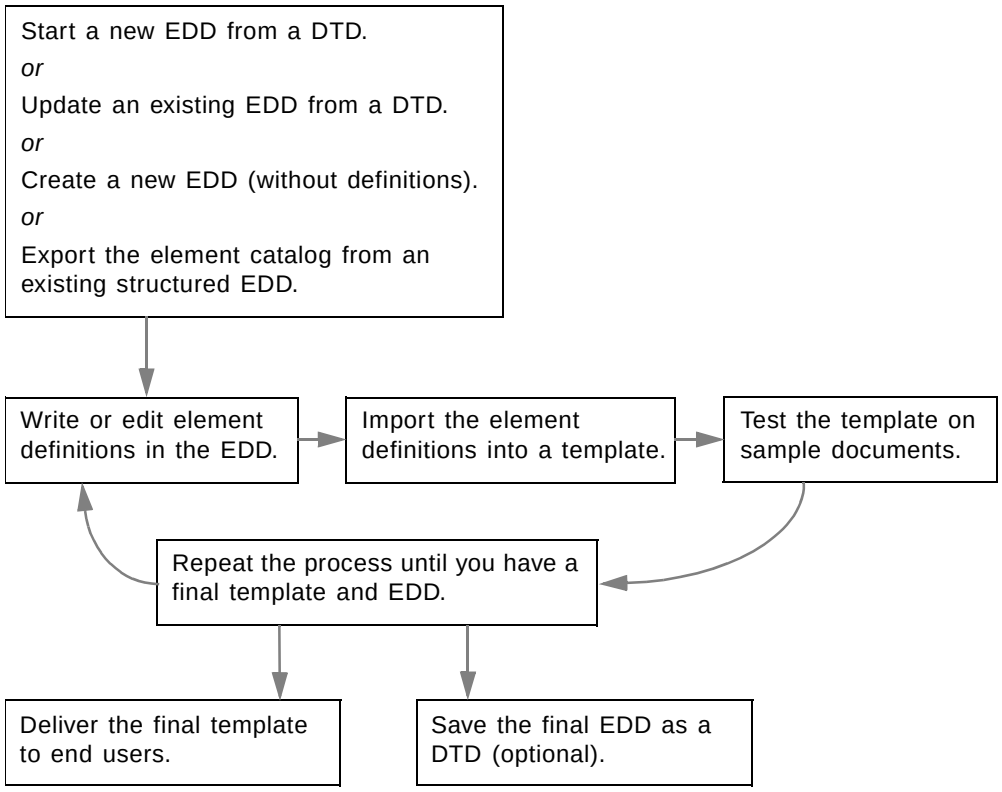
- "Creating an Element Catalog in a template" on page 90
- "Saving an EDD as a DTD for export" on page 92

Where to find structured documents and EDDs to review:

- "Sample documents and EDDs" on page 95

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 an SGML DTD, FrameMaker+SGML 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+SGML document to create a new EDD or create a new empty EDD by selecting File>Developer Tools>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+SGML template that also stores formats which 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 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 SGML application. If you didn't start from a DTD but your users will export documents to SGML, 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 an SGML DTD, you can use the DTD as a starting point for your EDD. FrameMaker+SGML 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

The prologue of every SGML document includes a DTD that provides element and attribute definition list declarations. The SGML 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 an SGML 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 "SGML DTDs" on page 9.

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+SGML 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 67), test the results on sample SGML 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 26.

You develop read/write rules in a special rules document that is part of an SGML 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 11, "SGML Read/Write Rules and Their Syntax."

An application definition file (such as `sgmlapps.fm`) describes what files are used in each SGML application you deliver to an end user. If necessary, update an application definition in the application definition 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 42.

Creating an EDD from a DTD

To create an EDD from a DTD, choose Open DTD from the File>Developer Tools submenu in any open FrameMaker+SGML 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 SGML application you select is stored in an `SGMLApplication` element in the EDD for future updates and exports. All FrameMaker+SGML 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 SGML application with no rules. To specify an SGML application for the EDD later, update the EDD from the DTD (see "Updating an EDD from a DTD" on page 67), and this time select the application, or insert and fill in an `SGMLApplication` element in the EDD. For information on filling in the element, see "Setting an SGML application" on page 77.

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+SGML makes assumptions about how the constructs from the DTD should be represented. FrameMaker+SGML 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 SGML element declaration to a FrameMaker+SGML 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 SGML elements correctly. (An exception to this is *CALS* tables. If your DTD uses the CALS table model, FrameMaker+SGML does recognize those elements as table elements.) You can modify the default translation using read/write rules.

A complete DTD at the beginning of an SGML document may include a reference to an SGML declaration. If the DTD you're translating has this reference, FrameMaker+SGML uses the SGML declaration it refers to, rather than the declaration defined in the SGML application.

For details on the translation of each type of element, see Part III, "Translating between SGML and FrameMaker+SGML."

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>Developer Tools submenu in the EDD. Select the DTD in the Import DTD dialog box. If the Use Application dialog box appears, select an SGML application for the EDD. (Use Application appears only if no application is specified in the EDD.)

In the updated EDD, FrameMaker+SGML 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+SGML 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 SGML 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/sgml/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+SGML log files, see "Log files" on page 58.

Starting an EDD without using a DTD

If you do not have an SGML DTD, or if you do not plan to translate structured documents to SGML, 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+SGML 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 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+SGML release, and one empty `Element` element.

To create a new EDD, choose New EDD from the File>Developer Tools submenu.

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+SGML 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.

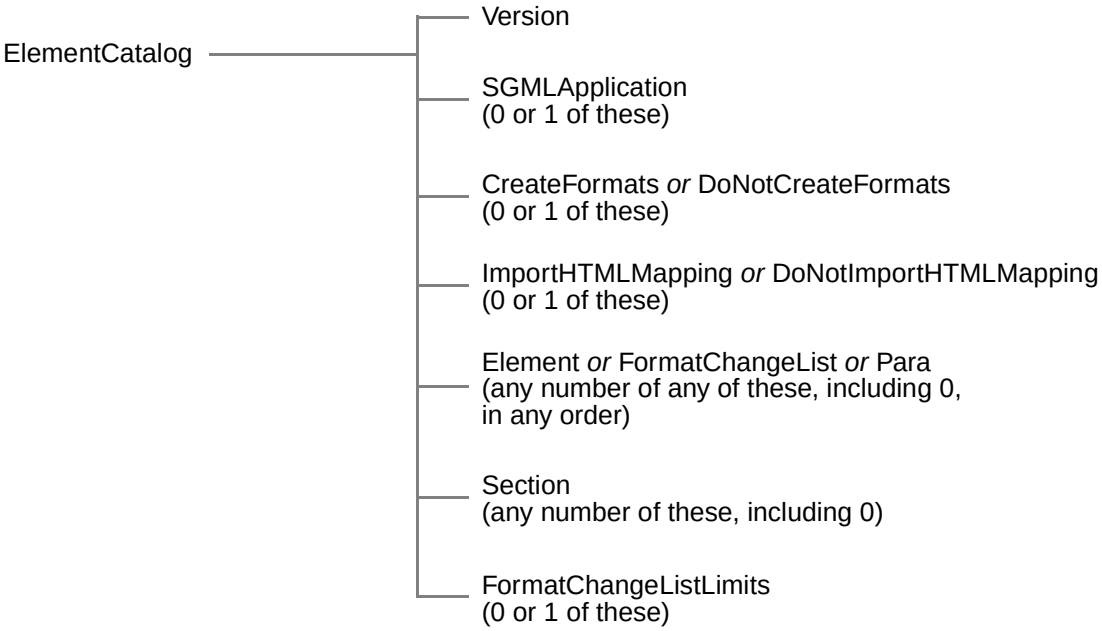
The exported EDD and the EDD from which it was created are equivalent in that they have the same element definitions. The two EDDs may differ, however, in the order and grouping of definitions. `Section`, `Head`, and `Para` elements from the original EDD are also not preserved in the exported EDD.

The Element Catalog in an EDD

An EDD is a regular structured FrameMaker+SGML 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+SGML release used when the EDD was created; the number is not editable. The following list describes what the optional child elements are for and where you can find more discussion of them in this manual:

- `SGMLApplication`: Specifies an SGML application for the EDD and for documents that use the EDD. You need to type the name of the application. See "Setting an SGML application" on page 77
- `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 77.
- `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 template or other document. See "Specifying whether to transfer HTML mapping tables" on page 77.
- `Element`: Begins the definition of an element. See "Writing element definitions" on page 79.
- `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 151.
- `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 78.
- `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 78.
- `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 152.

All elements in the catalog

This section lists all of the elements in an EDD's Element Catalog—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
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

Element tag	Purpose of the 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
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

Element tag	Purpose of the 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
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

Element tag	Purpose of the element
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
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

Element tag	Purpose of the element
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
SGMLApplication	Reference to an SGML application for 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
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
Subrule	Nested format rule
Suffix	Text string for a suffix

Element tag	Purpose of the element
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
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 `SGMLApplication`. FrameMaker+SGML 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+SGML 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+SGML 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+SGML also import the EDD mapping table. If you are importing into a book, FrameMaker+SGML 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 xxx.

Setting an SGML application

If you are working with SGML, you need to specify which SGML application to associate with the EDD. An SGML application defines information such as a DTD, an SGML declaration, a read/write rules document, an application definition file, entity catalogs, and a FrameMaker+SGML template (which has the elements from your EDD).

FrameMaker+SGML uses the application when you translate between a DTD and an EDD and when an end user shares documents between SGML and FrameMaker+SGML.

When you first convert an SGML DTD to an EDD using File>Developer Tools>Open DTD, you can select an SGML application for the EDD. The new EDD has an `SGMLApplication` 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 an SGML application in an EDD that does not have an application, insert an `SGMLApplication` 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 `SGMLApplication` element.

All documents that use the EDD are also associated with the SGML application. Users can change to a different application in an individual document by using the Set SGML Application command (File menu).

For information on the parts of an SGML application and the process of developing one, see Chapter 3, “Creating an SGML Application.”

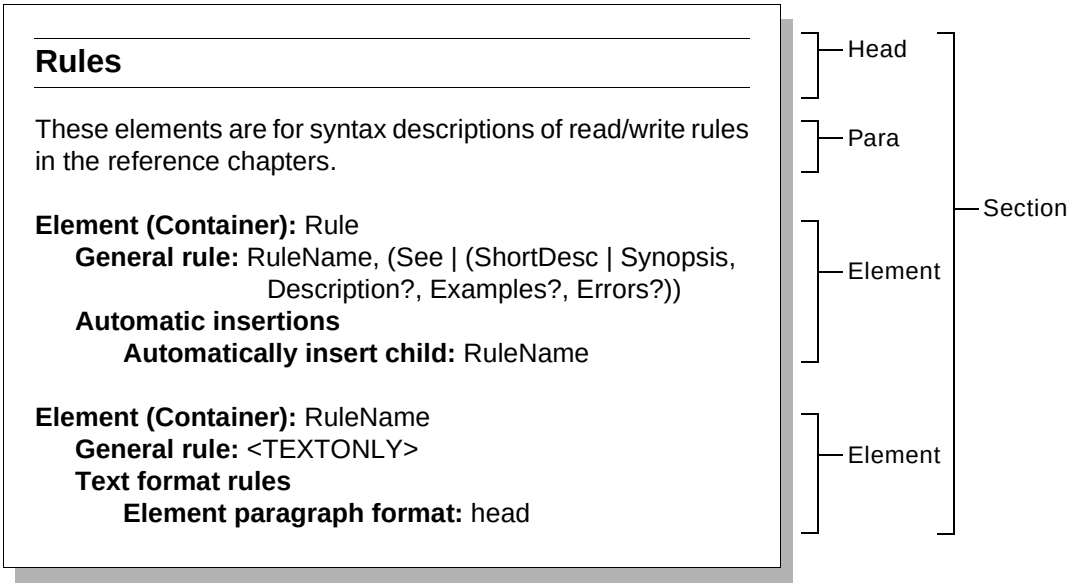
Organizing and commenting an EDD

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 SGML 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+SGML 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+SGML, 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).

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 209.

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 90.

- 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 89.

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:

```
Element (Container): Section                                   Element tag
    General rule: Head, Para+, Section*                        Content rules
    Inclusion: Index
    Automatic insertions                                       Structure rule
        Automatically insert child: Head                       for autoinsertion
    Attribute list
        1. Name: Author      String      Required              Attribute
        2. Name: Security    Choice      Optional              definitions
           Choices: Top Secret, Classified, Unclassified
    Text format rules                                          Text format rules
        Element paragraph format: body
        1. If context is: Section
               Basic properties
                   Move left indent by: +0.5"
                                                               Element type
Element (Footnote): Footnote
    General rule: (<TEXT> | Emphasis)+

RulesTable is for listing groups of read/write rules.          Comment
Element (Table): RulesTable
    General rule: Heading?, Body, Footing?
    Initial table format                                       Table object
        1. If context is: Item < List                          format rule
               Table format: IndentedTable
           Else
               Table format: StandardTable
    Text format rules
        Element paragraph format: cell

Element (TableBody): Body
    General rule: Row+
```

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 80.

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+SGML 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 99.

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 104.

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 104.

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 106.

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 108.

7.(Optional) Write attribute definitions to define attributes that can store additional information about instances of the element.

In FrameMaker+SGML, 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 8, “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 7, “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 178.

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.

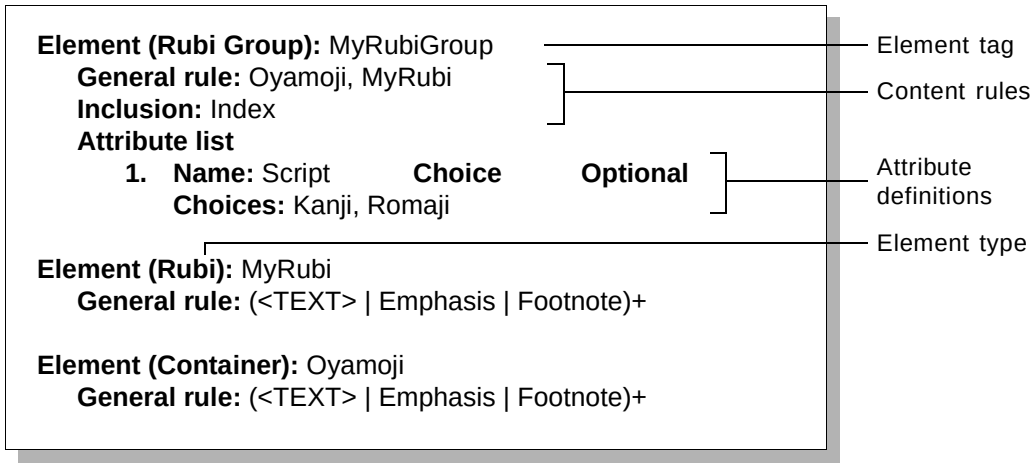
The Rubi element must be the last child element of the Rubi Group element. The Rubi Group and Rubi elements can contain other child elements. For example, the Oyamoji can be in a container of its own, or you can insert footnotes within Oyamoji or Rubi text. Note that the JIS rules for a Rubi group state 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+SGML 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 111.

Defining an object element

A FrameMaker+SGML 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 81.

An element definition for a FrameMaker+SGML 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:

```
Element (Marker): GlossaryTerm                                  Element tag
    Initial marker type
        1.  In all contexts.
                Use marker type: Glossary
                                                                Element type
Element (Graphic): Figure
    Initial graphic element format
        1.  If context is: Item < Procedure                     Object format rule
                Insert imported graphic file.
            Else
                Insert anchored frame.

Element (CrossReference): CrossRef
    Attributes list
        1.  Name: Reference   IDReference   Required            Attribute definition
    Initial cross-reference format
        1.  In all contexts.
                Use cross-reference format: Heading & Page

Date can be used in the title or in other front matter.         Comment
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)
```

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 80.

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+SGML, 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 8, "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

Element type	Property
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 9, “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+SGML 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

To	Press
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+SGML 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 an SGML application. To do this, insert a `TemplateForImport` element in the application definition in the `sgmlapps.fm` file and type the pathname of the template. When end users open an SGML document that uses that application, FrameMaker+SGML applies structure and formatting from the template to the document. For more information, see "Application definition file" on page 42.

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 *FrameMaker User Guide*.

Even after importing element definitions into a template, you will probably want to 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+SGML creates a new catalog for the template. If you import definitions into a template that already has a catalog, FrameMaker+SGML replaces the old catalog with your new one.

Log files for imported element definitions

If FrameMaker+SGML 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+SGML needs to create because they are referenced in the EDD but not yet stored in the template. If you do not want FrameMaker+SGML 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.) For general information on FrameMaker+SGML log files, see "Log files" on page 58.

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+SGML 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 112
- "Debugging text format rules" on page 154
- "Debugging object format rules" on page 185

Saving an EDD as a DTD for export

If your end users will be saving FrameMaker+SGML documents as SGML and you did not begin the development process with an SGML DTD, you need to save your EDD as a DTD so that users will have a DTD for exported documents. FrameMaker+SGML 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 an SGML application before your end users can save documents as SGML. To do this, insert a `DTDForExport` element in the

application definition in `sgmlapps.fm` and type the pathname of the DTD. For more information, see "Application definition file" on page 42.

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+SGML 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 26.

You develop read/write rules in a special rules document that is part of an SGML 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 11, "SGML 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 an SGML 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 SGML application with no rules. To specify an SGML 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+SGML makes assumptions about how the constructs from the EDD should be represented. FrameMaker+SGML 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+SGML element definition to an SGML element declaration of the same name, and it produces an attribute list declaration for each attribute defined for the element.

FrameMaker+SGML 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 SGML 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+SGML reflect this information.

For details on the translation of each type of element, see "Translating between SGML and FrameMaker+SGML" on page 187.

> **Important:** When exporting an EDD as a DTD, if the new DTD file has the same name as the old DTD file, FrameMaker+SGML 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. On the Macintosh, this option is turned off by default.

SGML declarations

FrameMaker+SGML runs a new DTD through an SGML parser. In the process, it may identify errors in the syntax of the DTD. Two of the most common errors are invalid SGML names and an inappropriate SGML declaration. You can use read/write rules to translate FrameMaker+SGML element tags to valid SGML names. And if the default SGML declaration that FrameMaker+SGML provides is not appropriate for your DTD, you can modify the declaration to avoid capacity and quantity errors.

The default SGML declaration for FrameMaker+SGML 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 `sgmlapps.fm` and type the pathname for the new declaration. For a description of the default SGML declaration and the variations that FrameMaker+SGML supports, see Appendix D, "SGML Declaration."

If you need to reapply the parser to a DTD but not recreate the DTD, select File>Developer Tools>Parse SGML Document.

Log files for a translated EDD

If FrameMaker+SGML 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+SGML 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.) For general information on log files, see "Log files" on page 58.

Sample documents and EDDs

FrameMaker+SGML 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.

- For UNIX:

  `$FMHOME/fminit/uilanguage/Samples/FMSGML/`
  `$FMHOME/fminit/uilanguage/Samples/Templates/Structured/`

- For Windows:

  `$FMHOME\samples\fmsgml\`
  `$FMHOME\samples\templates\Structured\`

- For Macintosh:

  `$FMHOME:Samples:FMSGML:`
  `$FMHOME:Samples:Templates:Structured:`

The `FMSGML` directory contains a commented EDD for this developer's guide and one structured document that uses the EDD. You can also unlock the documents in the developer's guide to examine their structure. (To unlock or relock a document, press Esc Flk.)

The `Structured` directory contains EDDs and structured documents for outlines, reports, and viewgraphs.

6 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+SGML. These rules translate to content models and declared content in SGML. If you convert an SGML 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. In the outline below, click a topic to go to its page.

Summary of required and optional structure rules:

- "Overview of EDD structure rules" on page 98

Syntax of content rules and the translation of content rules to SGML:

- "Writing an EDD general rule" on page 99
- "Specifying validity at the highest level in a flow" on page 104
- "Adding inclusions and exclusions" on page 104
- "How content rules translate to SGML" on page 106

Optional rules that specify structure properties for new instances of an element:

- "Inserting descendants automatically in containers" on page 106
- "Inserting table parts automatically in tables" on page 108
- "Inserting Rubi elements automatically in Rubi groups" on page 111

Information to help you correct errors in structure rules:

- "Debugging structure rules" on page 112

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*.

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.

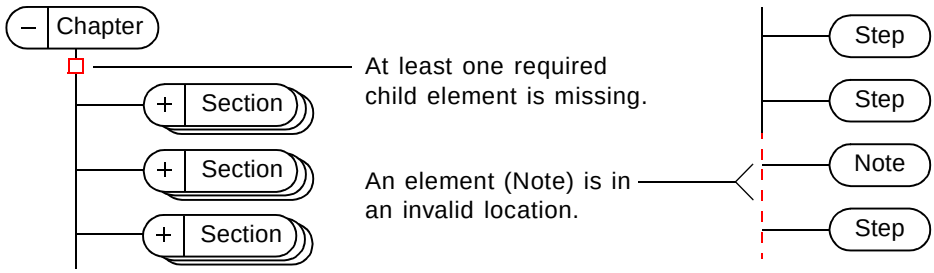
Element (Table): RuleTable
General rule: Title?, Heading, Body, Footing?
Initial structure pattern: Heading, Body

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+SGML 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 102.

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+SGML 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.

Symbol	Meaning
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+SGML 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+SGML 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 101.

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 SGML elements with declared content CDATA or RCDATA.

Use the <EMPTY> symbol for elements you want to remain empty. These are some possible examples of empty 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 SGML, see “How content rules translate to SGML” on page 106.

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+SGML inserts outer parentheses when SGML 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.) 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.

Element type	Restrictions
	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 178.

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+SGML displays an alert.

Default general rules for EDD elements

If you do not specify a general rule, FrameMaker+SGML gives the element a default general rule that depends on the element's type. When you import the EDD into a template, FrameMaker+SGML 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 SGML syntax errors. You need to import the EDD into a template first so that FrameMaker+SGML 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

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+SGML 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`.

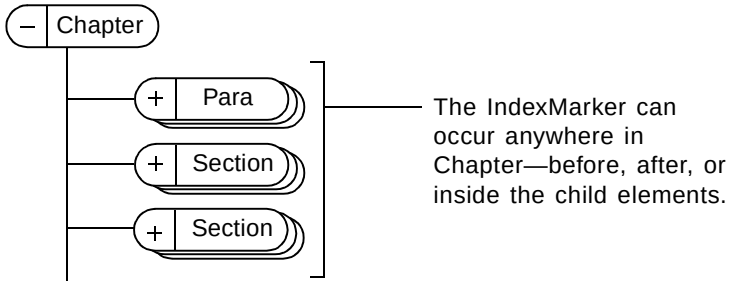
Inclusions

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 SGML

In FrameMaker+SGML, 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 an SGML 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+SGML.

When you convert an EDD to an SGML DTD, FrameMaker+SGML also translates content symbols in general rules:

- The content symbol `<TEXT>` translates to an SGML content token of `#PCDATA`. (`#PCDATA` can be combined with element tags, as `<TEXT>` can be in FrameMaker+SGML.)
- The general rule `<TEXTONLY>` translates to an SGML declared content of `RCDATA`.
- The general rule `<ANY>` translates to the reserved name `ANY` in an SGML content model.
- The general rule `<EMPTY>` translates to an SGML declared content of `EMPTY`.

When you convert a DTD to an EDD, FrameMaker+SGML performs the translations in reverse. In addition, an SGML declared content of `CDATA` translates to `<TEXTONLY>`.

For more detailed information on how content rules translate to SGML, see Chapter 12, “Translating Elements and Their Attributes.” For information on how structured tables translate to SGML, see Chapter 14, “Translating Tables.”

Inserting descendants automatically in containers

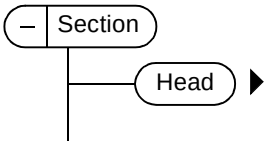
In the definition of a container, you can specify nested descendants to insert automatically. Whenever an end user inserts the container in a document, the descendants are inserted automatically along with it. This makes it convenient for users to work with containers that always begin with the same 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 one `InsertChild` element and 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:

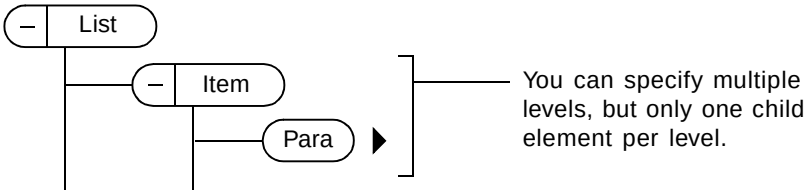


FrameMaker+SGML cannot insert sibling elements automatically. In the `Section` and `Head` example above, even though a `Section` may always begin with a `Head` and then a `Para`, you can have only the `Head` inserted automatically and not a `Para` too.

You can, however, have FrameMaker+SGML 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+SGML 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+SGML 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+SGML 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+SGML 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+SGML 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+SGML 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`:

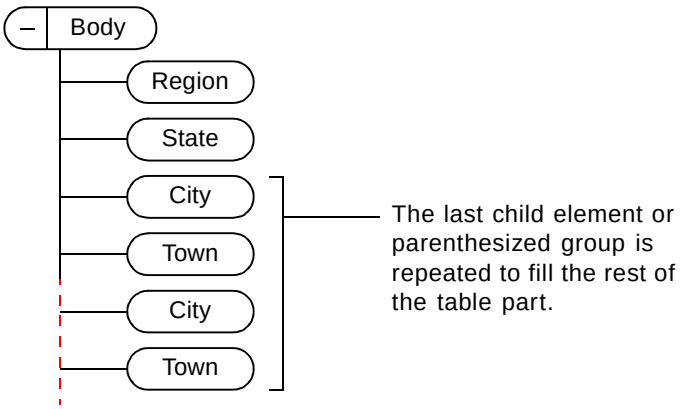
Element (Table): Table
General rule: Title?, Heading, (RulesBody | StandardBody)
Initial structure pattern: Heading, StandardBody

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 178.

For a heading, body, footing, or row, if the table part needs more elements than appear in the structure pattern, FrameMaker+SGML 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 102.

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+SGML uses the general rule to give a new table or table part its initial structure.

For a table element, FrameMaker+SGML 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:

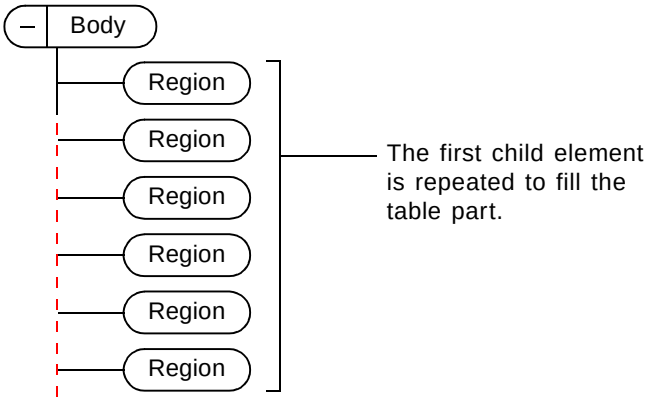
Element (Table): Table
General rule: Title?, Heading, (Normal | Special)

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+SGML automatically inserts the necessary child elements to build a basic structure.

You can define an initial structure pattern for a Rubi group and FrameMaker+SGML 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+SGML 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+SGML 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 101.)
- 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+SGML 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 91.

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

7 Text Format Rules for Containers, Tables, and Footnotes

You can define text formatting properties for containers, tables, table parts, and footnotes in FrameMaker+SGML. 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+SGML is hierarchical—an element can inherit its properties from ancestors and pass on its properties to descendants.

SGML does not provide a mechanism for formatting text, so the text format rules you write in FrameMaker+SGML do not have counterparts in SGML. If you import an SGML DTD, you can add format rules to the resulting EDD for use in FrameMaker+SGML. If you import an SGML 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 SGML, 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+SGML. In the outline below, click a topic to go to its page.

Background on text format rules and inheritance:

- "Overview of text format rules" on page 114
- "How elements inherit formatting information" on page 115

Syntax of text format rules:

- "Specifying an element paragraph format" on page 119
- "Writing context-dependent format rules" on page 120
- "Defining the formatting changes in a rule" on page 131
- "Specifications for individual format properties" on page 133

Other format rules for special purposes:

- "Writing first and last format rules" on page 143
- "Defining prefixes and suffixes" on page 145
- "When to use an autonumber, prefix or suffix, or first or last rule" on page 150

Format change lists you can refer to and limits on values in change lists:

- "Defining a format change list" on page 151

• "Setting minimum and maximum limits on properties" on page 152

Information to help you correct errors in format rules:

• "Debugging text format rules" on page 154

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+SGML 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 — The base paragraph format for all Head elements
    1. Count ancestors named: Section
      If level is: 1
        No additional formatting.
      Else, if level is: 2
        Default font properties
          Size: 14pt

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+SGML 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+SGML 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+SGML searches up through the element's ancestors until it finds an element with a format and then uses that format.

  In general, if FrameMaker+SGML 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 117 or "Inheritance in a document within a book" on page 118.)

After FrameMaker+SGML 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 — This paragraph format applies to Section and its descendants.
  - **1. If context is:** Section
    - **Basic properties**
      - **Move first indent by:** +0.5"
      - **Move left indent by:** +0.5"

Element (Container): Head
- **General rule:** <TEXT>
- **Text formal rules**
  - **1. In all contexts**
    - **Default font properties**
      - **Weight:** Bold
      - **Size:** 14pt

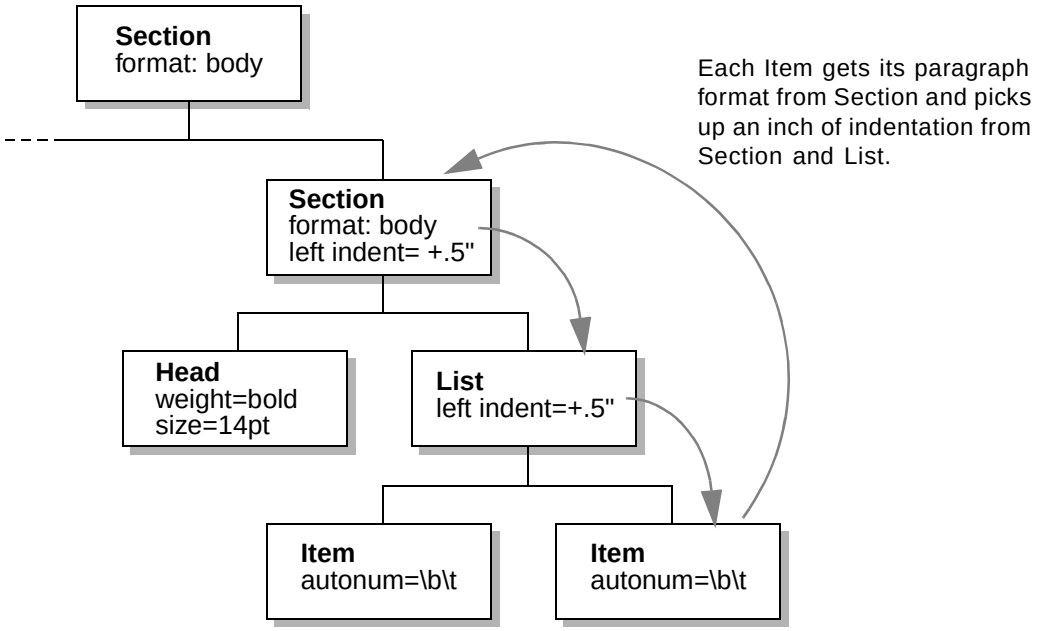
Element (Container): Para
- **General rule:** <TEXT>

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+SGML formats text in the `Item` elements by searching up to the second-level `Section` for a paragraph format. The `Item` elements are left-indented .5 inch more by the format rule for the second-level `Section` and another .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+SGML 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+SGML 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+SGML 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+SGML 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).

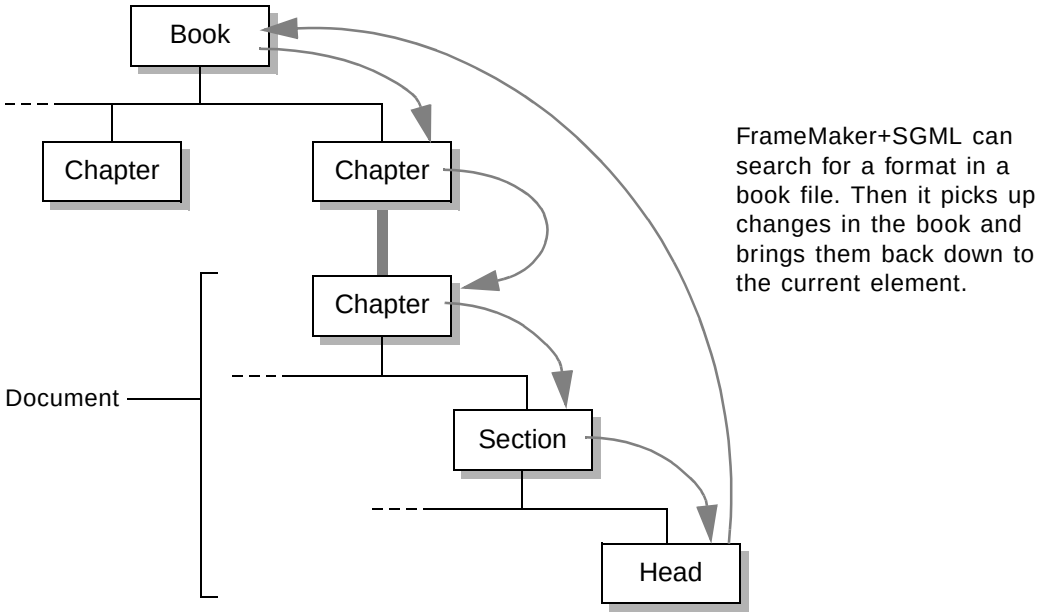
The Paragraph Catalog in a document stores default `Footnote` and `TableFootnote` paragraph formats, and an end user can also define custom footnote paragraph formats. FrameMaker+SGML applies whichever format is specified in the document's Footnote Properties or Footnote Table Properties dialog box. 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+SGML searches through an element's ancestors for a paragraph format in the same way that it does in other documents. If FrameMaker+SGML 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+SGML 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+SGML 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.



Only a document's main flow is considered to be part of a book's hierarchy. FrameMaker+SGML 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+SGML 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 115.

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+SGML 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.

This section 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 131.

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 115.

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, the text in a `TableHead` element is in 14-pt boldface no matter where the element occurs in a document:

Element (Table Head): TableHead
General rule: TableHeadRow+
Text format rules
1. **In all contexts.**
Default font properties
Size: 14pt
Weight: Bold

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+SGML 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 125.)

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+SGML 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.

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 143.

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.)

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 the value `Numbered`:

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 = "Numbered"]
   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+SGML 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 12, “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+SGML 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+SGML 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 count is: 1
   Default font properties
   Font size: 18
   If count is: 2
   Default font properties
   Font size: 14
   If count 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+SGML 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 .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+SGML 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 .5 inch or 1 inch. The `Item` is indented .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 count is: 2
        Basic properties
            Move left indent by: +0.5"
2. **Count ancestors named:** List
    Stop counting at first ancestor named: Chapter
    If count 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+SGML 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+SGML 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+SGML 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+SGML 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 count is:** 1
       - **Context label:** 1st-level
       - **Default font properties**
         - **Font size:** 18
     - **Else, if count 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.

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 133.

- 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 151.

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+SGML 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 137.

- 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 151.

For example:

Element (Container): Emphasis
General rule: <TEXT>
Text format rules
1. **In all contexts.**
   Text range. — A text range label appears when you format a container as a text range.
   Font properties
   Angle: Oblique

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+SGML 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+SGML 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 152.

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+SGML 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+SGML 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+SGML calculates the spacing at that point and overrides the value you set earlier.)

For determining the space between two adjacent paragraphs, FrameMaker+SGML 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+SGML 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+SGML 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+SGML 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+SGML interrupts a value of 20 as 20% of an em space.) A positive value moves the element up; a negative value moves it down.
`Outline` *`boolean`*	If `Yes`, applies an outline style to text (only on a Macintosh).
`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.
`Shadow` *`boolean`*	If `Yes`, applies a shadow style to text (only on a Macintosh).
`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 136.)

`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 136.)
`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.

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` *`Dimension`* For a frame above or below a paragraph, sets the horizontal distance from the paragraph. Type a relative value from the left indent of the paragraph to the frame. A positive value moves the frame right; a negative value moves it left.
`Hyphenation`	Hyphenation properties for all text in an element.

`Hyphenate` *`boolean`* If `Yes`, turns on automatic hyphenation for the element.

`Language` *`keyword`* Sets a language dictionary to use for hyphenation and spell checking. Insert a keyword child element to specify a dictionary.

`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+SGML 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+SGML 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.)

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.

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 120 and "Defining the formatting changes in a rule" on page 131.

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 150.

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+SGML 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 147.

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+SGML, 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 120 and "Text range formatting" on page 131.

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 150.

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+SGML 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 140.

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+SGML 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:

Element (Container): AuthorNote
- **General rule:** <TEXT>
- **Text format rules**
  1. **If context is:** Para
     - **Text range.**
     - **Font properties**
       - **Angle:** Italic

     Inside a paragraph, a note is a text range with italicized text.

  Else
     - **Basic properties**
       - **Indents**
         - **Left indent:** 40pt
     - **Default font properties**
       - **Angle:** Italic

     Outside a paragraph, a note is an indented paragraph with italicized text.
- **Prefix rules**
  1. **In all contexts.**
     - **Prefix:** [Author's comments:
     - **Text range.**
       - **Font properties**
         - **Weight:** Bold
- **Suffix rules**
  1. **In all contexts.**
     - **Suffix:**]
     - **Text range.**
       - **Font properties**
         - **Weight:** Bold

A note always has a prefix and a suffix in bold italics. (The italics come from the text format rules.)

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+SGML 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+SGML 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+SGML 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+SGML 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+SGML 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 140 (for autonumber strings)
- "Writing first and last format rules" on page 143
- "Defining prefixes and suffixes" on page 145

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 133.

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 78.

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 131.

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+SGML 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+SGML 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 133.

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

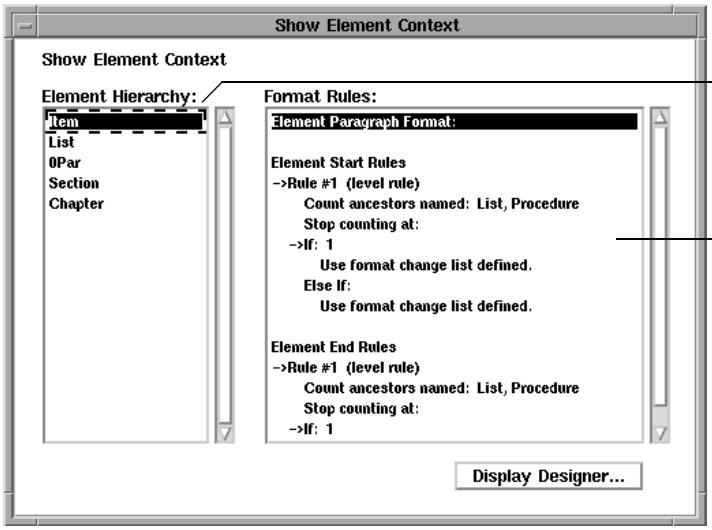
You might also need to look at formats in the document or the template to see if they are defined the way you expect. Check the definitions in these places:

- Paragraph formats in the Paragraph Designer (Format menu)
- Character formats in the Character Designer (Format menu)

If FrameMaker+SGML 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 91.

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 current element and its ancestors

The format rules for the element selected on the left. Arrows identify the clauses that apply to the instance.

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 185.

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.

8 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+SGML, 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+SGML.

Attribute definitions in FrameMaker+SGML translate to attribute declarations in SGML. If you move documents between FrameMaker+SGML and SGML and conform to SGML requirements, your attributes and their values are usually preserved. FrameMaker+SGML 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+SGML. In the outline below, click a topic to go to its page.

Background on attributes in FrameMaker+SGML documents:

- "Some uses for attributes" on page 157
- "How an end user works with attributes" on page 158

Syntax of attribute definitions:

- "Writing attribute definitions for an element" on page 159

Attributes for special purposes:

- "Using UniqueID and IDReference attributes" on page 164
- "Using attributes to format elements" on page 169
- "Using attributes to provide a prefix or suffix" on page 170

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=pjr`).
- In SGML, a size attribute in a graphic element can specify the width of a frame (`ArtWidth=8.5`).

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+SGML 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.

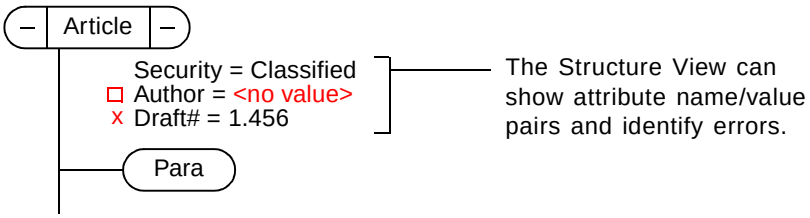
An end user supplies values for an attribute in the Attributes dialog box. The dialog box shows information from the attribute definition—name, type, range or choices, and default value—to guide the user to enter values that are appropriate. If the user tries to enter a value that is not valid according to the definition, FrameMaker+SGML does not accept the value and displays an alert.

You can also define read-only attributes with values that are set by an SGML API client or by a FrameMaker+SGML 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 an SGML 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+SGML 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+SGML 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+SGML.

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+SGML, but you should try to keep the names concise. End users often display attribute names with their elements in the Structure View.

> **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 209.

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

Type	Values allowed in the attribute
`UniqueID`	A string that uniquely identifies the element
`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 163 or "Range of values for numeric attributes" on page 163.
- The numeric types allow only integer or real values and no additional strings. This allows for precise validation in FrameMaker+SGML, 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 164.
- 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+SGML 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+SGML identifies the attribute as invalid. (A duplicate `UniqueID` value is discarded.)

> **SGML:** All attributes in FrameMaker+SGML translate to attributes in SGML. FrameMaker+SGML 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+SGML type, and vice versa. For more information, see Chapter 12, "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+SGML 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 an SGML API client. For more information, see "Default value" on page 164.

Hidden and Read-only attributes

You may sometimes want to restrict end users from editing the value of an attribute, and let FrameMaker+SGML or an SGML 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+SGML 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+SGML 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 164.

Your SGML 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+SGML 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**

> 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+SGML user interface, you can still edit its value with an SGML API client. For information on API clients, see the *SGML 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 123.

> **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 213.

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+SGML 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+SGML 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+SGML identifies the attribute value as invalid.

Default value

If a value is optional in an attribute, you can specify a default value. FrameMaker+SGML 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+SGML can use attribute values in formatting, see "Using attributes to format elements" on page 169. For information on SGML API clients, see the *SGML 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+SGML, this ID value is often used as a source address for cross-references to the element. An SGML API client or an SGML application might also use the ID for other purposes.

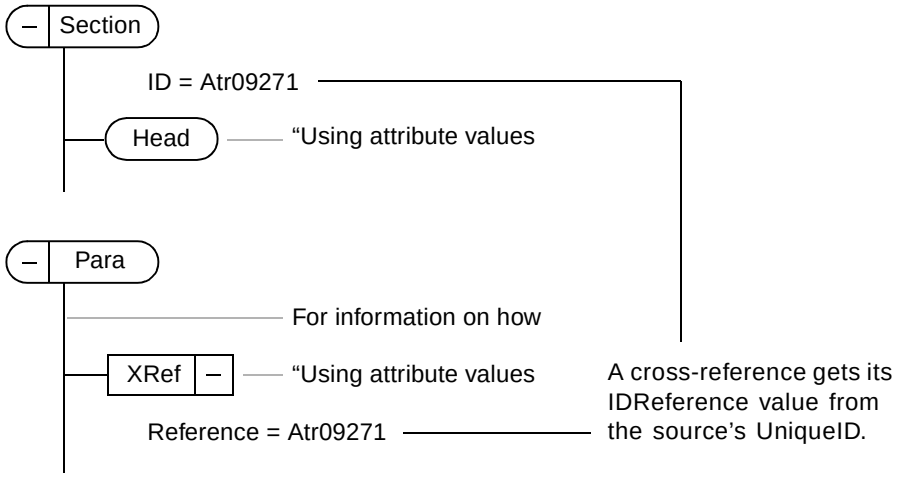
When a `UniqueID` attribute is used for element-based cross-referencing, the elements that reference the ID have an `IDReference` attribute. The `IDReference` attributes store the address from the source's `UniqueID` attribute.

FrameMaker+SGML 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+SGML

automatically fills in the cross-reference's `IDReference` attribute with the value from the source's `UniqueID` attribute.

With FrameMaker+SGML 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 182.

This example shows how a FrameMaker+SGML 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+SGML 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+SGML 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.

> **SGML:** When importing and exporting between FrameMaker+SGML and SGML, the UniqueID and IDReference attributes are preserved. UniqueID attributes in FrameMaker+SGML translate to ID attributes in SGML, and IDReference attributes translate to IDREF attributes. For more information, see Chapter 16, "Translating Cross-References."

UniqueID attributes

You can assign a UniqueID attribute to any element in FrameMaker+SGML. 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+SGML identifies duplicate IDs as invalid.

An end user can provide ID values, or FrameMaker+SGML 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+SGML 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+SGML 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+SGML 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+SGML 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+SGML 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+SGML 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+SGML 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+SGML 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.

> **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+SGML 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+SGML 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+SGML 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 SGML, 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+SGML generates thereafter, but any existing values are not replaced.

FrameMaker+SGML 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+SGML 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 162.

IDReference attributes

You can assign an `IDReference` or `IDReferences` attribute to any element in FrameMaker+SGML. If you plan to use the element as a FrameMaker+SGML 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+SGML automatically fills in the `IDReference` with the value from the source's `UniqueID`.

FrameMaker+SGML 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+SGML identifies the attribute as invalid.

SGML: For element-based cross-referencing in FrameMaker+SGML, 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+SGML 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+SGML cross-references, see page 165.

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+SGML 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+SGML 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+SGML 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+SGML 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 123.

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+SGML 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 149.

9 Object Format Rules

A FrameMaker+SGML 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+SGML object format rules have no direct counterparts in SGML. If you import an SGML DTD, you can add format rules to the resulting EDD for the objects you plan to use in FrameMaker+SGML. If you import an SGML 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 SGML, the object formatting information is not preserved.

In this chapter

This chapter explains how to write object format rules for tables, graphics, markers, cross-references, equations, and system variables in FrameMaker+SGML. In the outline below, click a topic to go to its page.

Background on object format rules:

- "Overview of object format rules" on page 174
- "Context specifications for object format rules" on page 175

Syntax and uses for object format rules:

- "Setting a table format" on page 178
- "Specifying a graphic content type" on page 179
- "Setting a marker type" on page 180
- "Setting a cross-reference format" on page 182
- "Setting an equation size" on page 182
- "Specifying a system variable" on page 183

Information to help you correct errors in format rules:

- "Debugging object format rules" on page 185

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+SGML 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 — A format rule clause has a context specification…
   Table format: StandardTable — …and a formatting property for that context.
   Else, if context is: * < Appendix
   Table format: SyntaxTable

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. This section describes the rules for "in all contexts" and the clauses for particular contexts, which are the specifications you're most likely to use in object formatting. For information on nested rules, context labels, and count statements in level rules, see Chapter 7, "Text Format Rules for Containers, Tables, and Footnotes."

All-contexts rules

A format rule can specify a format 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 a format for each context. The contexts are expressed in separate if, else/if, and else clauses. When applying a format rule

to an element, FrameMaker+SGML uses the first clause in the rule that is true for the instance of the element.

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 less-than signs (<).

In this example, a `Filename` system variable element displays the full pathname of the current file if the element occurs in an `Item` in a `List` in a `Preface`, or it displays only the filename if it occurs anywhere else:

Element (System Variable): Filename
System variable format rule
1. **If context is:** Item < List < Preface
   Use system variable: Filename (Long)
   Else
   Use system variable: Filename (Short)

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.

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

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+SGML 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.

Attribute indicators

You can also use an attribute name/value pair in an object 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.)

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.

In the following 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 determines a context for object formatting.

The syntax for using attributes in object format rules is the same as it is in text format rules. For details, see "Attribute indicators" on page 123.

Order of context clauses

When a context-specific format rule has more than one clause, keep in mind that FrameMaker+SGML 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 format to a cross-reference when it appears in an `Item` inside a nested `List` element (a `List` inside a `List`). If you put the context specifications for the cross-reference in the following order, FrameMaker+SGML would never apply the second clause because an `Item` in a nested `List` also matches the first specification:

Item < List
Item < List < List

You get the effect you want by reversing the clauses.

Setting a table format

A FrameMaker+SGML 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 SGML documents, the table format may be specified through a mapping of an attribute to the `fm property` table format. See "is fm property" on page 396 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 SGML attribute 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 7, "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.

Setting a marker type

FrameMaker+SGML 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 SGML 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 an SGML 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 SGML documents, the cross-reference format may be specified through a mapping of an attribute to the `fm property` cross-reference format. See "is fm property" on page 396 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 SGML attribute 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+SGML 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.

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+SGML 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>, <$year>	May 15, 1995
CreationDateShort	<$monthnum>/<$daynum>/ <$shortyear>	5/15/95
CurrentDateLong	<$monthname> <$dayname>, <$year>	July 6, 1995
CurrentDateShort	<$monthnum>/<$daynum>/ <$shortyear>	7/6/95
CurrentPageNum	<$curpagenum>	3
FilenameLong	<$fullfilename>	/usr/devguide/ objrules
FilenameShort	<$filename>	objrules
ModificationDateLong	<$monthname> <$dayname>, <$year>, <$hour>:<$minute00> <$ampm>	August 1, 1995, 2:30 pm
ModificationDateShort	<$monthnum>/<$daynum>/ <$shortyear>	8/1/95
PageCount	<$lastpagenum>	18
RunningHF1	<$paratext[Title]>	The Turbulent Oceans
RunningHF2	<$paratext[Heading1]>	Threat of Extinction
RunningHF3	<$marker1>	Inspection Checklist
RunningHF4	<$marker 2>	Drawing Objects
TableContinuation	(Continued)	(Continued)
TableSheet	(<Sheet <$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+SGML 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

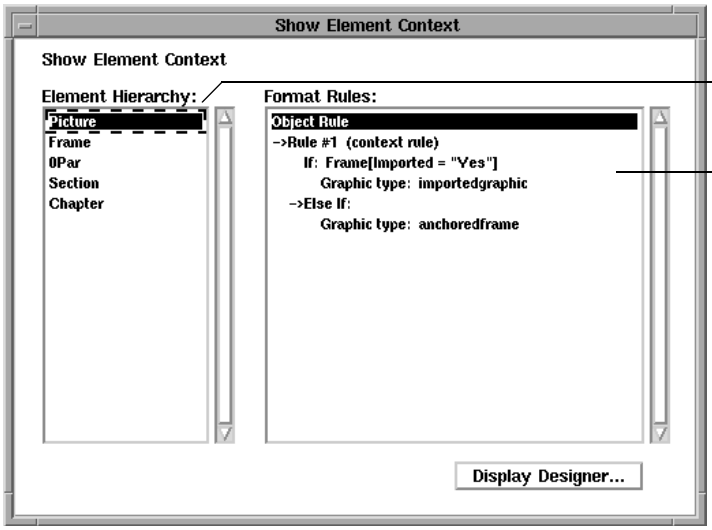
You might also need to look at formats in the document or the template to see if they are defined the way you expect. Check the definitions in these places:

- 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+SGML 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 91.

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.



The current element and its ancestors

The format rule for the element selected on the left. An arrow identifies the clause that applies to the instance.

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 154.

To display the formatting properties for a table format, select the format in the list on the right and click Display Designer.

Part III Translating between SGML and FrameMaker+SGML

Part III provides details of the model FrameMaker+SGML uses for translating between SGML and FrameMaker+SGML documents and on how you can modify this translation. You should be familiar with the material in Part I, “Developing a FrameMaker+SGML application” before using the material in this part.

This part of the manual includes:

- Chapter 10, “Introduction to Translating between SGML and FrameMaker+SGML”

  Describes the types of modifications you can accomplish with SGML read/write rules. The chapter also contains a simple example of using rules to create an SGML application.

- Chapter 11, “SGML Read/Write Rules and Their Syntax”

  Describes the mechanics of writing read/write rules and the document in which you define them.

- Chapter 12, “Translating Elements and Their Attributes”
  Chapter 13, “Translating Entities and Processing Instructions”
  Chapter 14, “Translating Tables”
  Chapter 15, “Translating Graphics and Equations”
  Chapter 16, “Translating Cross-References”
  Chapter 17, “Translating Variables and System Variable Elements”
  Chapter 18, “Translating Markers”
  Chapter 19, “Processing Multiple Files as Books”

  Contain information on how FrameMaker+SGML translates various specific SGML and FrameMaker+SGML constructs. Each of these chapters describes the software’s default translation followed by modifications you can make to that translation with read/write rules.

- Chapter 20, “Read/Write Rules Summary”

  Contains a summary of the read/write rules provided for translating between various SGML and FrameMaker+SGML constructs.

- Chapter 21, “Read/Write Rules Reference”

  Contains an alphabetical list of reference material on all read/write rules.

10 Introduction to Translating between SGML and FrameMaker+SGML

FrameMaker+SGML can read and write SGML documents without any help from you. However, its default translation of SGML constructs may not be suitable for your SGML DTD. In such cases, you have to write an SGML application to help the software appropriately translate your documents.

In this chapter

This chapter talks about the information your application typically contains. In the outline below, click a topic to go to its page.

Descriptions of the types of modifications to the default translation that you can make:

- "What you can do with SGML read/write rules" on page 189
- "What you can do with SGML API clients" on page 190

An example to illustrate these points:

- "A detailed example" on page 191

What you can do with SGML read/write rules

SGML 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 SGML 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+SGML makes assumptions about how to translate these FrameMaker+SGML constructs to SGML 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+SGML cannot recognize these constructs, so your rules need to be more extensive.

Renaming elements and attributes FrameMaker+SGML element tags and attribute names are frequently more descriptive than their SGML counterparts and you might use an SGML read/write rule to establish a correspondence between names. For example, you could use a rule to change the FrameMaker+SGML tag `Employee Name` to the SGML generic identifier `ename`.

Treating special characters In SGML you can use an entity to represent special characters. You might use rules to translate such entities.

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 SGML 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+SGML 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 Chapter 21, "Read/Write Rules Reference." Sections of the chapter that describe examples of such rules include:

- "character map" on page 337 describes the rule that tells FrameMaker+SGML how to map between characters in the SGML and FrameMaker+SGML character sets.
- "external dtd" on page 363 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+SGML includes an external DTD subset.
- "line break" on page 417 describes the rule that tells FrameMaker+SGML how to interpret line breaks when importing an SGML document and when to generate line breaks when exporting a FrameMaker+SGML document.
- "write sgml document" on page 441 and "write sgml document instance only" on page 441 describe the rules that tell FrameMaker+SGML when saving as SGML whether it should write only an SGML document instance or an entire document.

What you can do with SGML API clients

In situations in which SGML read/write rules are insufficient to express the proper translation between SGML and FrameMaker+SGML, you can create an SGML API client. You use the Frame Developer's Kit (FDK) to modify the import or export of SGML documents. You cannot use the FDK to modify the creation of a DTD or an EDD.

Working with the FDK requires significant C and FDK programming skills. The FDK allows you an arbitrary amount of control over FrameMaker+SGML's processing. For this reason, there is no way to provide a brief categorization of possible function types.

During import, your functions can use the FDK to inspect or modify the structure, format, or content of the FrameMaker+SGML document being constructed. FDK functions can process attribute values, processing instructions, and some entity references encountered in the imported SGML document.

During export, your functions can inspect the structure, format, and content of the exported FrameMaker+SGML document. They can inspect and modify every generated portion of the SGML document before it is actually written and can create new text (data and markup) to be inserted into the SGML document.

This manual does not describe how to create an SGML API client to modify the behavior of FrameMaker+SGML, although it does occasionally mention situations in which a client is appropriate. For information on creating SGML API clients, see the *SGML API Programmer's Guide*.

A detailed example

To give you a more concrete feeling for translating between SGML and FrameMaker+SGML representations of a document, 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+SGML 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 an SGML 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+SGML automatically generates most of these FrameMaker+SGML element definitions (other than their formatting rules) from the

corresponding SGML declarations without any intervention. You only need to specify format rules, and one SGML read/write rule that states that the SGML 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+SGML 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+SGML document that uses that entity; the information about the `oquote` and `cquote` entities remains solely in the rules, although a FrameMaker+SGML document created from an SGML 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+SGML document

With these rules and an appropriate template, FrameMaker+SGML translates the earlier SGML markup to the following in a FrameMaker+SGML 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+SGML document, on export FrameMaker+SGML produces SGML markup equivalent to that shown earlier. Without an SGML API client, it doesn't produce short references or markup minimization.

10 A detailed example

11 SGML Read/Write Rules and Their Syntax

SGML read/write rules are your primary device for modifying FrameMaker+SGML's default translations.

In this chapter

This chapter describes the syntax of the rules and how you create them. Chapter 21, "Read/Write Rules Reference," describes each of the rules in detail. In the outline below, click a topic to go to its page.

General description of a read/write rules document:

- "The rules document" on page 197

Discussion of the significance of the order of rules in a rules document:

- "Rule order" on page 199

Discussion of the syntax of individual rules:

- "Rule syntax" on page 199
- "Case conventions" on page 200
- "Strings and constants" on page 200
- "Comments" on page 202

Splitting the rules for an application between several files:

- "Include files" on page 202

Names you shouldn't use for your FrameMaker+SGML elements:

- "Reserved element names" on page 203

User interface commands you use when developing a read/write rules document:

- "Commands for working with a rules document" on page 203

The rules document

You create a FrameMaker+SGML document or an ASCII file containing SGML read/write rules. The software uses this rules document to modify its SGML import and export processing.

Assuming that you create a FrameMaker+SGML 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+SGML 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 SGML Read/Write Rules. This command creates a new rules document, using the template it finds in `$SGMLDIR/default.rw`. For information on the location of `$SGMLDIR`, see "Location of SGML files" on page 41.

The first rule in a rules document must be:

```
fmsgml version is "6.0";
```

Note that you would use the string `"5.5"` even though the product version may be an incremental release above 5.5, such as 5.5.1.

Most rules are relevant all the time—for example, a rule to convert the SGML general entity `pname` to the FrameMaker+SGML 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+SGML documents. It is not relevant when exporting an EDD or when importing SGML documents or DTDs, because these situations do not require generating entity names for graphics.

There is so much overlap in the rules that FrameMaker+SGML 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 SGML document, export of an EDD, or export of a FrameMaker+SGML document. Most rules are expressed from the SGML perspective; the *element* of an `element` rule is the SGML generic identifier, not the FrameMaker+SGML element tag. The only exceptions to this convention are rules for FrameMaker+SGML constructs that have no SGML counterpart. For example, a rule might specify dropping FrameMaker+SGML 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. SGML and FrameMaker+SGML 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+SGML 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+SGML elements correspond to the same SGML element, and two SGML elements correspond to the same FrameMaker+SGML element. If you import an SGML document to FrameMaker+SGML with these rules, all occurrences of the SGML `list` element become `List` elements, and occurrences of the `proc` element become `Procedure` elements. If you export a FrameMaker+SGML document to SGML, both `List` and `Procedure` elements become `list` elements. And no `proc` elements are created. The result is that if you start with an SGML document containing a `proc` element, import that document to FrameMaker+SGML, and then export the result back to SGML, 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+SGML `XRefLabel` attribute in any element in which it occurs. The second rule tells the software to translate the FrameMaker+SGML `XRefLabel` attribute to the SGML `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 SGML and FrameMaker+SGML constructs, such as elements, attributes, and entities, are represented by strings. For information on strings, see “Strings and constants” on page 200. 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

Case is not significant in rule keywords, so the following are equivalent:

```
fmsgml version is "6.0";
FMSGML Version is "6.0";
```

To improve readability, this manual sometimes uses mixed case for keywords.

The case of FrameMaker+SGML element tags and other FrameMaker+SGML 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 more information, see “Naming elements and attributes” on page 209.

Strings and constants

You use strings to specify FrameMaker+SGML element tags, SGML generic identifiers, attribute names, attribute values, entity names, notation names, and so on.

String syntax

Strings are enclosed in matching straight or directional double quotation marks. For example, both of the following forms are acceptable:

```
element "taxtable" is fm table element;
element "taxtable" is fm table element;
```

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+SGML 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 13, “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+SGML 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 `sgmlapps.fm` file. The default search path for an include file consists of the directory containing the original rules document and the directory `$SGMLDIR/isoents/`, where `$SGMLDIR` is as defined in “Location of SGML files” on page 41. In addition, these directories are added to the end of the search path you specify in `sgmlapps.fm`.

If you plan to use the same rules on different systems, avoid specifying directory names in the `#include` directive. Instead, let FrameMaker+SGML find files in different directories through a search path.

> **Important:** File paths must be specified in the syntax that is native to the platform on which you are running FrameMaker+SGML. Also, for Windows platforms, pathname tokens are separated by a backslash character (“\”). When specifying a Windows path, you must escape the backslash character. For example, the following specify the same location on Unix, Macintosh, and Windows platforms:
>
> - Unix
> `$SGMLDIR/isoents`
> - Macintosh
> `$SGMLDIR:isoents`
> - Windows
> `$SGMLDIR\\isoents`

For information on working with the `sgmlapps.fm` file, see “Application definition file” on page 42.

Reserved element names

You may use any legal name for an SGML element. If there are no rules to the contrary, FrameMaker+SGML assumes most SGML elements correspond to FrameMaker+SGML elements. The following generic identifiers are used by the default declarations for translating FrameMaker+SGML cross-references, equations, footnotes, graphics, or tables to SGML: `CROSSREF`, `EQUATION`, `FOOTNOTE`, `GRAPHIC`, `TABLE`, `TITLE`, `HEADING`, `BODY`, `FOOTING`, `ROW`, and `CELL`.

FrameMaker+SGML uses the following entity names for translating FrameMaker+SGML 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.

Commands for working with a rules document

The File>Developer Tools menu provides two commands for working with an SGML read/write rules document: New SGML Read/Write Rules and Check SGML Read/Write Rules.

- New SGML Read/Write Rules

  To create a new rules document, choose New SGML Read/Write Rules. This command uses the rules template in `$SGMLDIR/default.rw`, if there is one. For information on the `$SGMLDIR` directory, see "Location of SGML files" on page 41.

- Check SGML Read/Write Rules

  To verify the correctness of a rules document, choose Check SGML 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 SGML 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 SGML Read/Write Rules command checks for syntax errors in rules and also reports semantic errors on the basis of the DTD and the FrameMaker+SGML 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 SGML constructs not mentioned in the current DTD—for details, see "The rules document" on page 197.

For information on adding a DTD file or rules document to an application, see "Defining an application" on page 44.

12 Translating Elements and Their Attributes

Elements and their attributes are the fundamental components of both FrameMaker+SGML and SGML documents.

Most information in this chapter is “bidirectional.” It is about translations that can be recognized and performed when both reading and writing SGML documents and DTDs, and the examples apply to both cases. You can start with a DTD containing SGML element and attribute declarations and produce the corresponding FrameMaker+SGML element definition, or you can start with an EDD containing the FrameMaker+SGML element definition and produce the corresponding SGML element and attribute declarations.

In this chapter

This chapter gives you an overview of how FrameMaker+SGML translates between its representation of elements and attributes and SGML’s representation. In the outline below, click a topic to go to its page.

How FrameMaker+SGML translates elements by default:

- “Translating model groups and general rules” on page 206
- “Translating attributes” on page 207
- “Naming elements and attributes” on page 209
- “Inclusions and exclusions” on page 210
- “Line breaks and record ends” on page 211

Some ways you can change the default translation:

- “Renaming elements” on page 211
- “Renaming attributes” on page 212
- “Renaming attribute values” on page 213
- “Translating an SGML element to a footnote element” on page 213
- “Translating an SGML element to a Rubi group element” on page 214
- “Changing the declared content of an SGML element associated with a text-only element” on page 215
- “Retaining content but not structure of an element” on page 216
- “Retaining structure but not content of an element” on page 216

- “Formatting an element as a boxed set of paragraphs” on page 217
- “Suppressing the display of an element’s content” on page 217
- “Discarding an SGML or FrameMaker+SGML element” on page 218
- “Discarding an SGML or FrameMaker+SGML attribute” on page 218
- “Specifying a default value for an attribute” on page 219
- “Changing an attribute’s type or declared value” on page 220
- “Creating read-only attributes” on page 221
- “Using SGML attributes to specify FrameMaker+SGML formatting information” on page 222

This chapter gives you information about elements and their attributes as a general class. For information on how to translate elements and attributes used for a particular purpose, see Chapters 14 through 19.

Default translation

The basic representations of elements and attributes in SGML and in FrameMaker+SGML 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.

Translating model groups and general rules

The general rule of a FrameMaker+SGML element uses a syntax based on SGML model groups. FrameMaker+SGML 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+SGML element definition is:

Element (Container): Lablist
General rule: Head, Par?, Item+

When FrameMaker+SGML 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+SGML general rule.
- A content model of `ANY` translates to the FrameMaker+SGML general rule `<ANY>`.
- A declared content of either `CDATA` or `RCDATA` translates to the FrameMaker+SGML general rule `<TEXTONLY>`.
- A declared content of `EMPTY` translates to the FrameMaker+SGML general rule `<EMPTY>`.

When FrameMaker+SGML 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 `RCDATA`.

Translating attributes

In SGML, attribute declarations for an element occur in a separate attribute definition list declaration. In FrameMaker+SGML, 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+SGML:

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 SGML attributes become optional attributes in FrameMaker+SGML. In addition, the interpreted attribute value specification given in the default value for the SGML `type` attribute translates to a default value in FrameMaker+SGML.

In general, any SGML attribute that is not a required attribute (that is, doesn't use the `#REQUIRED` declared value) becomes an optional attribute in FrameMaker+SGML. If the SGML attribute has an attribute value specification, the interpreted version of that value becomes the attribute's default value in FrameMaker+SGML.

By default, a small number of SGML attributes translate to formatting properties in FrameMaker+SGML. This happens only in the case of graphics, equations, and tables that use a standard representation recognized by FrameMaker+SGML. For information on these attributes, see Chapter 14, "Translating Tables," and Chapter 15, "Translating Graphics and Equations."

When FrameMaker+SGML writes a FrameMaker+SGML document as SGML, it writes SGML attribute specifications for attributes with an explicitly supplied value. Such explicit values may be entered directly by the end user, created by the FrameMaker+SGML cross-reference facility, or supplied by an FDK client.

Conversely, on import of an SGML document, if an element's start-tag doesn't include a value for an attribute, then FrameMaker+SGML doesn't supply a value, either.

Attribute types and declared values

FrameMaker+SGML has a set of attribute types that correspond to the SGML declared values for attributes, but neither is a subset of the other. That is, multiple SGML declared values can become the same FrameMaker+SGML attribute type, and conversely multiple FrameMaker+SGML attribute types can become the same SGML declared value. (SGML does 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+SGML uses a default translation between attribute types and declared values.

On import, FrameMaker+SGML makes the following conversions in the absence of read/write rules:

SGML Declared Value	FrameMaker+SGML Attribute Type
CDATA	String
ENTITY	String
ENTITIES	Strings
ID	UniqueID
IDREF	IDReference
IDREFS	IDReferences
NAME	String
NAMES	Strings
NMTOKEN	String
NMTOKENS	Strings
NUMBER	Integer
NUMBERS	Integers
NUTOKEN	String
NUTOKENS	Strings
NOTATION	Choice
Name token group	Choice

On export, FrameMaker+SGML makes the following conversions in the absence of read/write rules:

FrameMaker+SGML Attribute Type	SGML Declared Value
String	CDATA

FrameMaker+SGML Attribute Type	SGML Declared Value
`Strings`	`CDATA`
`Integer`	`NUMBER` if values are restricted to be positive, `CDATA` otherwise.
`Integers`	`NUMBERS` if values are restricted to be positive, `CDATA` otherwise
`Real`	`NUTOKEN` if values are restricted to be positive, `CDATA` otherwise
`Reals`	`NUTOKENS` if values are restricted to be positive, `CDATA` otherwise
`UniqueID`	`ID`
`IDReference`	`IDREF`
`IDReferences`	`IDREFS`
`Choice`	Name token group

Naming elements and attributes

Legal names in FrameMaker+SGML and in SGML are different. FrameMaker+SGML 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.

If you create an EDD by importing a DTD, you won't have trouble with illegal names. Any name that is legal in the SGML reference concrete syntax is legal in FrameMaker+SGML. However, if you create a 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 reflect the differences.

If you do not specify the name for an element or attribute when translating between SGML and FrameMaker+SGML, FrameMaker+SGML must pick a name to use. The chosen name has the same characters as the original; however, SGML and FrameMaker+SGML have different conventions regarding the case of names. To ensure that the case of a name is appropriately chosen, FrameMaker+SGML uses this information:

- If specified, an `element` rule that explicitly translates an SGML generic identifier to a FrameMaker+SGML element tag

  If an `element` rule gives both SGML and FrameMaker+SGML names for an element, the software uses the names and their case as specified.

- If specified, an `attribute` rule that explicitly translates an SGML attribute name to a FrameMaker+SGML attribute name

  If an `attribute` rule gives both SGML and FrameMaker+SGML 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+SGML 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+SGML element tag `Part`. The software performs the same conversion for attribute names.
  - On export, the software converts a FrameMaker+SGML element tag to a generic identifier that has all lowercase characters. So, each of the FrameMaker+SGML 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+SGML export as the same element GI or attribute name. For example, FrameMaker+SGML 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+SGML elements become distinct SGML elements as intended.

Inclusions and exclusions

Just as SGML content models can specify exceptions to the model group, FrameMaker+SGML 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+SGML produces this FrameMaker+SGML 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+SGML produces this SGML element declaration:

```
<!ELEMENT appendix - - (title, section+) -(list, table)>
```

Line breaks and record ends

You can control the behavior of SGML record ends and FrameMaker+SGML line breaks on import and export of documents.

By default, when importing an SGML document, FrameMaker+SGML 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 SGML parser.

By default, when exporting a FrameMaker+SGML document, FrameMaker+SGML 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 an SGML record end at the end of every paragraph and flow in the FrameMaker+SGML document.
- Within the content of an SGML element that can contain child elements but no text, FrameMaker+SGML writes a record end after every start-tag and before every end-tag.
- Within a start-tag, it puts a record end followed by a tab character between every set of attribute-value pairs.

For information on how you can change this behavior, see “line break” on page 417.

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+SGML and SGML.

The following sections describe some of the possible modifications to the default translations. Your situation may require different rules or an SGML 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 “All Elements” on page 323. For information on writing SGML API clients, see the*SGML API Programmer’s Guide*.

Renaming elements

There are many reasons you may choose to rename elements when translating between SGML and FrameMaker+SGML. As already mentioned, the FrameMaker+SGML 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.

The general form of the rule for renaming an element is:

```
element "gi"
   is fm element "fmtag";
```

where *gi* is an SGML generic identifier and *fmtag* is a FrameMaker+SGML element tag. For example, if you have an SGML element `par` the FrameMaker+SGML 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 SGML element `par` as the FrameMaker+SGML element `Paragraph`. Conversely, export of a FrameMaker+SGML document or EDD translates the FrameMaker+SGML element `Paragraph` as the SGML element `par`.

For information on these rules, see "element" on page 345 and "is fm element" on page 391.

Renaming attributes

Just as you can choose to rename elements, you can choose to rename their attributes when translating between SGML and FrameMaker+SGML. 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 "sgmlattr"
   is fm attribute "fmattr";
```

```
element "gi"
   attribute "sgmlattr"
       is fm attribute "fmattr";
```

where *sgmlattr* is an SGML attribute name, *fmattr* is a FrameMaker+SGML attribute name, and *gi* is an SGML generic identifier.

The first form renames the attribute no matter in which element it occurs. For example, if you have an SGML attribute `sec` the FrameMaker+SGML 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 SGML attribute `sec` as the FrameMaker+SGML attribute `Security`. Conversely, export of a FrameMaker+SGML document or EDD translates the FrameMaker+SGML attribute `Security` as the SGML 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+SGML. You could use these rules:

```
element "article"
   attribute "ref" is fm attribute "Referee";
```

```
element "xref" {
   is fm cross-reference element "XRef";
   attribute "ref" is fm attribute "ID";
}
```

For information on these rules, see “attribute” on page 335, “element” on page 345, “is fm attribute” on page 385, and “is fm cross-reference element” on page 389.

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 SGML and FrameMaker+SGML. 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 SGML attribute, and *gi* is an SGML 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 “element” on page 345, “attribute” on page 335, “value” on page 439, and “is fm value” on page 413.

Translating an SGML element to a footnote element

Many documents require footnotes to provide additional information. Handling of footnotes is primarily a formatting issue. Because of this, SGML does not directly address them. FrameMaker+SGML, however, has a special type of element for creating and formatting footnotes.

Since footnote elements are elements, there is no problem when writing a FrameMaker+SGML document as SGML. A footnote element becomes an SGML element. However, if you have an SGML document or DTD with an element that you want to represent a footnote in FrameMaker+SGML, you must write a rule to indicate this.

For example, to translate the SGML element `fn` as the `Footnote` footnote element in FrameMaker+SGML, use this rule:

```
element "fn" is fm footnote element "Footnote";
```

In FrameMaker+SGML, a footnote element is always a footnote element. That is, you can't have the same element format as a footnote in some contexts and as a normal paragraph in other contexts. If you have a single SGML element that can be formatted as both a footnote and as a normal paragraph, you'll need to translate the same SGML element to two different FrameMaker+SGML elements. This translation cannot be accomplished with read/write rules. In this case, you'll need to write an SGML API client.

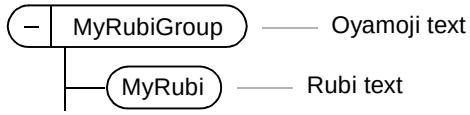
For information on these rules, see "element" on page 345 and "is fm footnote element" on page 393. For information on writing SGML API clients, see the *SGML API Programmer's Guide*.

Translating an SGML 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 SGML does not directly address Rubi. However, FrameMaker+SGML 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+SGML document as SGML a Rubi Group element and its contents become SGML elements. However, if you have an SGML document or DTD with an element you want to translate as a Rubi group in FrameMaker+SGML, you must use read/write rules to indicate this.

The minimal structure for a Rubi group is:



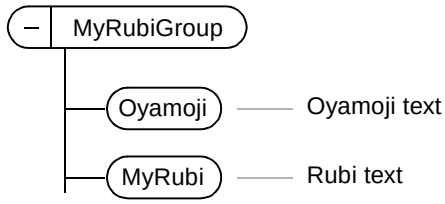
One typical way to represent a Rubi group in SGML includes Oyamoji in a container element as follows:

```
<rubigroup>
   <oyamoji>Oyamoji text</oyamoji>
   <rubi>Rubi text</rubi>
</rubigroup>
```

To translate this Rubi group, use these rules:

```
element "rubigroup" is fm rubi group element "MyRubiGroup";
element "rubi" is fm rubi element "MyRubi";
```

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 SGML will translate with the following structure:



Another typical representation of a Rubi group in SGML is:

```
<rubigroup rubi = "Rubi text">Oyamoji text</rubigroup>
```

FrameMaker+SGML 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 SGML API client.

If you have an 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 SGML element to two different FrameMaker+SGML elements. This translation cannot be accomplished with read/write rules. In this case, you will need to write an SGML API client.

For information on these rules, see "element" on page 345, "is fm rubi group element" on page 405, and "is fm rubi element" on page 404. For information on writing SGML API clients, see the *SGML API Programmer's Guide*.

Changing the declared content of an SGML element associated with a text-only element

By default, FrameMaker+SGML 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 SGML 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+SGML 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 SGML 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 SGML, 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 SGML, you must write an SGML API client 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 O O (to & from & date)>
<!ELEMENT body O O (par+)>
```

You can choose to omit the `top` and `body` elements on import to FrameMaker+SGML. 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 “element” on page 345, “fm element” on page 367, and “unwrap” on page 436.

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 “preserve fm element definition” on page 422.

For information on creating an SGML API client for your SGML application, see the *SGML API Programmer’s Guide*.

Retaining structure but not content of an element

You can use some elements in SGML or FrameMaker+SGML as place holders for information generated by a tool. For example, FrameMaker+SGML can generate a table of contents for a document. So when you import an SGML 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 SGML, you depend on an SGML application to create the appropriate content for the `toc` element. In FrameMaker+SGML, 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+SGML, you have two choices. You can require the end user to manually call the Generate command. You can create an SGML API client to generate the table of contents as the last step in importing the document. With this rule, when you export the FrameMaker+SGML document to SGML the `toc` element retains its content in the exported SGML document.

For information on these rules, see "element" on page 345, "drop content" on page 344, "reader" on page 427, and "writer" on page 442. For information on creating an SGML API client for your SGML application, see the *SGML API Programmer's Guide*.

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+SGML, you must have a table element to get this formatting. For information on how to translate a single SGML element as a one-cell table, see "Using a table to format an element as a boxed set of paragraphs" on page 270.

Suppressing the display of an element's content

You may have an SGML element you want to retain but whose content you do not want displayed when you translate SGML documents containing the element to documents in FrameMaker+SGML. 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 18, “Translating Markers.” For more information on these rules, see “element” on page 345, “is fm marker element” on page 395, and “marker text is” on page 418.

Alternatively, you may not want to create a marker element from your SGML 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 an SGML API client. For example, you could have an SGML API client attach a condition tag to the element and then hide that condition tag. For information on creating SGML API clients, see the *SGML API Programmer’s Guide*.

Discarding an SGML or FrameMaker+SGML element

You can use SGML elements for purposes that have no counterpart in FrameMaker+SGML, and the reverse is also true. For example, assume you use an SGML element `break` to indicate that a page or line break is desired. You can discard this element when the SGML document is imported to FrameMaker+SGML and have an SGML API client insert the necessary break; there is no need to mark the break with an element. To indicate that import of an SGML document should discard the `break` element, 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 SGML representation. If your table is to have column heading in FrameMaker+SGML, however, they must be in elements so you would discard these elements on export, using this rule:

```
fm element "ColumnHead" drop;
```

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 an SGML API client.

For information on these rules, see “element” on page 345, “fm element” on page 367, and “drop” on page 342.

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 “preserve fm element definition” on page 422.

For information on creating an SGML API client for your SGML application, see the *SGML API Programmer’s Guide*.

Discarding an SGML or FrameMaker+SGML attribute

If you use SGML attributes for purposes that have no counterpart in FrameMaker+SGML, or for purposes for which you must supply an SGML 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 SGML or FrameMaker+SGML 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 an SGML generic identifier and *attr* is an SGML or FrameMaker+SGML attribute name.

For example, assume that you have these declarations in SGML:

```
<!ELEMENT par - O (#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 an SGML 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 an SGML API client.

For information on these rules, see "attribute" on page 335, "drop" on page 342, "element" on page 345, and "fm attribute" on page 366. For information on writing format rules, see Chapter 7, "Text Format Rules for Containers, Tables, and Footnotes." For information on creating an SGML API client for your SGML application, see the *SGML API Programmer's Guide*.

Specifying a default value for an attribute

An SGML attribute 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 SGML, 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 SGML doesn't include a value for the `label` attribute.

FrameMaker+SGML 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 an SGML attribute and *val* is the proposed value for that attribute.

To specify a value only for the attribute within a particular SGML element, use this rule:

```
element "gi" attribute "attr" implied value is "val";
```

where *gi* is an SGML generic identifier, *attr* is an SGML attribute, 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+SGML, a default attribute value can only be specified in the EDD, so this rule has no effect when importing an SGML instance or exporting a FrameMaker+SGML document. Also, the default value is used only to initiate formatting. FrameMaker+SGML does not actually provide an attribute value for the element. When you export the FrameMaker+SGML document, the software does not generate an attribute value where there previously was none.

For more information, see “Default value” on page 164 and “Default translation” on page 206 For information on these rules, see “element” on page 345, “attribute” on page 335, and “implied value is” on page 377.

Changing an attribute’s type or declared value

SGML declared values for attributes and FrameMaker+SGML attribute types are different. In the absence of read/write rules, FrameMaker+SGML must make decisions on how to translate between them. The default translations are listed in “Attribute types and declared values” on page 208. You can change the attribute type created for an EDD or the declared value for a DTD by using the following rule:

```
[sgmldv] attribute "sgmlattr"
        is fm [read-only] [fmtype] attribute "fmattr"
        [range from a to b];
```

where *sgmldv* is an SGML declared value, *sgmlattr* is an SGML attribute name, *fmtype* is a FrameMaker+SGML attribute type, *fmattr* is a FrameMaker+SGML attribute name, and *a* and *b* are numbers. The possible values for *sgmldv* are:

cdata	id	name	number
entity	idref	names	numbers
entities	idrefs	nmtoken	nutoken
group	notation	nmtokens	nutokens

The possible values for *fmtype* are:

string	integer	unique-id

```
strings        integers       id-reference
choice         real-number    id-references
               real-numbers
```

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 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+SGML 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 SGML documents.

For information on read-only attributes, see “Creating read-only attributes,” next. For information on these rules, see “attribute” on page 335 and “is fm attribute” on page 385.

Creating read-only attributes

FrameMaker+SGML 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 an SGML API client to extract elements from an SGML database that you then combine into a FrameMaker+SGML 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 SGML API client store them as the value of an attribute of the element when read into FrameMaker+SGML. 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+SGML 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 "sgmlattr" is fm read-only attribute ["fmattr"];
```

where *`sgmlattr`* is an SGML attribute name and *`fmattr`* is a FrameMaker+SGML attribute name.

For information on these rules, see “attribute” on page 335 and “is fm attribute” on page 385.

Using SGML attributes to specify FrameMaker+SGML formatting information

A common use of attributes is for storing formatting information. FrameMaker+SGML does not recognize SGML 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 an SGML 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 SGML API client to set the size on import of an SGML document.

FrameMaker+SGML automatically handles certain FrameMaker+SGML formatting properties associated with tables, graphics, and equations. For information on these properties, see Chapter 14, “Translating Tables,” and Chapter 15, “Translating Graphics and Equations.”

For information on writing format rules, see Chapter 7, “Text Format Rules for Containers, Tables, and Footnotes.” For information on creating an SGML API client for your SGML application, see the online manual *SGML API Programmer’s Guide*, included with the FDK.

13 Translating Entities and Processing Instructions

SGML entities serve a variety of functions in SGML documents. These functions can be represented in widely different ways in FrameMaker+SGML. FrameMaker+SGML does not have a single construct that corresponds to the SGML entity. For information on common uses of entities, see "Entities" on page 13.

Processing instructions (PIs) in SGML provide a mechanism for performing system-specific actions on an SGML document. These actions can be almost anything. FrameMaker+SGML interprets a small number of processing instructions. It stores others for possible use by some other system or for interpretation by an SGML API client.

In this chapter

This chapter discusses the default translations used by FrameMaker+SGML 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+SGML constructs such as variables or books. For information on the processing instructions used for books and book components, see Chapter 19, "Processing Multiple Files as Books." In the outline below, click a topic to go to its page.

How FrameMaker+SGML translates entities and processing instructions by default:

- "On export to SGML" on page 226
- "On import to FrameMaker+SGML" on page 228

Some ways you can change the default translation:

- "Specifying the location of entity declarations" on page 235
- "Renaming entities that become variables" on page 235
- "Translating SDATA entity references on import and export" on page 235
- "Translating SDATA entities as FrameMaker+SGML variables" on page 236
- "Translating SDATA entities as special characters in FrameMaker+SGML" on page 237
- "Translating SDATA entities as FrameMaker+SGML text insets" on page 238
- "Translating SDATA entities as FrameMaker+SGML reference elements" on page 240
- "Translating external SGML text entities as text insets" on page 241
- "Translating internal SGML text entities as text insets" on page 241

- “Changing the structure and formatting of a text inset on import” on page 243
- “Discarding external data entity references” on page 244
- “Translating ISO public entities” on page 244
- “Facilitating entry of special characters that translate as SGML entities” on page 244
- “Creating book components from general entities” on page 245
- “Discarding unknown processing instructions” on page 245
- “Using entities for storing graphics or equations” on page 245

Default translation

With few exceptions, FrameMaker+SGML preserves all of the entity structure of a document. For example, while it can read a DTD that uses parameter entities, it does not mark the replacement text as an entity within the FrameMaker+SGML EDD. When exporting to SGML, it generates entity references for user variables, graphics, equations, text insets, books, and book components by default.

FrameMaker+SGML 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 but does not do anything special with them.

FrameMaker+SGML preserves SGML 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 SGML instance; if the entity is not defined in the SGML application’s DTD, the software writes an entity declaration in the internal DTD subset for the exported SGML instance.

If a referenced text file is an SGML fragment (at leasFinal Draftt one element, but no SGML declaration or DTD subset), the software interprets its structure according to the current SGML 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 SGML

FrameMaker+SGML uses entities for text insets, user variables, graphics, equations, books, and book components. If the entities are not declared in the SGML application’s DTD, FrameMaker+SGML generates appropriate entity declarations in the internal DTD subset of the exported SGML instance.

If the FrameMaker+SGML document was made by importing an SGML instance, and that instance had entities declared in the internal DTD subset, FrameMaker+SGML will regenerate those entity declarations in the exported SGML instance.

If the user created a FrameMaker+SGML 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 SGML instance.

For information on export of variables, graphics, equations, books, or book components, see Chapter 17, “Translating Variables and System Variable Elements,” Chapter 15, “Translating Graphics and Equations,” or Chapter 19, “Processing Multiple Files as Books.”

Generating processing instructions on export

By default, FrameMaker+SGML creates processing instructions under the following circumstances:

- When the exported SGML turns out to be invalid, the software generates a `<?Fm: Validation Off>` PI. When importing an SGML instance with this PI in its DTD subset, FrameMaker+SGML imports the complete file whether it is valid or not. Note that this PI is only reliable for SGML files that were exported by FrameMaker+SGML.
- 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. For information on export of books and book components, see Chapter 19, “Processing Multiple Files as Books.”
- When exporting a document containing non-element markers other than `SGML PI` or `SGML Entity Reference` markers, 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+SGML writes `<?FM: MARKER [Index] lionfish, ocellated>`.
- When exporting a marker of type SGML PI, the software generates a `<?text>` PI where *`text`* is the marker text.

Exporting special marker types as processing instructions and entity references

FrameMaker+SGML uses special marker types to store information on processing instructions, PI entities, and external data entities.

This marker type	Defines
`SGML PI`	a processing instruction
`SGML Entity Reference`	an entity reference, a PI entity, or a reference to a PI entity

Where *`text`* is the marker text, the software exports `SGML PI` markers as processing instructions of this form:

```
<?text>
```

and it exports `SGML 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+SGML uses to store PI, entity reference, or PI entity information.

Exporting text insets

FrameMaker+SGML text insets generally export as external text entities. On import, if the external entity is declared in the internal DTD subset of the SGML instance, FrameMaker+SGML 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+SGML exports it as
A text or SGML file, or an SGML fragment	The SGML application's DTD has an entity definition that maps to the referenced file	An entity reference to the entity declaration in the SGML application's DTD.
A text or SGML file, or an SGML fragment	The Entity Declarations reference page maps an entity declaration to the referenced file	An entity declaration in the internal DTD subset for the SGML instance, using the original entity name. It also writes a reference to that entity declaration.
A text or SGML file, or an SGML 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 SGML application's DTD	An entity declaration in the internal DTD subset for the SGML 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 SGML instance.

On import to FrameMaker+SGML

By default, FrameMaker+SGML 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+SGML document object that corresponds with the entity reference in SGML.
- If the entity was declared in the internal DTD subset of the SGML instance, the software stores that information on the Entity Declarations reference page of the resulting FrameMaker+SGML document.
- If the entity was declared in the external DTD subset, the software assumes it will be declared in your SGML application's DTD. No entity declaration information is stored with the FrameMaker+SGML document.

In this way, the software can import and export entity references correctly. You can also modify your DTD or use SGML read/write rules to have FrameMaker+SGML treat some entities differently.

The rest of this section describes how FrameMaker+SGML imports various entities and unknown processing instructions in the absence of rules or clients.

Internal SGML text entities

Internal SGML text entities are SGML text entities whose replacement text is determined solely by information in their declarations. The parser recognizes any markup within the entity text as such. The following are examples of declarations of internal SGML text entities:

```
<!ENTITY product "FrameMaker+SGML">

<!ENTITY blist "<list type=bulleted>">

<!ENTITY blist STARTTAG "list type=bulleted">
```

On import, FrameMaker+SGML 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+SGML translates it to a variable of the same name. If the document template does not define a variable of that name, FrameMaker+SGML uses the entity text as the variable's definition.

If the template already contains a variable with that name, FrameMaker+SGML uses the template's variable definition and writes a message to the log file warning of potential mismatches. No error message is written to the error log file in this case. The file imports without complaining but it does use the template's definition for the imported variable. However, in all cases the information on the Entity Declarations reference page contains the full text from the SGML entity declaration.

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, the first 240 characters are imported and used as the variable's definition; it is not imported 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 thelimit". If the entity text contains any markup characters (as in the last two examples above), FrameMaker+SGML imports the entity as plain text and discards all information about the entity reference. The last two examples generate

For information on SGML log files, see "Log files" on page 58.

Internal character data (CDATA) entities

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+SGML always translates these entities to variables of the same name, with the variable text determined by the CDATA string. If the template already contains a variable with that name, FrameMaker+SGML uses the template's variable definition and writes a message to the log file warning of potential mismatches. No error message is written to the log file in this case. The file imports without complaining but it does use the template's definition for the imported variable. If the length of the replacement text exceeds the maximum length supported by a variable, FrameMaker+SGML imports the entity as plain text and discards all information about the entity reference.

In the above example, the variable name is tag, and the variable text is <TAG>.

Internal special character data (SDATA) entities

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+SGML. In this case, the intent is to use the FrameMaker+SGML system variable Current Date (Long).

Since FrameMaker+SGML 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+SGML 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+SGML, you could use this declaration in your DTD:

```
<!ENTITY oquote SDATA "FM char: \xd2">
```

With this entity declaration, when FrameMaker+SGML 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+SGML document, it generates an oquote entity reference. In

either case, you do not need rules to accomplish the translation. For information on how FrameMaker+SGML interprets each of these parameter literals, see the procedures in "Modifications to the default translation" on page 234.

If FrameMaker+SGML 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 SGML text entities. If the template already contains a variable with that name, FrameMaker+SGML uses the template's variable definition.

If FrameMaker+SGML 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+SGML what type of entity to create on export.

External data entities

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>
```

Except for external text entities (see "External SGML text entities," next), FrameMaker+SGML translates a direct reference to an external entity as a marker of type `SGML 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+SGML graphic element, the software would create an anchored frame that imports `door.cgm` by reference. For more about importing graphic elements, see Chapter 15, "Translating Graphics and Equations."

If the entity declaration was made in the SGML document's internal DTD subset, the software saves the entity information on the Entity Declarations reference page of the resulting `document` document. This will be used to reconstruct the entity declaration on export.

You can use a read/write rule to drop references to external data entities on import. 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+SGML graphic. For more information about using entity references in attributes to import graphic files, see Chapter 15, "Translating Graphics and Equations."

External SGML text entities

External SGML 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+SGML 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 SGML 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+SGML translates some external SGML text entities as book components. On export, it translates these text insets and book components back into external SGML text entities. For more information on working with books, see Chapter 19, “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+SGML expands these entities and imports their replacement text. On export, FrameMaker+SGML exports only the entity replacement text.

Subdocument (SUBDOC) entities

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+SGML 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

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+SGML (to represent books and book components), FrameMaker+SGML stores them in markers of type SGML Entity Reference by default. In the above example, the marker text would be:

```
break
```

Processing instructions

As stated at the beginning of this chapter, processing instructions in SGML provide a way to perform system-specific actions on an SGML document. By default, FrameMaker+SGML recognizes a small set of processing instructions. Those it does not recognize in a document instance it stores in a marker of type SGML 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+SGML creates an SGML PI marker with this marker text:

```
mypi
```

In addition to processing instructions for books and book components, FrameMaker+SGML 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+SGML 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.

Limitations to Writing Processing Instructions

When importing an SGML document, be sure not to have processing instructions after the end tag for the highest-level element in the SGML document. For example, even though it is valid SGML, the following document will not import to FrameMaker+SGML:

```
<!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>
```

On import, FrameMaker+SGML will abort the import process when it encounters the `<?pi at end>` processing instruction because it comes after the `</example>` end tag.

Modifications to the default translation

You can handle internal SDATA entities in a variety of ways in FrameMaker+SGML, depending on the information they represent. In some cases, you can convert an SDATA entity to a FrameMaker+SGML variable or to particular characters using selected character formats. In other situations, you might convert the entity to one or more FrameMaker+SGML objects. For example, you may have an entity for a company's logo, which you represent in FrameMaker+SGML 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. 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+SGML document.

You can declare SDATA entities with FrameMaker+SGML parameter literal text. A parameter literal is a way to declare in the entity itself how FrameMaker+SGML will translate the entity. If, for a single entity, you have both an entity declaration with a FrameMaker+SGML parameter literal and a rule that applies to the entity, the entity declaration takes precedence. For an example of parameter literal text, see "Translating SDATA entities as FrameMaker+SGML variables" on page 236.

For a summary of read/write rules relevant to translating entities, see "Entities" on page 325.

Specifying the location of entity declarations

Your SGML application may include files of entity declarations to use in addition to declarations in a particular SGML document. You need to tell FrameMaker+SGML where it can find these declarations. You do so in the application definition. For more information, see “Application definition file” on page 42.

Renaming entities that become variables

FrameMaker+SGML 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 SGML entity name and *var* is a FrameMaker+SGML variable. For example, if you have an entity `date`, the FrameMaker+SGML 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 “entity” on page 349 and “is fm variable” on page 414.

Translating SDATA entity references on import and export

Regardless of precisely what you want an SGML `SDATA` entity to correspond to in FrameMaker+SGML, you have two different approaches for defining the correspondence. One approach is to specify a string FrameMaker+SGML understands as the parameter literal of the `SDATA` entity declaration in the SGML DTD. The other approach is to use the `entity` read/write rule. FrameMaker+SGML parameter literals and read/write rules have the same effect, but read/write rules don’t require you to create a DTD specific to FrameMaker+SGML. 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`	237
Variables	`"fm variable:"`	`entity "ename" is fm variable`	236
Text insets	`"fm text inset:"`	`entity "ename" is fm text inset "fname"`	238
Other entities	`"fm ref:"`	`entity "ename" is fm reference element`	240

If you use the `entity` rule, you can choose to have it effective only when the software reads an SGML document, not when it writes a FrameMaker+SGML document as SGML. To do so, you make the `entity` rule a subrule of a highest-level `reader` rule. For example,

your SGML document might use a `period` entity for entering some instances of the period (.) character in your SGML document. 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 "entity" on page 349, "is fm char" on page 387, and "reader" on page 427.

Translating SDATA entities as FrameMaker+SGML variables

You can equate an `SDATA` entity with a FrameMaker+SGML 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 SGML entity and *var* is the desired FrameMaker+SGML variable.

For example, to have an entity named `date` correspond to the FrameMaker+SGML 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+SGML 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 SDATA entity references on import and export," (the previous section).

For information on these rules, see "entity" on page 349 and "is fm variable" on page 414. For more information on the treatment of FrameMaker+SGML variables, see Chapter 17, "Translating Variables and System Variable Elements."

Translating SDATA entities as special characters in FrameMaker+SGML

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+SGML, 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 200). 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+SGML character tag.

Without the `in` clause, this parameter literal or rule causes FrameMaker+SGML 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+SGML 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+SGML encounters a reference to the reg entity when importing an SGML document, it replaces the reference with ® (assuming your FrameMaker+SGML template defines the Trademark character format appropriately). When exporting a document, if FrameMaker+SGML 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+SGML document. For information on how to create entity palettes, see “Facilitating entry of special characters that translate as SGML entities” on page 244.

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+SGML includes copies of these entity sets and provides rules to handle them for your application. For information on how FrameMaker+SGML supports ISO public entities, see Appendix F, “ISO Public Entities.”

You can choose to have this translation only happen on import of an SGML document. For more information, see “Translating SDATA entity references on import and export” on page 235.

For information on the rules used in these examples, see “entity” on page 349 and “is fm char” on page 387. For information on the FrameMaker+SGML character set, see Appendix E, “Character Set Mapping,” in this manual, and see the FrameMaker user’s manual.

Translating SDATA entities as FrameMaker+SGML text insets

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+SGML can filter automatically. Typically, such files will be FrameMaker+SGML documents or text files,

although you can use files created by other word processing systems. If the source file is a FrameMaker+SGML 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+SGML generates a structured document with a highest-level element named `SGMLFragment`. When you import such a structured flow as a text inset, FrameMaker+SGML automatically unwraps all the children of the `SGMLFragment` element so the document that contains the text inset can be valid.

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+SGML 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+SGML 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+SGML 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 243.

You can choose to have this translation only happen on import of an SGML document. For more information, see "Translating SDATA entity references on import and export" on page 235.

For information on these rules, see "entity" on page 349 and "is fm text inset" on page 411.

Translating SDATA entities as FrameMaker+SGML reference elements

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+SGML 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+SGML reference page in the associated FrameMaker+SGML 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+SGML encounters references to the specified entity while importing an SGML document, it copies the appropriate element from its reference page in the associated FrameMaker+SGML 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 SDATA entity references on import and export” on page 235.

For information on these rules, see “entity” on page 349 and “is fm reference element” on page 402.

Translating external SGML text entities as text insets

External text entities specify a reference to an external file that is either SGML or text. By default, on import FrameMaker+SGML translates the entity as a text inset. Note that the specified file cannot be a FrameMaker+SGML document because such an entity declaration would not be valid SGML.

If the referenced entity declaration was made in the DTD subset of the SGML document, FrameMaker+SGML 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 SGML.

Unless the source file is SGML, insertion of a FrameMaker+SGML 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+SGML 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 SGML fragments into a document as text insets. When exporting to SGML, FrameMaker+SGML might not have entity declarations for such user-created text insets. In the absence of entity declarations, FrameMaker+SGML generates entity declarations with entity names of `ti1`, `ti2`, etc.

Translating internal SGML text entities as text insets

By default, on import FrameMaker+SGML translates internal SGML text entities either as variables or as plain text. You can choose to have it instead import an SGML text entity as a FrameMaker+SGML text inset. You may do so if the FrameMaker+SGML cannot import the entity as a variable. For example, the entity might contain too many characters, or it might contain SGML elements.

To import an SGML text entity as a FrameMaker+SGML 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+SGML document, and *flowname* is a flow in the FrameMaker+SGML document. This rule translates the SGML 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+SGML 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+SGML can filter automatically. Typically, such files will be FrameMaker+SGML documents or text files, although you can use files created by other word processing systems. Also, a FrameMaker+SGML text inset is always the entire contents of a flow or file.

Insertion of a FrameMaker+SGML 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. Also, FrameMaker+SGML text inset in a document 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+SGML generates a structured document with a highest-level element named `SGMLFragment`. When you import such a structured flow as a text inset, FrameMaker+SGML automatically unwraps all the children of the `SGMLFragment` element so the document that contains the text inset can be valid.

By default, the structure of a text inset is retained and the text reformatted to match the FrameMaker+SGML 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 an SGML document. For more information, see “Translating SDATA entity references on import and export” on page 235.

For information on these rules, see “entity” on page 349 and “is fm text inset” on page 411.

Changing the structure and formatting of a text inset on import

As described in “Translating SDATA entities as FrameMaker+SGML text insets” on page 238 and “Translating internal SGML text entities as text insets,” (the previous section), you can choose to import some entities as FrameMaker+SGML text insets. When you do so, the inset is to a FrameMaker+SGML 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 SGML 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 SGML, these rules will have no effect. For example, for insets to FrameMaker+SGML 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 "entity" on page 349, "is fm text inset" on page 411, "reformat as plain text" on page 429, "reformat using target document catalogs" on page 429, and "retain source document formatting" on page 430.

Discarding external data entity references

By default, FrameMaker+SGML imports direct references to external data entities as markers of type `SGML 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+SGML 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 "external data entity reference" on page 362 and "drop" on page 342.

Translating ISO public entities

For information on how FrameMaker+SGML supports ISO public entities, see Appendix F, "ISO Public Entities."

Facilitating entry of special characters that translate as SGML entities

Your end users do not need to know the details of the entity names or definitions used to represent special characters within a FrameMaker+SGML document. They can simply insert any special character into the document and have FrameMaker+SGML 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+SGML 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+SGML requesting the insertion of an entity with the given name. FrameMaker+SGML 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+SGML 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 an SGML 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 SGML and with a book file in FrameMaker+SGML. For information on this use of general entities, see Chapter 19, “Processing Multiple Files as Books.”

Discarding unknown processing instructions

By default, FrameMaker+SGML imports processing instructions it does not understand as markers of type `SGML PI`. (In previous versions of FrameMaker+SGML, 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 “processing instruction” on page 425 and “drop” on page 342.

Using entities for storing graphics or equations

In SGML, 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 “Translating SDATA entities as FrameMaker+SGML reference elements” on page 240 and Chapter 15, “Translating Graphics and Equations.”

14 Translating Tables

Many documents require the use of tables to organize information into cells in rows and columns. FrameMaker+SGML has a complete tool for creating tables and has a specialized element structure for representing them. SGML 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+SGML directly supports.

DTDs that do not use the CALS table model can have arbitrary representations for tables. To support arbitrary table models, FrameMaker+SGML needs you to provide information in the form of read/write rules.

FrameMaker+SGML supports the CALS table model in the sense that you can import 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+SGML 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+SGML interprets the CALS table model: what it does by default when reading an SGML 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. In the outline below, click a topic to go to its page.

How FrameMaker+SGML translates tables by default:

- “On import to FrameMaker+SGML” on page 248
- “On export to SGML” on page 251

Some ways you can change the default translation:

- “Formatting properties for tables” on page 252
- “Identifying and renaming table parts” on page 256
- “Representing FrameMaker+SGML table properties as SGML attributes” on page 257
- “Representing FrameMaker+SGML table properties implicitly in SGML” on page 258
- “Adding format rules that use CALS attributes (CALS only)” on page 259
- “Working with colspecs and spanspecs (CALS only)” on page 260
- “Specifying which part of a table a row or cell occurs in” on page 260

- "Specifying which column a table cell occurs in" on page 261
- "Omitting explicit representation of table parts" on page 262
- "Creating parts of a table even when those parts have no content" on page 264
- "Specifying the ruling style for a table" on page 266
- "Exporting table widths proportionally" on page 267
- "Creating vertical straddles" on page 267
- "Using a table to format an element as a boxed set of paragraphs" on page 270
- "Creating tables inside other tables" on page 272
- "Rotating tables on the page" on page 272

Default translation

FrameMaker+SGML 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+SGML elements to represent tables, see Chapter 6, "Structure Rules for Containers, Tables, and Footnotes." For a description of the element and attribute structure of the CALS table model, see Appendix B, "The CALS Table Model."

On import to FrameMaker+SGML

If your DTD does not use the CALS table model and you create an EDD from the DTD or read an SGML document into FrameMaker+SGML, 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 SGML attributes as attributes in FrameMaker+SGML. It does not retain others as attributes, but uses their values to format the table when reading an SGML document. 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 rules that identify the element and attribute structure of those tables. Appendix C, "SGML Read/Write Rules for CALS 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+SGML 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+SGML table requires the table body to precede the table footing. When importing a `tgroup` element definition, FrameMaker+SGML 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+SGML 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+SGML 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, the SGML tables will not import into your FrameMaker+SGML document.

Also, FrameMaker+SGML does not create elements corresponding to the `colspec` and `spanspec` elements. These elements exist only for their attributes. When you open an SGML 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 250.

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+SGML 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+SGML 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 SGML 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+SGML element of type table and to unwrap the `tgroup` element. If your SGML application includes a FrameMaker+SGML template, the software accommodates this behavior without read/write rules.

That is, if the definition of the `Table` element in your FrameMaker+SGML template is as follows:

Element (Table): Table
General rule: Title?, Thead?, Tbody, Tfoot?

then when FrameMaker+SGML opens an SGML 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+SGML 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+SGML 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+SGML document from an SGML document, the software applies all the format rules in the corresponding EDD. If your SGML documents use the CALS attributes that remain attributes in FrameMaker+SGML, 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 SGML document. The formatting supplied by CALS attributes that correspond only to formatting properties in FrameMaker+SGML (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 259.

How colspec and spanspec elements translate

As stated earlier, the `colspec` and `spanspec` elements do not appear in the final FrameMaker+SGML table structure. These elements exist in SGML 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+SGML 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+SGML 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+SGML processes any attribute of a `colspec` or `spanspec` element, you refer to the attribute as a formatting property.

On export to SGML

By default, if your EDD does not use the CALS table model, FrameMaker+SGML performs no special changes when writing the table and table part elements as elements in SGML. It writes only the element and attribute structure visible in the document. However, if your EDD or FrameMaker+SGML 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 a DTD:

Element (Container): Table
General rule: Title?, Tgroup+

Element (Table): Tgroup
General rule: Thead, Tbody, Tfoot

FrameMaker+SGML 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, Tfoot

the software creates the same element definitions as before. That is, it automatically creates the required SGML `tgroup` element, even though that element was not present in the FrameMaker+SGML 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 SGML, FrameMaker+SGML 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 SGML and FrameMaker+SGML 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 "Tables" on page 329.

Formatting properties for tables

Some properties of FrameMaker+SGML table formatting are not explicit in the element and attribute structure of the table. By default, the software maps these properties to attributes in SGML and gives names to them so that you can refer to them in rules. For example the number of columns in a table becomes the `columns` attribute.

There are many formatting properties associated with FrameMaker+SGML tables that you might want to reference in your read/write rules. These properties are for describing general table properties, 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 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+SGML 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)

Unit	Abbreviation
point	pt

In FrameMaker+SGML 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 SGML documents to FrameMaker+SGML and your SGML table declaration does not include an attribute that corresponds to this table property, you *must* use the `fm property value is` rule to set a value for it.

`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+SGML 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+SGML 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+SGML 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 SGML, 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 “Attribute structure” on page 465.

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. Other elements must use attributes to refer to this information.

FrameMaker+SGML 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+SGML 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 259.

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+SGML.

FrameMaker+SGML 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+SGML and SGML.

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 SGML generic identifier and *fmtag* is a FrameMaker+SGML 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+SGML 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+SGML handles this, see "Omitting explicit representation of table parts" on page 262.

For information on these rules, see "element" on page 345, "is fm table element" on page 408, and "is fm table part element" on page 409.

Representing FrameMaker+SGML table properties as SGML attributes

If you use the CALS table model, FrameMaker+SGML automatically represents some table properties as attributes by default. If you use another table model, FrameMaker+SGML 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 SGML 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 252. For information on the rules used in these examples, see "attribute" on page 335, "element" on page 345, "is fm table element" on page 408, and "is fm property" on page 396.

Representing FrameMaker+SGML table properties implicitly in SGML

A table formatting property in FrameMaker+SGML may always have the same value when applied to a particular SGML element representing tables in your application. If you don't have an SGML attribute for this property so that you can assume that the software will appropriately format the table on the basis of the element, 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 an SGML 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+SGML 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 SGML table declarations do not include an attribute that corresponds to the `columns` property and you plan to open SGML document in FrameMaker+SGML, 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 "element" on page 345, "is fm table element" on page 408, and "fm property" on page 370.

Adding format rules that use CALS attributes (CALS only)

Four attributes of the CALS table model remain attributes when an SGML document is read into FrameMaker+SGML. These attributes, `align`, `valign`, `char`, and `charoff`, provide information on the formatting of text within table cells. In FrameMaker+SGML, this formatting is controlled by format rules in the EDD. Keeping the attributes in the FrameMaker+SGML 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 7, “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 an SGML generic identifier.

As usual with `colspec` and `spanspec` elements, the named SGML element does not become an element in FrameMaker+SGML when you use these rules. Rather, its attributes are used in the creation of the table element.

For more information on these rules, see “element” on page 345, “is fm colspec” on page 389, and “is fm spanspec” on page 406.

Specifying which part of a table a row or cell occurs in

Your SGML 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 an SGML 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 262.

For more information on these rules, see “element” on page 345, “is fm table part element” on page 409, and “fm property” on page 370.

Specifying which column a table cell occurs in

Your SGML 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+SGML 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+SGML 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 an SGML generic identifier and *n* is an integer indicating the table column. Table columns are numbered starting with 1.

If you tell FrameMaker+SGML to put a table cell element in a particular column and there is already content in that column, FrameMaker+SGML 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+SGML 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+SGML 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+SGML 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 an SGML 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 264.

For more information on these rules, see “element” on page 345, “is fm table part element” on page 409, “fm property” on page 370, and “start new row” on page 433.

Omitting explicit representation of table parts

In SGML, 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+SGML, though, elements must be table (and table part) elements if they are to be formatted as tables (and table parts). To facilitate formatting SGML element structures as tables even when those elements do not reflect table structure, however, FrameMaker+SGML will create missing pieces of table structure for you.

For example, assume you have a table of terms and their definitions. In SGML, 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+SGML 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 SGML 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 SGML 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 SGML structure becomes the following table in a FrameMaker+SGML document:

Term	Definition
Mouse	A small animal
Cat	A bigger animal
Elephant	An even bigger animal

When this FrameMaker+SGML table is written as SGML, the `fm element` rules specify that the intervening levels of table structure are not written as SGML. Consequently, the resultant SGML 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 "element" on page 345, "fm element" on page 367, "is fm table element" on page 408, "is fm table part element" on page 409, "fm property" on page 370, and "unwrap" on page 436.

Creating parts of a table even when those parts have no content

When FrameMaker+SGML creates a table while importing an SGML 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 SGML table instance 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+SGML 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 SGML representation assumes that the software will put in the appropriate labels. In this case, the SGML 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 SGML 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 “element” on page 345, “fm element” on page 367, “is fm table element” on page 408, “is fm table part element” on page 409, “fm property” on page 370, “reader” on page 427, “insert table part element” on page 381, “drop” on page 342, and “unwrap” on page 436.

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+SGML 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 "reader" on page 427 and "table ruling style is" on page 435 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+SGML 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+SGML 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 "writer" on page 442, "use proportional widths" on page 438, and "proportional width resolution is" on page 426.

Creating vertical straddles

FrameMaker+SGML 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 SGML 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 SGML 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 260, “Omitting explicit representation of table parts” on page 262, and “Creating parts of a table even when those parts have no content” on page 264.

For more information on these rules, see

- “element” on page 345
- “fm element” on page 367
- “is fm table element” on page 408
- “is fm table part element” on page 409
- “fm property” on page 370
- “reader” on page 427
- “insert table part element” on page 381
- “end vertical straddle” on page 348
- “start vertical straddle” on page 434
- “start new row” on page 433
- “drop” on page 342
- “unwrap” on page 436

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+SGML, 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:

> This boxed paragraph is implemented as a one-cell table.

Using read/write rules, to you can translate a single SGML element that needs to be formatted as one or more boxed paragraphs into a one-cell table in FrameMaker+SGML. 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+SGML creates the `Note` table element when it encounters a `note` element while importing an SGML 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 SGML 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 “element” on page 345, “fm element” on page 367, “is fm table element” on page 408, “fm property” on page 370, and “unwrap” on page 436. For information on creating table formats, see the FrameMaker user’s manual.

Creating tables inside other tables

The FrameMaker+SGML 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 SGML, you’ll need to write an SGML API client to do so.

For information on writing SGML API clients, see the*SGML API Programmer’s Guide*.

Rotating tables on the page

In FrameMaker+SGML, you cannot directly specify that an entire table be rotated on the page. If you need to rotate tables, the *FrameMaker User Guide* 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 SGML. Since the second method places the table in a separate text flow, you must write an SGML API client to perform export to SGML if you use it.

For information on writing SGML API clients, see the*SGML API Programmer’s Guide*.

15 Translating Graphics and Equations

FrameMaker+SGML 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+SGML document. SGML, on the other hand, does not standardize the representation of either graphics or equations; each DTD can treat them differently.

FrameMaker+SGML has a default set of element and attribute definition list declarations for representing graphics and equations as elements. This structure represents an equation as a single graphic in an anchored frame. It does not represent the equation's internal structure. You can use SGML read/write rules to support variations of the default representation, whether you start with an EDD or a DTD, although you cannot use rules to describe the internal structure of equations. Alternatively, you can write an SGML API client to support a completely different model for equations.

In this chapter

This chapter describes how FrameMaker+SGML translates graphics and equations and how you can change that translation. In the outline below, click a topic to go to its page.

How FrameMaker+SGML translates graphics and equations by default:

- "On export to SGML" on page 276
- "On import to FrameMaker+SGML" on page 282

Some ways you can change the default translation:

- "Identifying and renaming graphic and equation elements" on page 284
- "Exporting graphic and equation elements" on page 285
- "Representing the internal structure of equations" on page 286
- "Renaming SGML attributes that correspond to graphic properties" on page 286
- "Omitting representation of graphic properties in SGML" on page 288
- "Omitting optional elements and attributes from the default DTD declarations" on page 288
- "Specifying the data content notation on export" on page 289
- "Changing the name of the graphic file on export" on page 290
- "Changing the file format of the graphic file on export" on page 291
- "Creating graphic files on export" on page 282
- "Specifying the entity name on export" on page 294

• "Changing how FrameMaker+SGML writes out the size of a graphic" on page 295

Default translation

FrameMaker+SGML 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+SGML, 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+SGML treats the equation as a single graphic object inside an anchored frame and exports this anchored frame to SGML.

Supported graphic file formats

FrameMaker+SGML 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 SGML or from FrameMaker+SGML.

The available export graphic formats for FrameMaker+SGML 6.0 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 (Windows)
EPSB	Encapsulated PostScript Binary (Windows)
EPSD	Encapsulated PostScript with Desktop Control Separations (DCS)
EPSF	Encapsulated PostScript (Macintosh)
EPSI	Encapsulated PostScript Interchange
FRMI	FrameImage
FRMV	FrameVector
G4IM/GP4	CCITT Group 4 to Image
GEM	GEM file (Windows)
GIF	Graphics Interchange Format (Compuserve)

Code	Meaning
HPGL	Hewlett-Packard Graphics Language
IGES	Initial Graphics Exchange Specification (CAD files)
IMG4	Image to CCITT Group 4 (UNIX)
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
TIFF	Tag Image File Format
WMF	Windows Metafile
XBM	X BitMap (Unix only)
XWD	X Windows System Window Dump file

General import and export of graphic elements

SGML 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+SGML, or a variant of that structure.

Even though SGML does not have a standard representation for graphics, there are some commonly used frameworks and FrameMaker+SGML 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 an SGML API client.

Basically, FrameMaker+SGML assumes that in SGML 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+SGML anchored frame in which the graphic sits inside a FrameMaker+SGML document.

If the graphic file is specified in an entity declaration, the entity name is always kept with the graphic inset in the FrameMaker+SGML 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 SGML instance, FrameMaker+SGML stores the information so it can recreate the entity declaration on export. For information on how FrameMaker+SGML saves the entity definition, see "Importing graphic entities" on page 283. For information on how FrameMaker+SGML exports entity declarations, see "Exporting entity declarations" on page 281. For information on how FrameMaker+SGML exports graphic files, see "Creating graphic files on export" on page 282.

An external data entity has an associated notation, designed to tell the SGML application how to render the information in the entity. When the software reads an SGML document, rather than storing this information in attributes or variables in FrameMaker+SGML, it stores the information directly in the graphic's associated anchored frame. This results in fewer attributes on the FrameMaker+SGML element than were on the SGML element. When exporting a document to SGML, the software recreates this information in the attributes and entities it writes.

You can use rules to modify some of what FrameMaker+SGML 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 SGML

Some properties of FrameMaker+SGML graphics and equations are not explicit in their element and attribute structure and the software translates these properties as attributes in SGML. 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+SGML will write out a new graphic file when exporting a document to SGML. For more information, see "Creating graphic files on export" on page 282

Text of default graphics and equations declarations

The default DTD declarations for graphics and equations are described in detail in the following sections. In summary, the default declarations are:

```
<!--SGML graphic and equation elements-->
<!ELEMENT list_of_graphic_and_equation_elements - O EMPTY>

<!--Attributes for equations-->
<!ATTLIST equation_element
   entity   ENTITY  #IMPLIED
   file     CDATA   #IMPLIED
```

```
   align    NAME    #IMPLIED --anchored frame alignment on page--
   angle    CDATA   #IMPLIED --anchored frame angle in degrees--
   bloffset CDATA   #IMPLIED --anchored frame baseline offset--
   cropped  NUMBER  #IMPLIED --nonzero for cropped--
   float    NUMBER  #IMPLIED --nonzero for floating--
   height   CDATA   #IMPLIED --anchored frame height--
   nsoffset CDATA   #IMPLIED --anchored frame near-side offset--
   position NAME    #IMPLIED --anchored frame position--
   width    CDATA   #IMPLIED --anchored frame width--
attribute_declarations_specific_to_this_equation_element
>

<!--Attributes for graphics-->
<!ATTLIST graphic_element
   entity   ENTITY  #IMPLIED
   file     CDATA   #IMPLIED

   align    NAME    #IMPLIED --anchored frame alignment on page--
   angle    CDATA   #IMPLIED --anchored frame angle in degrees--
   bloffset CDATA   #IMPLIED --anchored frame baseline offset--
   cropped  NUMBER  #IMPLIED --nonzero for cropped--
   float    NUMBER  #IMPLIED --nonzero for floating--
   height   CDATA   #IMPLIED --anchored frame height--
   nsoffset CDATA   #IMPLIED --anchored frame near-side offset--
   position NAME    #IMPLIED --anchored frame position--
   width    CDATA   #IMPLIED --anchored frame width--

   dpi      NUMBER  #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 NUMBER  #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.

When FrameMaker+SGML 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 an SGML API client to change these behaviors. For information about exporting graphic or equation files, see "Creating graphic files on export" on page 282. 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+SGML. 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 SGML element that corresponds to a graphic or equation. When exporting a FrameMaker+SGML 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 SGML element declaration.

If the SGML element has an `entity` attribute defined, then on export FrameMaker+SGML writes a value for the `entity` attribute in SGML. If there is no corresponding entity in the SGML application's DTD, it also generates the corresponding entity declaration in the document's internal DTD subset. By default, FrameMaker+SGML doesn't generate a public identifier; you must supply an SGML API client to do so.

If the SGML element has a `file` attribute, but no `entity` attribute, the software writes a value for the `file` attribute. On import, if the 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 SGML 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+SGML 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+SGML 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`

Attribute value	Property value
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 *FrameMaker+SGML User Guide* for information about anchored frames.

Other graphic properties SGML elements corresponding to graphic elements can have the following additional implied attributes:

- 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+SGML 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+SGML document on import from SGML. 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.
- 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 *FrameMaker+SGML User Guide* for information about importing graphics.

Exporting entity declarations

When importing an SGML instance, use read/write rules to import SGML 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 SGML instance, that information is stored on the Entity Declarations reference page of the FrameMaker+SGML document. On export, the software will use this information to recreate the entity declarations in the resulting SGML instance. For more about saving entity declaration information in a FrameMaker+SGML document, see "Importing graphic entities" on page 283.

On export to SGML, the software ensures the appropriate entities are in the SGML instance as follows. Unless otherwise specified, this table assumes the SGML graphic element's declaration includes an `entity` attribute:

If:	FrameMaker+SGML:
The graphic filename and entity name match an entity declaration in the SGML application's DTD...	Uses the matching entity name as the value of the SGML graphic element's `entity` attribute.
The graphic filename matches an entity declaration on the Entity Declarations reference page...	Declares the entity in the DTD subset of the resulting SGML instance. It uses the entity name that was stored with the graphic inset for the entity declaration and for the `entity` attribute in the resulting SGML graphic element.
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 SGML instance. It names the entity graphic1, graphic2, etc. It writes the entity name to the `entity` attribute in the resulting SGML 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 SGML instance as it does for the FrameMaker+SGML 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+SGML. 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 SGML 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 SGML graphic element, 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+SGML 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+SGML's graphics tools. By default, the software creates the following SGML entity for it:

```
<!ENTITY graphic1 graph1 SYSTEM "graphic1.cgm" graph1.cgm NDATA
cgm> replace graphic with graph
```

Also, the `entity` attribute of the graphic element has a value of `graphic1` to correspond with the above entity.

On import to FrameMaker+SGML

In the absence of read/write rules, FrameMaker+SGML cannot identify the elements and attributes of an SGML document that correspond to a graphic or equation. Therefore, it translates them as container elements in the EDD. You must supply rules and perhaps an SGML 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 276.

Your SGML document can use either the `entity` attribute or 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+SGML 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 SGML graphic element uses the `entity` attribute to identify the graphic file, then the actual graphic file must be identified in an entity declaration. FrameMaker+SGML will read the SGML, 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 SGML document's internal DTD subset. In that case, FrameMaker+SGML 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 SGML. For graphics 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 281. For information about when the software does and does not use the same graphic file, see "Creating graphic files on export" on page 282.

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+SGML. On import, none of these attributes translate to FrameMaker+SGML 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 SGML attributes specified in the instance 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 SGML instance. If a MIF file doesn't contain an anchored frame and the SGML 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+SGML expects the graphic file to be in one of the formats it supports. For the list of graphic file formats FrameMaker+SGML supports, see "Supported graphic file formats" on page 274.

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+SGML 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 SGML read/write rules for graphics and equations allow you, among other things, to:

- identify which SGML elements correspond to graphic or equation elements
- associate SGML attributes with FrameMaker+SGML 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 "Graphics" on page 327. For a summary of read/write rules relevant to translating equations, see "Equations" on page 326. For a list of the formatting properties associated with translating graphics and equations, see "Text of default graphics and equations declarations" on page 276.

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 SGML elements correspond to graphic or equation elements in FrameMaker+SGML. 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 an SGML 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 SGML element `pict` corresponds to the graphic element `Picture` in FrameMaker+SGML, use this rule:

```
element "pict" is fm graphic element "Picture";
```

For information on the rules used in this example, see "element" on page 345, "is fm equation element" on page 392, and "is fm graphic element" on page 394.

Exporting graphic and equation elements

FrameMaker+SGML 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 an SGML generic identifier, *fmtag* is an optional FrameMaker+SGML 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+SGML 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+SGML graphic tools, there is no corresponding entity. The software will write the graphic file in PICT format, and create a corresponding graphic entity. The SGML 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 282 and "Changing the file format of the graphic file on export" on page 291. For information on these rules, see

- "anchored frame" on page 333
- "facet" on page 364
- "equation" on page 355
- "element" on page 345
- "is fm equation element" on page 392
- "is fm graphic element" on page 394
- "export to file" on page 359
- "writer" on page 442

Representing the internal structure of equations

As mentioned earlier in this chapter, FrameMaker+SGML treats an equation in the same way it does a graphic element. That is, on export to SGML, 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+SGML equation as an element with subelement structure instead, you must write an SGML 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 SGML API clients, see the *SGML API Programmer's Guide*.

Renaming SGML attributes that correspond to graphic properties

FrameMaker+SGML 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 SGML, however, these properties may be represented as attributes. In other words, there are certain SGML attributes that do not translate to FrameMaker+SGML attributes, but instead directly to graphic properties. "Anchored frame properties" on page 278 and "Other graphic properties" on page 280 describe the default names for the SGML attributes corresponding to graphic properties.

You can choose names other than the default ones for these SGML 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 SGML 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 SGML 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+SGML writes out the size of a graphic" on page 295.

For information on the rules used in these examples, see "attribute" on page 335, "element" on page 345, "is fm graphic element" on page 394, and "is fm property" on page 396.

Omitting representation of graphic properties in SGML

Some properties of graphics and equations have no explicit representation in SGML. In such cases, you may want to use a rule to make the information explicit in FrameMaker+SGML. You can use this rule to do so:

```
fm property prop value is "propval";
```

where *prop* is the FrameMaker+SGML property and *propval* is the value to use.

With this rule, the software creates a DTD from an EDD without creating the corresponding SGML attribute. When importing an SGML document, it causes the software to assign a particular value to one of the formatting properties. When exporting a document, it tells FrameMaker+SGML 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 an SGML element `bitmap` treated as a graphic element in FrameMaker+SGML. Furthermore, you know that you never want to make a copy of such an object in the FrameMaker+SGML 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+SGML document to SGML, it does not try to write a value for the `impby` attribute. When importing an SGML document, it automatically sets the property value to `ref`.

For a summary of the FrameMaker+SGML graphic properties, see “Text of default graphics and equations declarations” on page 276. For information on the rules used in this example, see “element” on page 345, “is fm graphic element” on page 394, and “fm property” on page 370.

Omitting optional elements and attributes from the default DTD declarations

The default DTD declarations provided with FrameMaker+SGML 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+SGML's graphic tools, you could remove these declarations:

```
<!ATTLIST graphic_element
   dpi      NUMBER #IMPLIED --ignored if impsize is 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 NUMBER #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+SGML won't produce or expect values for these attributes.

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+SGML facets, it uses CGM as the data content notation. Your SGML 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 an SGML generic identifier, *`graphic_or_equation`* is one of the keywords `graphic` or `equation`; *`fmtag`* is an optional FrameMaker+SGML 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 SGML 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+SGML 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:

- "notation is" on page 420
- "element" on page 345
- "is fm graphic element" on page 394
- "is fm equation element" on page 392
- "anchored frame" on page 333
- "equation" on page 355
- "facet" on page 364
- "writer" on page 442

Changing the name of the graphic file on export

Under certain conditions, when it exports a FrameMaker+SGML document, the software creates a file for each graphic and equation. For more information, see "Creating graphic files on export" on page 282. 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 an SGML generic identifier; *graphic_or_equation* is one of the keywords `graphic` or `equation`; *fmtag* is an optional FrameMaker+SGML 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
`$(entity)`	The value of the corresponding SGML element's `entity` attribute. If the source of the graphic inset wasn't originally an SGML entity, this variable evaluates to a unique name based on the name of the element.
`$(docname)`	The name of the FrameMaker+SGML 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+SGML document to SGML, 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:

- "export to file" on page 359
- "notation is" on page 420
- "attribute" on page 335
- "is fm property" on page 396
- "is fm attribute" on page 385
- "element" on page 345
- "is fm graphic element" on page 394
- "is fm equation element" on page 392
- "anchored frame" on page 333
- "equation" on page 355
- "facet" on page 364
- "writer" on page 442

Changing the file format of the graphic file on export

By default, when it creates a graphic file on export, FrameMaker+SGML 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 282.

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+SGML. For the list of graphic file formats FrameMaker+SGML supports, see "Supported graphic file formats" on page 274.

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 an SGML generic identifier; *`graphic_or_equation`* is one of the keywords `graphic` or `equation`; *`fmtag`* is an optional FrameMaker+SGML 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
`$(entity)`	The value of the corresponding SGML element's `entity` attribute. If the source of the graphic inset wasn't originally an SGML entity, this variable evaluates to a unique name based on the name of the element.
`$(docname)`	The name of the FrameMaker+SGML file, excluding any extension or directory information.

For example, FrameMaker+SGML 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 "export to file" on page 359, "element" on page 345, "is fm graphic element" on page 394, "is fm equation element" on page 392, "anchored frame" on page 333, "equation" on page 355, "facet" on page 364, and "writer" on page 442.

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+SGML 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+SGML document, it would export as follows:

For a graphic element:	On export to SGML the software:
Imported as an SGML 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 an SGML graphic element 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+SGML document by the author...	Writes a graphic file named `graphic1.pic` (`graphic2.pic`, graphic3.pic, etc.). It also creates an SGML graphic element 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 an SGML generic identifier; *graphic_or_equation* is one of the keywords `graphic` or `equation`; *fmtag* is an optional FrameMaker+SGML 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+SGML 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 SGML document's internal DTD subset:

```
<!ENTITY cologo1 SYSTEM cologo1.tif NDATA TIFF>
```

For more information, see "Creating graphic files on export" on page 282. For information on specifying a graphic filename, see "Changing the name of the graphic file on export" on page 290. For more information on these rules, see:

- "entity name is" on page 352
- "element" on page 345
- "is fm graphic element" on page 394
- "is fm equation element" on page 392
- "anchored frame" on page 333

- "equation" on page 355
- "facet" on page 364
- "writer" on page 442

Changing how FrameMaker+SGML 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 are used for `impsize` and the resolution in which sizes are reported. If there is no `specify size` rule, FrameMaker+SGML uses the `dpi` attribute and exports the graphic at a resolution of 1. The general form of this rule is:

```
element "gi" {
   is fm graphic_or_equation element ["fmtag"];
   writer type ["name"]
      specify size in units [with resolution res];
}
```

where *`gi`* is an SGML generic identifier; *`graphic_or_equation`* is one of the keywords `graphic` or `equation`; *`fmtag`* is an optional FrameMaker+SGML 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; and *`res`* is a number indicating how many decimal places to use in writing its size.

For example, a resolution of 1 means to write the size as an integer value, and a resolution of 0.01 means to write the size as a real number with 2 decimal places.

FrameMaker+SGML reports the size of the imported graphics object in the indicated units, at the indicated resolution. It calculates a value of `impsize` by determining the width and height of the smallest possible rectangle that can contain the imported object using that resolution and unit type.

For example, assume the graphic is a circle with a diameter of 3.15 centimeters. Given the rule:

```
specify size in cm;
```

FrameMaker+SGML generates the attribute `impsize="4cm 4cm"`.

However, with the same graphic, if the rule is:

```
specify size in cm with resolution 0.1;
```

FrameMaker+SGML generates `impsize="3.2cm 3.2cm"`.

For more information on these rules, see:

- “specify size in” on page 431
- “element” on page 345
- “is fm graphic element” on page 394
- “is fm equation element” on page 392
- “anchored frame” on page 333
- “equation” on page 355
- “facet” on page 364
- “writer” on page 442

16 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 the SGML standard 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+SGML can also use this model for cross-references within a FrameMaker+SGML document.

There are several differences between the FrameMaker+SGML cross-reference mechanism and the customary way of interpreting the related SGML attributes. For information on these differences, see "Cross-references" on page 17.

In this chapter

This chapter starts by describing the default translation provided by FrameMaker+SGML. What it does by default differs depending on whether you start from an EDD or from a DTD. The chapter then describes how to modify the default behavior. In the outline below, click a topic to go to its page.

How FrameMaker+SGML translates cross-references by default:

- "On export to SGML" on page 298
- "On import to FrameMaker+SGML" on page 299

Some ways you can change the default translation:

- "Translating SGML elements as FrameMaker+SGML cross-reference elements" on page 300
- "Renaming the SGML attributes used with cross-references" on page 300
- "Translating FrameMaker+SGML cross-reference elements to text in SGML" on page 301
- "Maintaining attribute values with FrameMaker+SGML" on page 301

Default translation

On both import and export, FrameMaker+SGML assumes that elements having an attribute with a declared value of `ID` and elements with declared content `EMPTY` having an attribute with a declared value of `IDREF` are for cross-references. So, by default, within a single SGML document, FrameMaker+SGML document, or SGML book, FrameMaker+SGML automatically translates cross-reference elements (those that are empty and use an `IDREF` attribute) and element-based cross-reference sources (those that use an `ID` attribute).

FrameMaker+SGML interprets internal and external cross-references in multiple-file SGML documents and FrameMaker+SGML books in special ways:

- When you export a FrameMaker+SGML *book* to SGML, it becomes a single SGML document. Similarly, when you import an SGML *document* to a FrameMaker+SGML book, it becomes multiple FrameMaker+SGML documents. Because of this, cross-references that are external in FrameMaker+SGML may be internal to a single SGML document. For a discussion of the importance of the distinction between internal and external cross-references, see "Cross-references" on page 17.
- When you export a *single* FrameMaker+SGML document to SGML, FrameMaker+SGML treats all cross-references that were external in FrameMaker+SGML as external in SGML. However, if you export an *entire* FrameMaker+SGML book to SGML, treatment of the individual FrameMaker+SGML documents that make up the book is more complicated: cross-references to other documents in the book are considered internal cross-references and only those cross-references to documents not in the book are considered external.

For example, assume your FrameMaker+SGML book, `manual.bk`, contains the files `ch1.doc` and `ch2.doc` and that there is an external cross-reference in `ch1.doc` to `ch2.doc`. When you export `manual.bk` to SGML, it creates one SGML document and the external cross-reference from `ch1.doc` to `ch2.doc` becomes an internal cross-reference in this document. If you import this SGML document back into FrameMaker+SGML (thus again creating multiple files), the cross-reference again becomes an external cross-reference. For information on exporting FrameMaker+SGML books, see Chapter 19, "Processing Multiple Files as Books."

On export to SGML

When creating a DTD from an EDD, FrameMaker+SGML translates `ID` attributes used for cross-reference sources in the same way as any other attributes. FrameMaker+SGML translates a cross-reference element as an SGML empty element of the same name and 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+SGML creates an additional impliable attribute, `format`, with the declared value `CDATA`. When exporting a FrameMaker+SGML 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.

When exporting a document, FrameMaker+SGML 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 SGML 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 SGML.

Whether or not the cross-reference element has an IDREF attribute, if the source of the cross-reference is not an element, then FrameMaker+SGML exports the cross-reference element as text. That is, the fact that an element was present in the FrameMaker+SGML document is lost; instead, the text provided by the cross-reference format appears in the SGML 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+SGML exports the cross-reference as text.

A cross-reference source that isn't an element is exported in the same way as any other FrameMaker+SGML non-element marker. Its corresponding cross-reference exports as text in the way described above.

For more information on FrameMaker+SGML's representation of cross-references, see "Using UniqueID and IDReference attributes" on page 164 of this manual and see the FrameMaker and FrameMaker+SGML user's manuals.

On import to FrameMaker+SGML

When creating an EDD from a DTD, FrameMaker+SGML creates a cross-reference element when it encounters an element declaration with the 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 the 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+SGML UniqueID attribute.

When importing an SGML document, FrameMaker+SGML creates the appropriate attribute in FrameMaker+SGML 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 SGML 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.

When importing an SGML document, FrameMaker+SGML 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 an SGML API client to change this behavior.

Other SGML elements may be intended to represent cross-references. For example, SGML elements that have attributes with the declared values NAMES or IDREFS or that have multiple attributes with the declared value IDREF, and SGML elements that have content

and attributes with the declared value `IDREF`, may be intended for this purpose. You can write an SGML API client 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 an SGML 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 "Cross-references" on page 324. For information on writing SGML API clients, see *SGML API Programmer's Guide*.

Translating SGML elements as FrameMaker+SGML cross-reference elements

You can identify which SGML elements correspond to FrameMaker+SGML cross-reference elements and choose to use the same name or different names for the SGML and FrameMaker+SGML 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 an SGML generic identifier and *`fmtag`*, if specified, indicates the name of the corresponding FrameMaker+SGML element. For example, to indicate that both of the SGML elements `xref` and `link` correspond to FrameMaker+SGML 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 "element" on page 345 and "is fm cross-reference element" on page 389.

Renaming the SGML attributes used with cross-references

(For details, see "On export to SGML" on page 298) FrameMaker+SGML 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 SGML 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 an SGML generic identifier and *attr* is an SGML attribute.

For example, to specify that all relevant SGML elements use the `fmform` attribute to specify a FrameMaker+SGML cross-reference format, but that the SGML `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 an SGML element, you can set that value to be the FrameMaker+SGML 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 "element" on page 345, "attribute" on page 335, "is fm property" on page 396, and "is fm cross-reference element" on page 389.

Translating FrameMaker+SGML cross-reference elements to text in SGML

You may not want your FrameMaker+SGML cross-reference elements to remain elements in the SGML representation. You can choose to translate them to text in SGML instead. To do so, use this rule:

```
fm element "fmtag" unwrap;
```

where *fmtag* is the FrameMaker+SGML element tag for a cross-reference element.

For more information on these rules, see "fm element" on page 367 and "unwrap" on page 436.

Maintaining attribute values with FrameMaker+SGML

FrameMaker+SGML 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 221.

17 Translating Variables and System Variable Elements

You use variables in FrameMaker+SGML 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 SGML, you can use either SGML elements or SGML entities for similar purposes. Some of the material in this chapter is closely related to the handling of SGML entities. For more information on entities, see Chapter 13, "Translating Entities and Processing Instructions."

In this chapter

This chapter starts by describing FrameMaker+SGML'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. In the outline below, click a topic to go to its page.

How FrameMaker+SGML translates variables by default:

- "On export to SGML" on page 304
- "On import to FrameMaker+SGML" on page 305

Some ways you can change the default translation:

- "Renaming or changing the type of entities when translating to variables" on page 305
- "Translating SGML elements as system variable elements" on page 306
- "Translating FrameMaker+SGML system variable elements to text in SGML" on page 307
- "Translating FrameMaker+SGML variables as SDATA entities" on page 307
- "Discarding FrameMaker+SGML variables" on page 307

Default translation

SGML has no unique representation for variables. There are two types of FrameMaker+SGML 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+SGML 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+SGML 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+SGML variables and system variable elements, see the FrameMaker user's manual.

On export to SGML

FrameMaker+SGML translates a system variable element as an SGML empty element. It does not record the definition of the corresponding variable, because that is formatting information for the element.

When writing an SGML document, FrameMaker+SGML 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+SGML creates a new entity definition using the variable name and variable text. If an entity definition with the given name does exist, FrameMaker+SGML writes a message to the log file warning of a potential mismatch.

If the variable name is not a valid SGML name, FrameMaker+SGML uses `fmv1` as a default entity name. If an entity already exists by that name, FrameMaker+SGML increments the counter until an unused name is found, for example, `fmv2`, `fmv3`, and so forth.

FrameMaker+SGML 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+SGML exports the variables as a `CDATA` entity. If the variable text uses the `FmSdata` character format, FrameMaker+SGML exports it as an `SDATA` entity. In the absence of any relevant character format information and markup, FrameMaker+SGML exports the variable as an SGML text entity. If the variable text contains markup, then FrameMaker+SGML exports the variable as a `CDATA` entity.

FrameMaker+SGML 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+SGML uses the string `"FM variable"` as the parameter literal.

FrameMaker+SGML translates a non-element system variable as a reference to an entity with one of the following names:

System variable	Entity
Page Count	`fm.pgcnt`
Current Date (Long)	`fm.ldate`
Current Date (Short)	`fm.sdate`
Creation Date (Long)	`fm.lcdat`

System variable	Entity
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+SGML's treatment of entities, see Chapter 13, "Translating Entities and Processing Instructions."

On import to FrameMaker+SGML

By default, FrameMaker+SGML 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+SGML translates entities by default as variables, see "On import to FrameMaker+SGML" on page 228.

Modifications to the default translation

The following sections describe some possible modifications to the default translation of FrameMaker+SGML variables and system variable elements. You may require different rules or an SGML 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 13, "Translating Entities and Processing Instructions."

For a summary of read/write rules relevant to translating variables and system variable elements, see "Variables" on page 331 and "Entities" on page 325. For information on writing SGML API clients, see the *SGML API Programmer's Guide*.

Renaming or changing the type of entities when translating to variables

By default, FrameMaker+SGML 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 an SGML entity and *`fmvar`* is a FrameMaker+SGML variable. With this rule, FrameMaker+SGML 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 SGML" on page 304.

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+SGML 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+SGML 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>">
```

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+SGML inserts this entity declaration:

```
<!ENTITY startex SDATA "<tag>">
```

If the DTD already has an appropriate entity declaration, FrameMaker+SGML retains that declaration. For that reason, 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+SGML 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 "entity" on page 349 and "is fm variable" on page 414.

Translating SGML elements as system variable elements

You can translate some SGML elements as FrameMaker+SGML system variable elements. To do so, use the following rule:

```
element "gi" is fm system variable element ["fmtag"];
```

where *`gi`* is an SGML generic identifier and the optional argument *`fmtag`* is a FrameMaker+SGML element tag corresponding to a system variable element. For example,

to translate an SGML 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 SGML element `date` as the FrameMaker+SGML 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 "element" on page 345 and "is fm system variable element" on page 407.

Translating FrameMaker+SGML system variable elements to text in SGML

You may not want your FrameMaker+SGML system variable elements to remain elements in the SGML representation. You can choose to translate them to text in SGML instead. To do so, use this rule:

```
fm element "fmtag" unwrap;
```

where *`fmtag`* is the FrameMaker+SGML element tag for a system variable element.

For more information on these rules, see "fm element" on page 367 and "unwrap" on page 436.

Translating FrameMaker+SGML variables as SDATA entities

You can translate FrameMaker+SGML variables as `SDATA` entities by using the `entity` rule or by manipulating parameter literals. For information on how to do so, see "Translating SDATA entities as FrameMaker+SGML variables" on page 236.

Discarding FrameMaker+SGML variables

FrameMaker+SGML always allows you to insert a variable in a document. To modify this behavior, you must use an SGML API client. If you don't want FrameMaker+SGML to export some or all variables to SGML, you can choose to have it discard all variables or particular variables.

To have FrameMaker+SGML 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 "writer" on page 442, "fm variable" on page 372, and "drop" on page 342.

18 Translating Markers

You use markers in FrameMaker+SGML documents to store various kinds of information you don't want visible to the document's audience. Although SGML doesn't have a completely analogous concept, markers frequently correspond to SGML attributes or elements.

FrameMaker+SGML provides a variety of marker types. Two of these marker types have special default translations.

- SGML PI markers store information about some processing instructions, and SGML Entity Reference markers store information about entity references. For information on this use of markers, see Chapter 13, "Translating Entities and Processing Instructions." You can change which marker type FrameMaker+SGML uses to store this information.
- Conditional Text markers indicate conditional text. FrameMaker+SGML ignores conditional text markers on export of SGML. For information on the treatment of conditional text on export, see "Marked sections and conditional text" on page 17.

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.

How FrameMaker+SGML translates markers by default:

- "On export to SGML" on page 310
- "On import to FrameMaker+SGML" on page 310

Some ways you can change the default translation:

- "Translating SGML elements as FrameMaker+SGML marker elements" on page 311
- "Writing SGML marker text as element content instead of as an attribute" on page 311
- "Using SGML attributes and FrameMaker+SGML properties to identify markers" on page 312
- "Discarding non-element FrameMaker+SGML markers" on page 313

Default translation

SGML has no special representation for markers. FrameMaker+SGML allows you to represent markers either as marker elements or as non-element markers.

On export to SGML

FrameMaker+SGML exports a marker element as an SGML 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.

FrameMaker+SGML exports non-element markers, other than those of Type `SGML 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+SGML exports an Index marker with the text “translating, PIs” as:

```
<?FM: MARKER [Index] translating, PIs>
```

If an SGML PI marker has marker text of this form:

```
text
```

then FrameMaker+SGML outputs this processing instruction:

```
<?text>
```

If an SGML Entity Reference marker has marker text of this form:

```
entname
```

where *`entname`* is the entity name, then FrameMaker+SGML outputs this entity reference:

```
&entname;
```

You can use a rule to change the marker type whose text is treated in this manner.

FrameMaker+SGML uses the Cross-Ref marker type to mark the source of a non-element cross-reference. On export to SGML, 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 16, “Translating Cross-References.”

On import to FrameMaker+SGML

In the absence of SGML read/write rules, FrameMaker+SGML cannot identify an SGML 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 a FrameMaker+SGML EDD instead of an SGML DTD, the default DTD translates FrameMaker+SGML marker elements as SGML elements with a declared content of `EMPTY`. If you have an SGML document that uses this DTD, and your application specifies a FrameMaker+SGML template to use on import, then it translates the appropriate SGML elements to marker elements when you import the document to FrameMaker+SGML.

Also, by default FrameMaker+SGML imports some entity references and processing instructions as non-element markers. For information on when this happens, see Chapter 13, “Translating Entities and Processing Instructions.”

Modifications to the default translation

The following sections describe some modifications to the default translation of markers. You may require different rules or an SGML 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 “Markers” on page 328.

Translating SGML elements as FrameMaker+SGML marker elements

You can identify which SGML elements correspond to FrameMaker+SGML marker elements, and use the same name or different names for the FrameMaker+SGML and SGML 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 an SGML generic identifier and *fmtag* is a FrameMaker+SGML element tag. If *fmtag* is specified, it is the name of the corresponding FrameMaker+SGML marker element; otherwise, the FrameMaker+SGML element name is the same as the SGML generic identifier.

For information on the rules used in this example, see “element” on page 345 and “is fm marker element” on page 395.

Writing SGML marker text as element content instead of as an attribute

By default, on export FrameMaker+SGML writes the text of a marker element as the value of an attribute of an empty SGML 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 SGML element. Note that the SGML element cannot be declared as an empty element.

To treat marker text for a specific FrameMaker+SGML marker element as content for the SGML element, use the `marker text` rule as a subrule of the `element` rule:

```
element "gi" marker text is content;
```

where *gi* is an SGML generic identifier.

For information on the rules used in these examples, see “element” on page 345 and “marker text is” on page 418.

Using SGML attributes and FrameMaker+SGML properties to identify markers

When translating FrameMaker+SGML marker elements to SGML elements, by default FrameMaker+SGML uses the SGML attributes `type` and `text`. These correspond, respectively, to the FrameMaker+SGML properties `marker type` and `marker text`. You can choose to have FrameMaker+SGML use different attribute names or to not use one or both of the attributes altogether.

To have FrameMaker+SGML not use one of these attributes, simply remove the attribute from the definition of the corresponding SGML element in the DTD. When exporting a FrameMaker+SGML document, if the corresponding SGML 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 an SGML generic identifier, *`attr`* is an SGML attribute and *`prop`* is one of `marker type` or `marker text`. For example, to have the SGML 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+SGML 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:

- “element” on page 345
- “is fm marker element” on page 395
- “fm property” on page 370
- “drop” on page 342
- “attribute” on page 335
- “is fm property” on page 396

Discarding non-element FrameMaker+SGML markers

FrameMaker+SGML always allows you to insert a non-element marker in a document. Modifying this behavior requires an SGML API client. However, if you don't want FrameMaker+SGML to export some or all non-element markers to SGML, 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+SGML document, use this rule:

```
fm marker ["type1", . . ., "typen"] drop;
```

Each $type_1$ is a marker type. A particular $type_1$ can appear in only one rule. If there is a rule with no $type_1$ 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+SGML 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+SGML would drop all non-element markers, including Index and Hypertext markers.

For information on the rules used in this example, see "fm marker" on page 368, "drop" on page 342, and "is processing instruction" on page 415. For information on writing SGML API clients, see *SGML API Programmer's Guide*, an online manual supplied with the Frame Developer's Kit.

19 Processing Multiple Files as Books

FrameMaker+SGML provides a mechanism for grouping multiple FrameMaker+SGML documents into a single unit called a book. Each document remains a separate, complete FrameMaker+SGML document, but FrameMaker+SGML 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.

SGML doesn't explicitly provide such a mechanism. However, SGML documents are sometimes divided into external SGML text entities so that a large document can be distributed over several files.

This use of SGML text entities as include files is analogous to a FrameMaker+SGML book file, but SGML allows more freedom in specifying the files that correspond to a book:

- A FrameMaker+SGML book can have only one level of documents. An SGML document can have more than one level of nesting.
- A document in a FrameMaker+SGML book can either be unstructured or a single, complete element. An SGML entity can include a partial element.

Because of this difference, if you start with an SGML document that doesn't match the FrameMaker+SGML model for books, the entity structure will not correspond to the book structure. On the other hand, any valid FrameMaker+SGML book can be easily translated to SGML with each book component exported as an SGML text entity.

In this chapter

This chapter starts by describing the default translation for books. The chapter then describes your options for modifying the translation. In the outline below, click a topic to go to its page.

How FrameMaker+SGML translates books and book components by default:

- "On import to FrameMaker+SGML" on page 316
- "On export to SGML" on page 318

Some ways you can change the default translation:

- "Using elements to identify book components on import" on page 319
- "Suppressing the creation of processing instructions for a book on export" on page 321

Default translation

Books in FrameMaker+SGML and their counterparts in SGML 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+SGML books or SGML documents.

On import to FrameMaker+SGML

FrameMaker+SGML does not attempt to automatically subdivide an SGML document into FrameMaker+SGML 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.

To instruct FrameMaker+SGML to generate a book, you can place the following processing instruction before the start-tag of the document element of an SGML 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 SGML document, you indicate the start of a new document in a FrameMaker+SGML book by placing a processing instruction immediately before the start-tag of the element that is the highest-level element in the FrameMaker+SGML document. The processing instruction has the form:

<?FM: document *fname*>

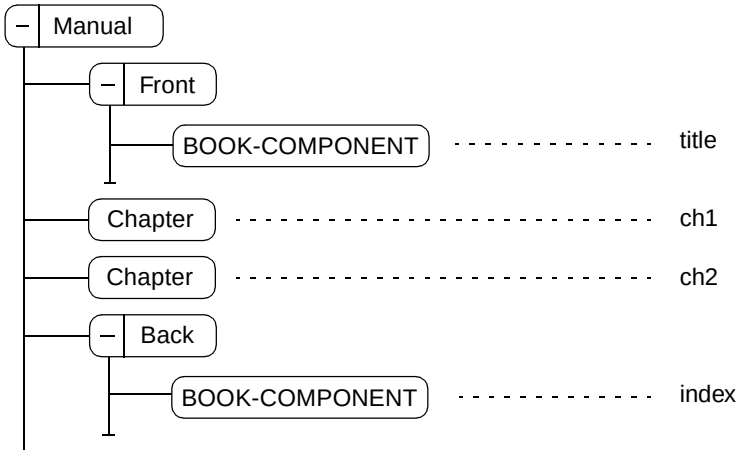
where *fname* is the name of the new document file. If *fname* is a relative pathname, it is relative to the directory for the book file. Any spaces within *fname* must be escaped by a backslash. If *fname* contains the processing instruction close delimiter string (> in the reference concrete syntax), the processing instruction must be entered through an entity. On export, FrameMaker+SGML 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. If you did so, the SGML 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+SGML, each part is in a separate document. In SGML this situation might be represented as:

```
<!DOCTYPE manual SYSTEM "manual.dtd"
<?FM: book>
<manual>
<?FM: document "title">
<front>
. . .
</front>
<?FM: document "ch1">
<chapter>
. . .
</chapter>
<?FM: document "ch2">
<chapter>
. . .
</chapter>
<?FM: document "index">
<back>
. . .
</back>
</manual>
```

In FrameMaker+SGML this SGML structure appears as:



On export to SGML

FrameMaker+SGML treats documents in a FrameMaker+SGML book as external SGML text entities. In addition to the file it creates for the entire book (the SGML 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+SGML also produces the processing instructions that allow it to recreate the original book structure.

In particular, FrameMaker+SGML 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+SGML increments the counter and tries again. For example, if your DTD already defines an entity `bkc1`, FrameMaker+SGML 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+SGML 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+SGML exports a book, it creates a new file for each structured document. Unstructured documents appear as text within an element. You can provide rules to change this.

In general, you can use a different EDD for each document in a FrameMaker+SGML book. However, if you export the book to SGML, the result will be invalid unless the DTD provides a superset of the element types used in all the EDDs.

Modifications to the default translation

FrameMaker+SGML provides a small number of SGML 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 an SGML 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 "drop" on page 342.
- To import or export a document as an empty element, use the `drop content` rule described in "drop content" on page 344.

For a summary of read/write rules relevant to translating books and book components, see "Books" on page 324.

Using elements to identify book components on import

The default method to specify how an SGML document should be broken into components of a FrameMaker+SGML 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+SGML, 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+SGML 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+SGML always creates a book for these elements. If you want FrameMaker+SGML 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+SGML 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+SGML 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+SGML creates one file containing the entire structure; it does not create a separate book file.

For information on these rules, see "reader" on page 427, "generate book" on page 373, and "output book processing instructions" on page 422.

Suppressing the creation of processing instructions for a book on export

By default, FrameMaker+SGML creates processing instructions for books and book components whenever you export a FrameMaker+SGML book to SGML. This happens even if you use the `generate book` rule to cause FrameMaker+SGML not to need the processing instructions on import. To keep FrameMaker+SGML from writing any book or book component processing instructions, use the following rule:

```
writer do not output book processing instructions;
```

To confirm FrameMaker+SGML's default behavior of creating these processing instructions, use this rule:

```
writer output book processing instructions;
```

For information on these rules, see "writer" on page 442 and "output book processing instructions" on page 422.

19 Modifications to the default translation

20 Read/Write Rules Summary

This chapter lists the available read/write rules by category and briefly describes the purpose of each rule. The categories, which are arranged alphabetically, are as follows: all elements, attributes, books, entities, cross-references, footnotes, equations, markers, graphics, processing instructions, SGML documents, tables, text, text insets, and variables.

All Elements

To	Use this rule	Page
Translate an SGML element	element	345
Discard or unwrap a FrameMaker+SGML element on export	fm_element	367
Translate an SGML element to a FrameMaker+SGML element	is_fm_element	391
Translate an SGML attribute within the context of a single SGML element	attribute	335
Inform FrameMaker+SGML not to update a FrameMaker+SGML element's definition when updating an existing EDD	preserve_fm_element_definition	422
Discard a FrameMaker+SGML or SGML element	drop	342
Discard the content but not the structure of a FrameMaker+SGML or SGML element	drop_content	344
Discard the structure but not the content of an SGML or FrameMaker+SGML element	unwrap	436

Attributes

To	Use this rule	Page
Translate an SGML attribute	attribute	335
Discard a FrameMaker+SGML attribute	fm_attribute	366
Translate an SGML attribute to a FrameMaker+SGML attribute	is_fm_attribute	385
Translate an SGML attribute within the context of a single SGML element	element	345

To	Use this rule	Page
Discard an SGML or FrameMaker+SGML attribute	drop	342
Translate an SGML attribute to a particular FrameMaker+SGML property	is fm property	396
Translate a value for an SGML attribute to a FrameMaker+SGML property value	is fm property value	398
Translate a value of an SGML notation attribute or name token group to a value for a FrameMaker+SGML choice attribute	is fm value	413
Translate an SGML attribute value to a FrameMaker+SGML property or a choice attribute value	value	439
Specify the value to use for an SGML implied attribute when a document instance provides no value	implied value is	377

Books

To	Use this rule	Page
Specify whether to use elements or processing instructions to indicate book components when reading an SGML document	generate book	373
Specify elements to use to indicate book components when reading an SGML document	put element (described with generate book)	373
Specify the use of processing instructions to indicate book components when reading an SGML document	use processing instructions (described with generate book)	373
Specify whether or not to write processing instructions that indicate book components in an SGML document	output book processing instructions	422

Cross-references

To	Use this rule	Page
Translate SGML elements to FrameMaker+SGML cross-reference elements	is fm cross-reference element	389

To	Use this rule	Page
Import FrameMaker+SGML cross-reference properties when no SGML attribute exists	fm_property	370
Translate FrameMaker+SGML cross-reference properties when no SGML attribute exists	value_is (described with fm_property)	370
Translate an SGML attribute to a particular FrameMaker+SGML property	is_fm_property	396
Translate a value for an SGML attribute to a FrameMaker+SGML property value	is_fm_property_value	398
Translate a value of an SGML notation attribute or name token group to a value for a FrameMaker+SGML choice attribute	is_fm_value	413
Translate a FrameMaker+SGML cross-reference element to text in SGML	fm_element unwrap	367, 436

Entities

To	Use this rule	Page
Translate an SGML entity reference to an appropriate FrameMaker+SGML representation	entity	349
Determine the form of names of entities created for exported graphics	entity_name_is	352
Translate external data entity references to FrameMaker+SGML non-element markers	external_data_entity_reference	362
Translate an entity reference to a FrameMaker+SGML variable	is_fm_variable	414
Translate an entity reference to a single character	is_fm_char	387
Translate an entity reference to an element on a reference page	is_fm_reference_element	402
Translate an SDATA entity reference to a text inset	is_fm_text_inset	411
Determine the formatting of a text inset	reformat_as_plain_text	429
	reformat_using_target_document_catalogs	429
	retain_source_document_formatting	430

To	Use this rule	Page
Discard external data entity references	drop	342

Equations

To	Use this rule	Page
Translate an SGML element to a FrameMaker+SGML equation element	is_fm_equation_element	392
Specify export information for translating FrameMaker+SGML equations	equation	355
Specify the filename used for exporting an equation	export_to_file	359
Determine the form of names of entities created for exported equations	entity_name_is	352
Specify the data content notation for an exported equation	notation_is	420
Determine whether FrameMaker+SGML uses the dpi attribute or the impsize attribute for equations and also the resolution used	specify_size_in	431
Translate FrameMaker+SGML cross-reference properties when no SGML attribute exists	fm_property	370
Translate FrameMaker+SGML cross-reference properties when no SGML attribute exists	value_is (described with fm_property)	370
Translate FrameMaker+SGML equation properties to SGML attributes	is_fm_property	396
Translate a value for an SGML attribute to a FrameMaker+SGML property value	is_fm_property_value	398
Translate a value of an SGML notation attribute or name token group to a value for a FrameMaker+SGML choice attribute	is_fm_value	413
Translate an SGML attribute value to a FrameMaker+SGML property or a choice attribute value	value	439

Footnotes

To	Use this rule	Page
Translate an SGML element to a FrameMaker+SGML footnote element	`is_fm_footnote_element`	393

Graphics

To	Use this rule	Page
Translate an SGML element to a FrameMaker+SGML graphic element	`is_fm_graphic_element`	394
Specify export information for translating FrameMaker+SGML graphics	`anchored_frame`	333
Specify export information for translating FrameMaker+SGML graphics that have a single inset	`facet`	364
Specify the filename used for exporting a graphic or a facet of a graphic	`export_to_file`	359
Force the software to export graphic files that were imported by reference	`convert_referenced_graphics`	340
Determine the form of names of entities created for exported graphics	`entity_name_is`	352
Specify the data content notation for an exported graphic	`notation_is`	420
Determine whether FrameMaker+SGML uses the `dpi` attribute or the `impsize` attribute for imported graphics objects and also the resolution used	`specify_size_in`	431
Translate FrameMaker+SGML cross-reference properties when no SGML attribute exists	`fm_property`	370
Translate FrameMaker+SGML cross-reference properties when no SGML attribute exists	`value_is` (described with `fm_property`)	370
Translate FrameMaker+SGML graphic properties to SGML attributes	`is_fm_property`	396
Translate a value for an SGML attribute to a FrameMaker+SGML property value	`is_fm_property_value`	398
Translate a value of an SGML notation attribute or name token group to a value for a FrameMaker+SGML choice attribute	`is_fm_value`	413

To	Use this rule	Page
Translate an SGML attribute value to a FrameMaker+SGML property or a choice attribute value	value	439

Markers

To	Use this rule	Page
Discard FrameMaker+SGML non-element markers or translate them to processing instructions	fm_marker	368
Translate an SGML element to a FrameMaker+SGML marker element	is_fm_marker_element	395
Determine whether marker text for marker elements becomes content or an attribute value in SGML	marker_text_is	418
Translate external data entity references to FrameMaker+SGML non-element markers	external_data_entity_reference	362
Translate unrecognized processing instructions to non-element markers	processing_instruction	425
Translate FrameMaker+SGML non-element markers to processing instructions	is_processing_instruction	415
Discard non-element markers	drop	342
Translate FrameMaker+SGML cross-reference properties when no SGML attribute exists	fm_property	370
Translate FrameMaker+SGML cross-reference properties when no SGML attribute exists	value_is (described with fm_property)	370
Translate FrameMaker+SGML marker properties to SGML attributes	is_fm_property	396
Translate a value for an SGML attribute to a FrameMaker+SGML property value	is_fm_property_value	398
Translate a value of an SGML notation attribute or name token group to a value for a FrameMaker+SGML choice attribute	is_fm_value	413
Translate an SGML attribute value to a FrameMaker+SGML property or a choice attribute value	value	439

Processing instructions

To	Use this rule	Page
Specify the treatment of unrecognized processing instructions	processing_instruction	425
Specify the use of processing instructions to indicate book components when reading an SGML document	use_processing_instructions (described with generate_book)	373
Specify whether or not to write processing instructions that indicate book components in an SGML document	output_book_processing_instructions	422
Translate FrameMaker+SGML non-element markers to SGML processing instructions	fm_marker	368
Translate FrameMaker+SGML non-element markers to SGML	is_processing_instruction	415
Discard processing instructions	drop	342

SGML documents

To	Use this rule	Page
Specify whether or not to use an external DTD subset to contain the DTD for an SGML document created by FrameMaker+SGML	include_dtd	379
Specify whether or not to include an SGML declaration in an SGML document created by FrameMaker+SGML	include_sgml_declaration	380
Specify where to find system and public identifiers for an external DTD subset	external_dtd	363
Specify whether to create an entire SGML document or just an SGML document instance	write_sgml_document	441
	write_sgml_document_instance_only	441

Tables

To	Use this rule	Page
Translate an SGML element to a FrameMaker+SGML table element	is_fm_table_element	408

To	Use this rule	Page
Translate an SGML element to a FrameMaker+SGML element for a particular table part	`is_fm_table_part_element`	409
Specify an element that corresponds to the CALS `colspec` element	`is_fm_colspec`	389
Specify an element that corresponds to the CALS `spanspec` element	`is_fm_spanspec`	406
When creating a FrameMaker+SGML table, insert a table part even if that part is empty	`insert_table_part_element`	381
Specify that a particular element always indicates a new table row	`start_new_row`	433
Indicate the start of a vertical straddle	`start_vertical_straddle`	434
Indicate the end of a vertical straddle	`end_vertical_straddle`	348
Specify the ruling style used for all tables	`table_ruling_style_is`	435
Specify the resolution used for column widths with proportional widths	`proportional_width_resolution_is`	426
Specify that the software write the width of table columns using proportional units	`use_proportional_widths`	438
Translate FrameMaker+SGML cross-reference properties when no SGML attribute exists	`fm_property`	370
Translate FrameMaker+SGML cross-reference properties when no SGML attribute exists	`value_is` (described with `fm_property`)	370
Translate FrameMaker+SGML table properties to SGML attributes	`is_fm_property`	396
Translate a value for an SGML attribute to a FrameMaker+SGML property value	`is_fm_property_value`	398
Translate a value of an SGML notation attribute or name token group to a value for a FrameMaker+SGML choice attribute	`is_fm_value`	413
Translate an SGML attribute value to a FrameMaker+SGML property or a choice attribute value	`value`	439

Text

To	Use this rule	Page
Translate an entity reference to a FrameMaker+SGML variable	entity	349
Translate an entity reference to a single character	is_fm_char	387
Determine the treatment of line breaks in reading and writing SGML documents	line_break	417
Define mappings between characters in the SGML and FrameMaker+SGML character sets	character_map	337

Text insets

To	Use this rule	Page
Translate an SDATA entity reference to a FrameMaker+SGML text inset	entity	349
Translate an SDATA entity reference to a text inset	is_fm_text_inset	411
Determine the formatting of a text inset	reformat_as_plain_text	429
	reformat_using_target_document_catalogs	429
	retain_source_document_formatting	430

Variables

To	Use this rule	Page
Translate an SGML element to a FrameMaker+SGML system variable element	is_fm_system_variable_element	407
Translate an entity reference to a FrameMaker+SGML variable	is_fm_variable	414
Translate an SGML entity reference to a FrameMaker+SGML variable	entity	349
Determine treatment of FrameMaker+SGML non-element variables	fm_variable	372
Translate a FrameMaker+SGML system variable element to text in SGML	fm_element unwrap	367, 436
Discard nonelement variables	drop	342

21 Read/Write Rules Reference

This chapter provides a reference to all SGML read/write rules, listed in alphabetical order. The entry for each rule starts with a brief explanation of the purpose of the rule and how to use it. The rule's description may include the following sections:

Synopsis and contexts The rule's syntax and the context in which it can be used. If the rule occurs as a subrule of another rule, the more general rule is shown. If the rule can be used in multiple contexts, the synopsis shows each context. Each entry in this section shows a valid rule that has the current rule either at the highest level or as one of its subrules.

Rule synopses use the following conventions:

- Bold portions and nonitalicized portions of a rule are entered by you as shown.
- Italicized portions of a rule indicate the rule's arguments or possible subrules; you enter your values.
- Brackets [ ] indicate optional parts of a rule; the entire form within the brackets can be included or omitted.

Arguments The possible arguments to the rule. If an argument is optional, its default value is provided. Some rules have *`subrule`* as one of their arguments. In these cases, a list of possible subrules is provided. Some rule arguments allow variables. In these cases, a list of possible variables is provided.

Details Details on how to use the rule and on FrameMaker+SGML behavior when the rule is not supplied.

Examples Various examples of the rule.

See also Cross-references to other relevant information in the manual.

For information on how to create an SGML read/write rules file and on the syntax of rules, see Chapter 11, "SGML Read/Write Rules and Their Syntax."

anchored frame

Use the `anchored frame` rule only in an `element` rule for a graphic element, to provide information the software needs when writing to SGML a document containing graphics. For the circumstances when FrameMaker+SGML will write out the anchored frame contents as a graphic file, you use this rule to specify information such as graphic file format or notation name in subrules. Use this rule to specify information about the file FrameMaker+SGML

creates to contain the external data entity—for instances that are not anchored frames containing only a single imported graphic object.

Synopsis and contexts

```
element "gi" {
   is fm graphic element ["fmtag"];
   writer anchored frame subrule;
. . .}
```

Arguments

gi An SGML generic identifier.

fmtag A FrameMaker+SGML element tag.

subrule An `anchored frame` rule can have the following subrules:

`entity name is`, page 352, tells the software how to create the base name for the entity associated with this element type.

`export to file`, page 359, tells FrameMaker+SGML how to write the file name when it creates a new graphic file, and optionally the graphic format for the file.

`notation is`, page 420, specifies the data content notation of the entity file.

`specify size in`, page 431, specifies the units to use when writing the file.

Details

On export, if the anchored frame contains only a single imported graphic file, FrameMaker+SGML uses that graphic file for the resulting SGML graphic element by default. If the anchored frame contains more than one graphic file, or has been modified using FrameMaker+SGML graphics tools, the software writes out a graphic file to be used. The default format for these graphic files is CGM. The export format can be changed with the `export to file` rule. For more information about translating anchored frame contents, see Chapter 15, “Translating Graphics and Equations.”

Examples

- Assume you use the `Graphic` element for all graphic elements. If the graphic contains any single facet, assume the graphic was imported as an entity and you want the default behavior. However, if the author used FrameMaker+SGML graphic tools to create the objects in the graphic element, you want the file written in QuickDraw PICT format.

To accomplish all this, use this rule:

```
element "graphic" {
   is fm graphic element;
   writer anchored frame export to file "$(docname).pic"
   as "PICT";
}
```

Assume the FrameMaker+SGML document is named `mydoc.fm`. For the first graphic that is not a single facet, the software writes out a graphic file named `mydoc1.pic` in the PICT format.

If the export DTD declares an entity attribute to identify the graphic file with the `graphic` element, the software generates the following entity declaration:

```
<!ENTITY graphic1 SYSTEM "mydoc1.pic" NDATA PICT>
```

If the export DTD includes only a `file` attribute to associate the graphic file with the graphic element, the software uses this filename as its value.

See also

Related rules	"equation" on page 355 "facet" on page 364
Rules mentioned in synopses	"element" on page 345 "is fm equation element" on page 392 "is fm graphic element" on page 394 "writer" on page 442
General information on this topic	Chapter 15, "Translating Graphics and Equations"

attribute

Use the `attribute` rule to describe how to process an SGML attribute. By default, an SGML attribute translates to a FrameMaker+SGML attribute of the same name. Usually, this rule occurs as a subrule of the `element` rule, to describe treatment of the attribute *`attr`* within the SGML element *`gi`*.

Synopsis and contexts

```
1.[sgmldv] attribute "attr" {. . .
      subrule;
   . . .}

2.element "gi" {. . .
      [sgmldv] attribute "attr" {. . .
         subrule;
   . . .} . . .}
```

Arguments

sgmldv An optional SGML declared value, specifying the type of the SGML attribute. Legal values are: `cdata`, `name`, `names`, `nmtoken`, `nmtokens`, `number`, `numbers`, `nutoken`, `nutokens`, `entity`, `entities`, `notation`, `id`, `idref`, `idrefs`, and `group`.

attr The name of an SGML attribute.

gi An SGML generic identifier.

subrule An `attribute` rule can have one or more of the following subrules:

`drop`, page 342, discards the attribute.

`implied value is`, page 377, specifies the value to use for an impliable attribute for which no value is given in a document instance.

`is fm attribute`, page 385, translates an SGML attribute into a FrameMaker+SGML attribute.

`is fm property`, page 396, translates an SGML attribute to a FrameMaker+SGML property such as the width of columns in a table. This subrule is applicable only to cross-reference, marker, graphic, equation, table, and table part elements.

`value`, page 439, translates one of the possible values of an SGML name token group or a notation attribute to a specific token of a FrameMaker+SGML choice attribute.

Details

- In some cases, the same attribute may occur in several SGML elements and may require the same treatment for most of those occurrences. In these situations, you can use the `attribute` rule at the highest level to set the default treatment of the attribute. You can then override the default in individual `element` rules.
- The `drop`, `is fm attribute`, and `is fm property` subrules of the `attribute` rule are mutually exclusive. That is, if you use one of these rules, you cannot use either of the other rules.

Examples

- The following rule specifies that the `sec` attribute of the SGML `list` element has a name token group and corresponds to the attribute `Security` on the corresponding FrameMaker+SGML element:

```
element "list"
   group attribute "sec"
      is fm attribute "Security";
```

- Assume you have several elements that represent graphic objects. Each of them has an attribute `w`, representing the width of the object. Use this rule to make the width be 3 inches unless otherwise specified for a particular element:

```
attribute "w" {
   is fm property width;
   implied value is "3in";
}
```

- Assume you have an element `team` with an attribute `color`. The possible values for `color` are `r`, `b`, and `g`. To change the names of these values in the corresponding FrameMaker+SGML choice attribute, use this rule:

```
element "team" {
 attribute "color" {
      value "r" is fm value "Red";
      value "b" is fm value "Blue";
      value "g" is fm value "Green";
}}
```

See also

Related rules	"fm attribute" on page 366 "is fm attribute" on page 385
Rules mentioned in synopses	"element" on page 345
General information on this topic	Chapter 12, "Translating Elements and Their Attributes"

character map

Use the `character map` rule to define mappings between characters in the SGML and FrameMaker+SGML character sets. Many characters can be expressed using a string; others require using the appropriate integer character code.

Synopsis and contexts

1. **character map** is $cmap_1$ [, . . ., $cmap_n$];

2. reader **character map** is $cmap_1$ [, . . ., $cmap_n$];

3. writer **character map** is $cmap_1$ [, . . ., $cmap_n$];

Arguments

$cmap_i$

A mapping between the character set used in the SGML document and the FrameMaker+SGML character set. Each $cmap_i$ has one of the following forms:

```
sgmlch = fmch;
sgmlch = trap;
trap = fmch;
```

sgmlch is either a 1-character string or a character code representing a character in the SGML character set. *sgmlch* can be a single character only if that character has the same character code in both the FrameMaker+SGML and SGML character sets. Otherwise, you must use the integer character code.

fmch is either a 1-character string or a character code representing a character in the FrameMaker+SGML character set.

For information on how to represent character codes and special characters in strings, see "Strings and constants" on page 200.

Details

- Some characters might be defined in only one of the two character sets. The keyword `trap` is provided for this situation. By default, FrameMaker+SGML discards trapped characters.
- The character map need not be a one-to-one mapping. If a character in the input document is mapped to multiple characters in the output character set, FrameMaker+SGML uses the output character from the *last* mapping to appear in the `character map` rule.
- If you use the `character map` rule at the highest level, do not also use it inside either a `reader` rule or a `writer` rule. If you use this rule inside a `reader` rule or a `writer` rule and also use it at the highest level, FrameMaker+SGML ignores the highest-level `character map` rule. You can only have one occurrence of this rule at the highest level.

  Similarly, the `character map` rule can appear in one `reader` rule and one `writer` rule at most. The software ignores any subsequent uses of the `character map` rule.
- If you use the `character map` rule at the highest level, its behavior is bidirectional. For example, you could have this rule:

```
character map is 0x20 = 0x12;
```

This rule specifies that the ISO Latin-1 space character (character code `0x20`) maps to the FrameMaker+SGML thin space character (character code `0x12`). With this rule, FrameMaker+SGML translates a thin space to a standard space when it writes an SGML document. However, this rule translates *all* spaces in an SGML document to thin spaces in a corresponding FrameMaker+SGML document. This is unlikely to be the desired behavior. For this reason, instead you should use this rule:

```
reader character map is 0x20 = 0x12;
```

- By default, FrameMaker+SGML assumes that the character set your SGML documents use is ISO Latin-1. It provides a default mapping between those character sets. For details, see Appendix E, "Character Set Mapping." For information on other character sets you can use, see Appendix F, "ISO Public Entities."

- By default, on export FrameMaker+SGML produces a character in the SGML document for most printing characters in the corresponding FrameMaker+SGML document. FrameMaker+SGML documents occasionally include unusual characters that serve no purpose outside FrameMaker+SGML. For example, the codes `0x01` and `0x03` are nonprinting characters that represent information about the insertion point movement. On export FrameMaker+SGML traps such characters, so that they don't appear in an exported SGML document.

  Similarly, on import FrameMaker+SGML produces a character in the FrameMaker+SGML document for most printing characters. It traps all control characters other than a tab or newline character.

- FrameMaker+SGML has an 8-bit character set. The SGML declaration can specify any character set that the SGML parser can handle. Because part of the character set description in the SGML declaration is not human-readable and may not be interpretable automatically, any differences between the native FrameMaker+SGML character set and the character set in the SGML document must be specified with the `character map` rule.

- By default, FrameMaker+SGML discards trapped characters. You can provide an SGML API client to change the processing of trapped characters. For information on creating an SGML API client, see the *SGML API Programmer's Guide*.

Examples

- Both the FrameMaker+SGML and default SGML character sets have a character code for the character ó (lowercase o with an acute accent). In FrameMaker+SGML, the character code is `0x97`; in the default SGML character set, the character code is `0xF3`. If you want to trap the SGML character that looks like ó, you might try using this rule:

  ```
  character map is "ó" = trap;
  ```

  Because you enter your SGML read/write rules in a FrameMaker+SGML document, however, FrameMaker+SGML interprets that rule as:

  ```
  character map is 0x97 = trap;
  ```

which is not the behavior you want. Instead, you should use this rule:

```
character map is 0xF3 = trap;
```

- By default, FrameMaker+SGML maps the SGML broken bar character to the FrameMaker+SGML solid bar character |. The rule for doing so could be written in the following equivalent ways:

```
character map is 0xA6 = "|";
character map is 0xA6 = 0x7C;
character map is "\xA6" = "\x7C";
```

- To trap the SGML broken bar character, use this rule:

```
character map is 0xA6 = trap;
```

See also

- For information on the FrameMaker+SGML character set, see the FrameMaker user's manual.
- For details of the default mapping between the FrameMaker+SGML and ISO Latin-1 character sets, see Appendix E, "Character Set Mapping."

convert referenced graphics

Use the `convert referenced graphics` rule to force the software to write out a graphic file when exporting a graphic element that uses a referenced graphic. By default, FrameMaker+SGML doesn't write out graphic files in this case. It is usually more advantageous to simply reference the same graphic file in both the SGML and the FrameMaker+SGML document. However, you can use this rule to convert all such graphic files to a specific format.

Synopsis and contexts

```
element "gi" { . . .
writer facet "facetname" convert referenced graphics;
. . . }
```

Arguments

There are no arguments for this rule

Details

- This rule must be a subrule of a facet rule for a graphic element.
- By default, if a graphic or equation element is imported by reference, the software does not create a new graphic file for the element when exporting a FrameMaker+SGML document. You can change that behavior using this rule.

Examples

- Assume you want to convert imported graphic files in `graphic` elements which have not been edited in the FrameMaker+SGML document, to the PICT format. With the following example, the software would create PICT files for each of these graphic elements:

```
element "graphic" {
   is fm graphic element;

writer {
facet default {
   convert referenced graphics;
   export to file "$(entity).pic" as "PICT";
}}
```

- For graphic elements with a single TIFF facet, the following example converts the graphic files in the graphic element to PICT:

```
element "graphic" {
   is fm graphic element;
   writer facet "TIFF"{
   convert referenced graphics;
   export to file "$(entity).pic" as
      "PICT";
}}
```

See also

Related rules	"facet" on page 364
	"export to file" on page 359
	"writer" on page 442
General information on this topic	"Translating Graphics and Equations" on page 273

do not include dtd

See "include dtd" on page 379.

do not include sgml declaration

See "include sgml declaration" on page 380.

do not output book processing instructions

See "output book processing instructions" on page 422.

drop

Use the `drop` rule to indicate information that you want discarded. Examples of information you might discard include an SGML element or attribute that has no counterpart in FrameMaker+SGML, or a FrameMaker+SGML non-element marker that has no counterpart in SGML.

Synopsis and contexts

1. `attribute "`*`attr`*`"` **`drop`**`;`

2. `element "`*`gi`*`"` **`drop`**`;`

3. `element "`*`gi`*`" {. . .`
   `attribute "`*`attr`*`"` **`drop`**`;`
   `. . .}`

4. `external data entity reference` **`drop`**`;`

5. `fm attribute "`*`attr`*`"` **`drop`**`;`

6. `fm element "`*`fmtag`*`"` **`drop`**`;`

7. `fm marker` $type_1$ `[, . . ., ` $type_n$`]` **`drop`**`;`

8. `fm variable` **`drop`**`;`

9. `processing instruction` **`drop`**`;`

Arguments

attr	The name of an SGML or FrameMaker+SGML attribute. Note that `fm attribute` names are case-sensitive and should appear as in the EDD. The case of DTD attribute names depends on the setting of `NAMECASE` in the SGML.dcl file.
gi	An SGML generic identifier.
fmtag	A FrameMaker+SGML element tag. These names are case-sensitive and should appear in the rule the same as in the EDD.
$type_i$	A FrameMaker+SGML marker type, such as Index or Type 22.

Details

- When FrameMaker+SGML encounters something to be discarded, it makes no attempt to insert the corresponding information into the document it is creating. In the case of a dropped element, it also discards all descendant elements.
- When creating an EDD from a DTD or a DTD from an EDD, FrameMaker+SGML does not generate an element definition corresponding to a dropped element. It also removes any references to the specified element in content rules for other elements unless you've specified a `preserve fm element definition` rule for those elements.
- You can write an SGML API client to process dropped information. Your client must also handle retrieving discarded information if it is needed when the document is written back to its original format.
- If you use the `drop` rule in some rule, you can use no other subrules of the same rule. For example, you cannot specify that FrameMaker+SGML both drop an attribute and translate it to a FrameMaker+SGML property with the `is fm property` rule.

Examples

- An SGML element used instead of a processing instruction to indicate that a page or line break is desired may be discarded when the SGML document is read. Text formatting rules in the EDD can be used to indicate a page break in FrameMaker+SGML; there is no need to mark the break with an element. To drop the SGML element `break`, use this rule:

  ```
  element "break" drop;
  ```

- By default, FrameMaker+SGML stores processing instructions that it does not recognize in non-element markers. In this way, even though FrameMaker+SGML does not perform special processing on the processing instruction, when you save the FrameMaker+SGML document back to SGML, the software writes out the processing instruction so that a different application can use it. If you don't need to write out the processing instructions, you could use this rule:

  ```
  processing instruction drop;
  ```

See also

Related rules	“drop content” on page 344 “unwrap” on page 436 “preserve fm element definition” on page 422
Rules mentioned in synopses	“attribute” on page 335 “element” on page 345 “external data entity reference” on page 362 “fm attribute” on page 366 “fm element” on page 367 “fm marker” on page 368 “fm variable” on page 372 “processing instruction” on page 425
General information on this topic	Chapter 12, “Translating Elements and Their Attributes”

drop content

Use the `drop content` rule to either create a FrameMaker+SGML empty element or an SGML element with no content from occurrences of *gi*.

Synopsis and contexts

```
1.element "gi" {
   is fm element "fmtag";
   reader drop content;
   }

2.element "gi" {
   is fm element "fm tag";
   writer drop content;
   }
```

Arguments

gi	An SGML generic identifier.

Details

- You can use this rule when you have an element whose content is created in a system-specific way. Because you plan to rely on some system to create the content, the existing content at the time you import or export a document may not be relevant. For example, you may have an SGML element intended to contain a chapter number. In FrameMaker+SGML, you use FrameMaker+SGML's formatting capabilities to have the system maintain the value. When reading in the SGML document, you can drop the current content of the number element.
- Use `drop content` inside a `reader` rule when you translate SGML documents to FrameMaker+SGML documents. Use it inside a `writer` rule when you translate FrameMaker+SGML documents to SGML.

Examples

- Assume your DTD has a `toc` element that represents the table of contents for an SGML document. Because FrameMaker+SGML can automatically generate a table of contents, this SGML element can have its contents dropped upon import.

```
element "toc" reader drop content;
```

- Assume the `total` element's content is computed by an SGML API client. Outside the FrameMaker+SGML environment you will use a different program to perform the computation. Consequently, you do not want the value that is current when the document is exported. To discard the current value, use this rule:

```
element "total" writer drop content;
```

See also

Related rules	"drop" on page 342 "unwrap" on page 436
Rules mentioned in synopses	"element" on page 345 "reader" on page 427 "writer" on page 442
General information on this topic	Chapter 12, "Translating Elements and Their Attributes" *SGML API Programmer's Guide*

element

You use the `element` rule as the primary rule for translating between an SGML element and its corresponding FrameMaker+SGML representation.

Synopsis and contexts

```
element "gi" {. . .
   subrule;
. . .}
```

Arguments

gi An SGML generic identifier.

subrule The subrules of `element` indicate the treatment of the SGML element:

`attribute`, page 335, specifies what to do with an SGML element's attributes.

`drop`, page 342, discards the element.

`fm attribute`, page 366, specifies what to do with attributes present in the FrameMaker+SGML representation of the element but not in the SGML representation of it.

`fm property`, page 370, specifies what to do with FrameMaker+SGML properties associated with the element. This subrule applies only to elements that correspond to graphic, equation, table, table part, cross-reference, or marker elements.

`is fm colspec`, page 389, specifies that the element represents a CALS table colspec. This subrule applies only to CALS tables.

`is fm cross-reference element`, page 389, specifies that the element corresponds to a FrameMaker+SGML cross-reference element.

`is fm element`, page 391, translates the element to a particular FrameMaker+SGML element. You use this subrule to rename the element.

`is fm equation element`, page 392, specifies that the element corresponds to a FrameMaker+SGML equation element.

`is fm footnote element`, page 393, specifies that the element corresponds to a FrameMaker+SGML footnote element.

`is fm graphic element`, page 394, specifies that the element corresponds to a FrameMaker+SGML graphic element.

`is fm marker element`, page 395, specifies that the element corresponds to a FrameMaker+SGML marker element.

`is fm spanspec`, page 406, specifies that the element represents a CALS table spanspec. This subrule applies only to CALS tables.

`is fm system variable element`, page 407, specifies that the element corresponds to a FrameMaker+SGML system variable element.

`is fm table element`, page 408, specifies that the element corresponds to a FrameMaker+SGML table element.

`is fm table part element`, page 409, specifies that the element corresponds to a FrameMaker+SGML element for a particular table part, such as a table title or cell.

`marker text is`, page 418, specifies whether the text of a FrameMaker+SGML marker element should be element content or an attribute value in SGML. This subrule applies only to marker elements.

`reader drop content`, page 344, specifies that the content but not the structure of an element should be discarded on import of an SGML document.

`reader end vertical straddle`, page 348, indicates that the associated table cell or row element terminates a vertical table straddle. This subrule applies only to table cell or row elements.

`reader insert table part element`, page 381, indicates that the software should insert the specified table part (title, heading or footing), even if the SGML element structure does not contain the corresponding element. This subrule applies only to table elements.

`reader line break`, page 417, determines whether to interpret line breaks in text segments in elements in the SGML document as forced returns or spaces within the elements.

`reader start new row`, page 433, indicates that the occurrence of the associated table cell element always starts a new row in the table. This subrule applies only to table cell elements.

`reader start vertical straddle`, page 434, indicates that the associated table cell element starts a vertical table straddle. This subrule applies only to table cell elements.

`unwrap`, page 436, indicates that the content of the element, but not the element itself, should be included in the translated document.

`writer anchored frame`, page 333, tells FrameMaker+SGML what to do with graphic elements other than those with a single non-internal FrameMaker+SGML facet. This subrule applies only to graphic elements.

`writer drop content`, page 344, specifies that the content but not the structure of an element should be discarded on export of a FrameMaker+SGML document.

`writer equation`, page 355, tells FrameMaker+SGML what to do with equation elements. This subrule applies only to equation elements.

`writer facet`, page 364, tells FrameMaker+SGML what to do with a graphic element that has a single non-internal FrameMaker+SGML facet. This subrule applies only to graphic elements.

`writer line break`, page 417, limits the length of lines the software generates in the SGML document.

Details

- If you use either the `drop` or `unwrap` subrule of an `element` rule, that subrule must be the element's only subrule. For example, you cannot both unwrap an SGML element and translate it to a FrameMaker+SGML element.

Examples

- To translate the SGML element `p` to the FrameMaker+SGML element `Paragraph`, use this rule:

```
element "p" is fm element "Paragraph";
```

- To translate the SGML element `tab2` to a FrameMaker+SGML table element `Two Table` with two columns, use this rule:

```
element "tab2" {
   is fm table element "Two Table";
   fm property columns value is "2";
}
```

See also

Related rules	"fm element" on page 367
General information on this topic	Chapter 12, "Translating Elements and Their Attributes"

end vertical straddle

Use the `end vertical straddle` rule inside the `element` rule for a table row or table cell to specify that the row (or the row containing the cell) indicates the end of a vertical straddle started by some earlier table cell element. The straddle can end either before the current row or at the current row.

Synopsis and contexts

```
element "gi" {
   is fm table row_or_cell element ["fmtag"];
   reader end vertical straddle "name1" [, . . . "namen"]
      [before this row];
. . .}
```

Arguments

gi	An SGML generic identifier.

row_or_cell	One of the keywords: `row` or `cell`.
fmtag	A FrameMaker+SGML element tag.
$name_i$	A name associated with a table straddle. Each $name_i$ must occur in a corresponding `start vertical straddle` rule.

Details

- Your DTD may contain elements that you want to format as tables in FrameMaker+SGML even though the element hierarchy does not match that required by FrameMaker+SGML for tables. In such a situation, the nature of the element hierarchy may indicate where vertical straddles begin and end. The `end vertical straddle` rule allows you to specify such elements.
- Use this rule in conjunction with the `start vertical straddle` rule. That rule specifies a table cell that indicates the first cell in a vertical straddle. In the `start vertical straddle` rule, give a name to the particular straddle started by that element. In the `end vertical straddle` rule, you must specify by name which vertical straddles started by earlier cells are ended by the occurrence of *gi*.
- If you use this rule for a table cell element, you can end only one vertical straddle. If you use it for a table row element, you can end more than one vertical straddle.
- If you use this element without the `before this row` keyword phrase, the cell or row (*gi*) specified in the rule becomes part of the straddle. If you do include that keyword phrase, then the straddle ends in the row above the one specified.

Examples

- For an example of the use of this rule, see "Creating vertical straddles" on page 267.

See also

Related rules	"start vertical straddle" on page 434
General information on this topic	Chapter 14, "Translating Tables"

entity

You use the `entity` rule to translate an SGML entity to an appropriate FrameMaker+SGML representation. With this rule, you can translate an SGML entity to a particular character or set of characters, a reference element, a text inset, or a FrameMaker+SGML variable. If you choose to translate the entity to a text inset, you can also specify how to format that text inset in the resulting document.

Synopsis and contexts

```
1.entity "ename" {
      type_rule;
      [format_rule;]
   . . .}

2.reader entity "ename" {
      type_rule;
      [format_rule;]
   . . .}
```

Arguments

ename An SGML entity name.

type_rule One of the following:

`is fm char`, page 387, translates the entity to a particular character in FrameMaker+SGML.

`is fm reference element`, page 402, translates the entity to an element whose content resides on a reference page in the FrameMaker+SGML document.

`is fm text inset`, page 411, translates the entity to a FrameMaker+SGML text inset.

`is fm variable`, page 414, translates the entity to a FrameMaker+SGML non-element variable.

format_rule This subrule can be specified only if *type_rule* is `is fm text inset`. One of the following:

`reformat as plain text`, page 429, specifies that the software remove the internal structure and formatting from the text of the text inset and apply the formatting used at the insertion point.

`reformat using target document catalogs`, page 429, specifies that the software retain the text inset's internal structure and apply the containing document's formats and element format rules to the text. This rule is applied as if the following three options were checked when a file is imported through the File>ImportFile menu: 1. Reformat Using Target Document's catalog; 2. While importing Remove: Manual Page Breaks; and 3. While Importing Remove: Other Format Overrides. For more information, see the section "Importing text" in Chapter 5 of the FrameMaker+SGML User Guide.

`retain source document formatting`, page 430, specifies that the software remove the internal structure of the text inset and

retain the formatting of the text inset as it appeared in the source document.

Details

- If you use the `entity` rule at the highest level, then it applies both on import and export. If you use it inside a `reader` rule, then FrameMaker+SGML translates the SGML entity as specified when importing an SGML document, but does not create an entity reference on export.

- While you can use this rule to translate any entity type to a text inset, we recommend you convert only SDATA entities to text insets. Note that the source file for such a text inset must be a format FrameMaker+SGML can automatically filter. Also, such a text inset cannot use an SGML document as the source file.
- By default FrameMaker+SGML imports SGML external text entities as text insets. The source files for these insets can be SGML or text files. The software stores entity information on the Entity Definitions reference page so it can export the text inset as an external text entity.

Examples

- To translate the SGML text entity `mn` to the FrameMaker+SGML variable `Manual Name`, use this rule:

  ```
  entity "mn" is fm variable "Manual Name";
  ```

  Suppose the text entity mn is declared as `<!ENTITY mn "Developer's Guide">`, and the template for the application doesn't contain a variable named "Manual Name". Then the software will create a FrameMaker+SGML variable named "Manual Name" defined as "Developer's Guide" and replace the reference in the text with the variable text "Developer's Guide".

  However, if a FrameMaker variable named "Manual Name", defined for example as "My Favorite Manual", currently exists in the template for the application, the software will not create a new variable nor modify the existing one. It will replace the reference in the text with the variable text "My Favorite Manual".

- To have FrameMaker+SGML create a text inset for the `legalese` entity using the text in the file `legal.fm` and to have the software format that text inset as it appears in `legal.doc`, use this rule:

  ```
  entity "legalese" {
     is fm text inset "legal.fm";
     retain source document formatting;
  }
  ```

See also

General information on this topic	Chapter 13, "Translating Entities and Processing Instructions" Chapter 17, "Translating Variables and System Variable Elements"

entity name is

Use the `entity name is` rule only in an `element` rule for a graphic or equation element to provide information the software needs when writing a document containing graphics or equations to SGML. The `entity name is` rule determines the name FrameMaker+SGML gives an entity reference it generates for the graphic or equation.

Synopsis and contexts

```
element "gi" {
   is fm graphic_or_eqn element ["fmtag"];
   writer type ["facetname"] entity name is "ename";
. . .}}
```

Arguments

gi	An SGML generic identifier.
graphic_or_eqn	One of the keywords: `graphic` or `equation`.
type	One of the rules `anchored frame`, `facet`, or `equation`. If `facet`, you must also supply the *facetname* argument. If *type* is `equation`, the rule applies to equation elements. If *type* is `facet`, the rule applies to a graphic element that contains only a single facet with the name specified by *facetname*. This occurs when the graphic element is an anchored frame containing only a single imported graphic object whose original file was in the *facetname* graphic format. You can use this rule with *type* set to `facet` multiple times if you want the software to treat several file formats differently. If *type* is `anchored frame`, the rule applies to a graphic element under all other circumstances.
facetname	A facet name. You supply this argument if and only if *type* is `facet`.
ename	A string representing the base name for an SGML entity name.

Details

- By default, when FrameMaker+SGML exports an external data entity for a graphic or equation, it uses the entity name that is stored with the graphic inset. If there is no such entity name, the software generates a name for the entity based on the element name. You use the `entity name is` rule to change this behavior.

  The entity name you specify is a base name FrameMaker+SGML uses to generate a unique entity name. When it needs to create a new entity name, the software adds an integer to the name specified by *ename* to create a unique name.

Examples

Assume you have an SGML element `graphic` that corresponds to graphic elements in FrameMaker+SGML. Suppose further that some of the graphic elements in FrameMaker+SGML contain imported-by-copy graphics, or contain modifications to a graphic inset using FrameMaker graphic tools, or contain just graphic objects drawn using FrameMaker graphic tools. On export, the software must create new graphic files for these elements and declare entities for them. By default, FrameMaker+SGML would declare

entities for these graphic elements based on the element name "graphic," for example, `graphic1`, `graphic2`, and so on. To specify that the names of the entities associated with such successive graphic elements have the form `car1`, `car2`, and so on, use this rule:

```
element "graphic" {
   is fm graphic element;
   writer anchored frame entity name is "car";
}
```

- Assume with a single facet graphics in the `car` element sometimes use the IGES file format and sometimes use the TIFF file format. Also assume that the DTD for the application does not currently contain entity declarations for the imported-by-reference graphic files. By default, the software would declare entities for all such graphics based on the element name "car," for example, `car1`, `car2`, and so on. If you want to name the entities for the IGES graphics `icar` and the entities for the TIFF graphics `tcar`, then use this rule:

```
element "car" {
   is fm graphic element;
   writer facet "IGES" entity name is "icar";
   writer facet "TIFF" entity name is "tcar";
}
```

See also

Related rules	“export to file” on page 359 “notation is” on page 420 “specify size in” on page 431
Rules mentioned in synopses	“element” on page 345 “is fm graphic element” on page 394 “is fm equation element” on page 392 “anchored frame” on page 333 “equation” on page 355 “facet” on page 364 “writer” on page 442
General information on this topic	Chapter 15, “Translating Graphics and Equations”

equation

Use the `equation` rule only in an `element` rule for an equation element, to provide information the software needs when writing to SGML a document containing equations. FrameMaker+SGML creates graphic files to represent equations. Use this rule to specify information about the files FrameMaker+SGML creates for instances of the equation element. By default, the software creates a file in CGM format for each equation, and the filename is based on the element name. Also, by default, if the equation element is associated with an external data entity, then the entity name is based on the element name.

Synopsis and contexts

```
element "gi" {
   is fm equation element ["fmtag"];
   writer equation subrule;
. . .}
```

Arguments

gi — An SGML generic identifier.

fmtag — A FrameMaker+SGML element tag.

subrule — An `equation` rule can have the following subrules:

`entity name is`, page 352, tells the software how to create the base name for the entity associated with this element type.

`export to file`, page 359, tells the software to write a new file for the associated external data entity.

`notation is`, page 420, specifies the data content notation of the entity file.

`specify size in`, page 431, specifies the units to use when writing the file.

Examples

- Assume you have an element named `math` with an SGML attribute of type `Entity` that is mapped to the `fm property` entity for this element. If you want to create TIFF files for the equations in a document named `mytest.doc`, you might use this rule:

  ```
  element "math" {
     is fm equation element;
     writer equation export to file "$(docname).eqn" as "TIFF";
  }
  ```

  The software will create graphic files for each equation in `mytest.doc` named `mytest1`, `mytest2`,...and will declare entities named `math1`, `math2`, ...for each graphic.

See also

Related rules	“anchored frame” on page 333 “facet” on page 364 “is fm graphic element” on page 394
Rules mentioned in synopses	“element” on page 345 “is fm equation element” on page 392 “writer” on page 442
General information on this topic	Chapter 15, “Translating Graphics and Equations”

export dpi is

You use the `export dpi` rule only in an `element` rule for a graphic or equation element, to provide information the software needs when writing a document containing graphics or equations to SGML. The `export dpi` rule tells FrameMaker+SGML the dpi setting to use for an exported graphic file.

Synopsis and contexts

```
element "gi" {
   is fm graphic_or_eqn element ["fmtag"];
   writer type ["facetname"]
      export dpi is  number;
      . . .
. . .}
```

Arguments

gi	An SGML generic identifier.
graphic_or_eqn	One of the keywords: `graphic` or `equation`.
type	One of the rules `anchored frame`, `facet`, or `equation`. If `facet`, you must also supply the *facetname* argument. If *type* is `equation`, the rule applies to equation elements. If *type* is `facet`, the rule applies to a graphic element that contains only a single facet with the name specified by *facetname*. This occurs when the graphic element is an anchored frame containing only a single imported graphic object whose original file was in the *facetname* graphic format. You can use this rule with *type* set to `facet` multiple times if you want the software to treat several file formats differently. If *type* is `anchored frame`, the rule applies to a graphic element under all other circumstances.
facetname	A facet name. You supply this argument if and only if *type* is `facet`.
number	The value for your dpi setting.

Details

- In the absence of this rule, FrameMaker+SGML uses the dpi setting associated with the graphic file. If there is no setting associated with the graphic, the software assumes a value of 300.

- For Windows versions of FrameMaker+SGML, if the initialization file for a graphics filter specifies a dpi setting, that setting overrides this rule whenever that filter is used to export a graphic file.

Examples

- Assume you export the FrameMaker+SGML file `Math.doc` and have the following rule:

```
element "eqn" {
   is fm equation element "Equation";
   writer equation
      export dpi is 72;
}
```

  When FrameMaker+SGML finds an instance of the `Equation` element, it exports equations as graphic files at 72 dpi.

- Assume you have the rule:

```
element "imp" {
   is fm graphic element;
   writer facet “TIFF”{
      convert referenced graphics;
      export dpi is 1200;
      export to file "$(entity).tif";
}}
```

  This rule tells FrameMaker+SGML for every graphic element with a single TIFF facet, it should write a new graphic file with a dpi of 1200, using the entity name as part of the graphic file’s filename.

See also

Related rules	“convert referenced graphics” on page 340
	“entity name is” on page 352
	“notation is” on page 420
	“specify size in” on page 431
Rules mentioned in synopses	“element” on page 345
	“is fm graphic element” on page 394
	“is fm equation element” on page 392
	“anchored frame” on page 333
	“equation” on page 355
	“facet” on page 364
	“writer” on page 442

General information on this topic — Chapter 15, “Translating Graphics and Equations”

export to file

You use the `export to file` rule only in an `element` rule for a graphic or equation element, to provide information the software needs when writing a document containing graphics or equations to SGML. The `export to file` rule tells FrameMaker+SGML how to write the file name when it creates a new graphic file, and optionally the graphic format for the file.

Synopsis and contexts

```
element "gi" {
   is fm graphic_or_eqn element ["fmtag"];
   writer type ["facetname"]
      export to file "fname" [as "format"];
. . .}
```

Arguments

gi — An SGML generic identifier.

graphic_or_eqn — One of the keywords: `graphic` or `equation`.

type — One of the rules `anchored frame`, `facet`, or `equation`. If `facet`, you must also supply the *facetname* argument.

If *type* is `equation`, the rule applies to equation elements.

If *type* is `facet`, the rule applies to a graphic element that contains only a single facet with the name specified by *facetname*. This occurs when the graphic element is an anchored frame containing only a single imported graphic object whose original file was in the *facetname* graphic format. In this case, the rule is only executed if the “converted referenced graphics” rule is also used. Otherwise, it is ignored. You can use this rule with *type* set to `facet` multiple times if you want the software to treat several file formats differently.

If *type* is `anchored frame`, the rule applies to a graphic element under all other circumstances and does not use the “convert referenced graphics” rule.

facetname — A facet name. You supply this argument if and only if *type* is `facet`. The string for the facetname must exactly match the string for the facetname in the FrameMaker+SGML document. To determine a graphic file’s facetname, select the graphic, click

Graphics>ObjectProperties, and observe the facetname in the dialog box.

fname — A base filename which can be either absolute or relative to the output directory. Note: If path information is included in fname, the Windows platform requires double backslashes to translate correctly as path backslashes. The *fname* argument can contain the variables `$(docname)` and `$(entity)`, described below.

format — A file data content format code, such as TIFF or PICT. See Chapter 15, "Translating Graphics and Equations," for a complete list of graphic format codes. `format` must be one of these code names.

Details

- By default, if a graphic element has a single facet (other than a FrameMaker+SGML internal facet) that was imported by reference, FrameMaker+SGML does not create a new graphic file. On export, the original file will be associated with an SGML graphic element via the `file` attribute, or via the `entity` attribute plus a corresponding entity declaration. You can use the `convert referenced graphics` rule to force FrameMaker+SGML to export such graphic files.
- If your rules specify the software will write a graphic file, if a graphic element has a single facet (other than a FrameMaker+SGML internal facet), FrameMaker+SGML writes the graphic file in that format by default. It writes the graphic file for equation elements and all other graphic elements in CGM format by default.

  If you supply a *format* argument, you must first make sure that the format is one known to FrameMaker+SGML. For information on which graphic export filters the software provides and on how to add new ones, see Chapter 15, "Translating Graphics and Equations".
- The *fname* argument can use these variables:

Variable	Meaning
`$(entity)`	The value of the corresponding SGML element's entity attribute. If the source of the graphic inset wasn't originally an SGML entity, this variable defaults to a unique name based on the name of the element. You can change this name using the "entity name is" rule.
`$(docname)`	The name of the FrameMaker+SGML file, excluding any extension or directory information.

- The *fname* argument is used as a template for the actual filename FrameMaker+SGML generates for a particular graphic or equation element. FrameMaker+SGML takes the filename specified with the *fname* argument and may append an integer to the filename to ensure uniqueness of the filename. For an example of this behavior, see the first example below.

Examples

- Assume you export the FrameMaker+SGML file `Math.doc` and have the following rule:

```
element "eqn" {
   is fm equation element "Equation";
   writer equation
      export to file "$(docname).eqn" as "PICT";
}
```

  When FrameMaker+SGML finds an instance of the `Equation` element, it generates filenames of the form `Math`*N*`.eqn` until it finds a name that doesn't collide with an already existing file. For example, if you already have files in the specified directory named `Math1.eqn` and `Math2.eqn`, the software writes the first equation to a file named `Math3.eqn`. FrameMaker+SGML writes the equation file in PICT format, instead of the default CGM format.

- Assume you have the rule:

```
element "imp" {
   is fm graphic element;
   writer facet "TIFF" {
   convert referenced graphics;
   export to file "$(entity).tif";
}}
```

  This rule tells FrameMaker+SGML that if it encounters a graphic element with an imported graphic file with a single TIFF facet, it should write that graphic to the file specified by `$(entity).tif`.

See also

Related rules	"convert referenced graphics" on page 340 "entity name is" on page 352 "notation is" on page 420 "specify size in" on page 431
Rules mentioned in synopses	"element" on page 345 "is fm graphic element" on page 394 "is fm equation element" on page 392 "anchored frame" on page 333 "equation" on page 355 "facet" on page 364 "writer" on page 442

General information on this topic — Chapter 15, “Translating Graphics and Equations”

external data entity reference

Use the `external data entity reference` rule to drop references to SGML external data entities. By default, FrameMaker+SGML stores such references as the marker text in non-element SGML Entity Reference markers.

Synopsis and contexts

```
external data entity reference drop;
```

Arguments

None.

Details

- 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+SGML treats general entity name attributes.
- When you translate an SGML document to FrameMaker+SGML, when the software encounters an external data entity reference such as:

  ```
  &door;
  ```

  it stores the reference as the text of a non-element SGML Entity Reference marker by default, with the following marker text:

  ```
  door
  ```

  When you translate a FrameMaker+SGML document to SGML, it outputs the marker text of non-element SGML Entity Reference markers as entity references.

Examples

- To discard all external data entity references, use this rule:

  ```
  external data entity reference drop;
  ```

See also

Rules mentioned in synopses — “drop” on page 342

General information on this topic — Chapter 13, “Translating Entities and Processing Instructions”

external dtd

Use this rule to specify how an exported SGML instance refers to the current SGML application's DTD. By default, FrameMaker+SGML uses the name of the file containing the DTD as the system identifier in the external identifier for the DTD. The `external dtd` rule provides the software with a different external identifier. The different forms of the rule allow specification of a system identifier, public identifier, or both.

Synopsis and contexts

1. `writer` **`external dtd`** `is system;`
2. `writer` **`external dtd`** `is system "`*`sysid`*`";`
3. `writer` **`external dtd`** `is public "`*`pubid`*`";`
4. `writer` **`external dtd`** `is public "`*`pubid`*`" "`*`sysid`*`";`

Arguments

sysid	A system identifier.
pubid	A public identifier.

Details

- To use this rule, you must have a DTD specified for the current SGML application. You define an SGML application in the `sgmlapps.fm` file.
- Use this rule when you export FrameMaker+SGML documents to SGML documents.
- By default, FrameMaker+SGML does not reproduce the DTD in the document type declaration subset. Instead, it uses the filename of the DTD as the system identifier in an external identifier in a document type declaration of the form:

  `<!DOCTYPE` *`doctype`* `SYSTEM "`*`fname`*`" [. . .`

  where *`doctype`* is the document type name and *`fname`* is the name of the file containing the DTD. This rule allows you to specify different system and public identifiers.
- You cannot use the `external dtd` rule in the same read/write rules file as the `include dtd` rule.

Examples

- To specify a local DTD as an external DTD and include the path with the filename, you could use this rule:

  ```
  writer
    external dtd is
       system "/doc/dtds/manuals.dtd";
  ```

  Note that path information is system specific. The Windows platform requires two backslashes in paths in the rules file in order to translate as one backslash.

- To specify and locate the CALS DTD as an external DTD, you could use this rule:

```
writer external dtd is
   public "-//USA-DOD//DTD MIL-M-38784B//EN"
   "/doc/dtds/cals.dtd";
```

- To specify just the CALS DTD as an external DTD using a public identifier, you could use this rule:

```
writer external dtd is
   public “-//USA-DOD//DTD MIL-M-38784B//EN”;
```

You could then specify the location of the DTD in the SGML application using the `EntitiesLocation` element. A DTD is an entity in the strictest sense.

See also

Related rules	“include dtd” on page 379
	“include sgml declaration” on page 380
	“write sgml document” on page 441
	“write sgml document instance only” on page 441
Rules mentioned in synopses	“writer” on page 442

facet

Use the `facet` rule only in an `element` rule for a graphic element, to provide information the software needs when writing a document containing graphics to SGML. The `facet` rule applies only when a graphic element is an anchored frame containing only a single imported graphic object whose original file was in the *`facetname`* graphic format. Use this rule to specify information about the graphic file and/or entity declaration for instances of the graphic element.

Synopsis and contexts

```
element "gi" {
   is fm graphic element ["fmtag"];
   writer facet "facetname" subrule;
. . .}
```

Arguments

`gi`	An SGML generic identifier.
`fmtag`	A FrameMaker+SGML element tag.
`facetname`	The name of the particular facet to which this rule applies.

subrule

A `facet` rule can have the following subrules:

`convert referenced graphics`, page 340, tells the software to create new graphic files for imported graphic files with a single facet.

`entity name is`, page 352, tells the software how to create the base name for the entity associated with this element type.

`export to file`, page 359, tells the software the name to use for graphics it creates, and optionally, the graphic format to which it should convert.

`notation is`, page 420, specifies the data content notation of the entity.

`specify size in`, page 431, specifies the units to use when writing the file.

Details

To specify all facets, use the keyword default for the facetname argument. For example:

```
element "pict" {
   is fm graphic element "Picture";
   writer {
      facet default {
      convert referenced graphics;
      export to file "$(entity).tif" as "TIFF";
      . . .
}}}
```

will convert every imported graphic file in the document to a TIFF file, no matter what its original facet was.

Examples

- By default, FrameMaker+SGML does not create a new graphic file for a graphic element that originated as an SGML external entity, and was not modified by the user in any way. Assume you want the software to generate a graphic file for every imported TIFF file, whether it was modified or not. Then you could use this rule:

```
element "pict" {
   is fm graphic element "Picture";
   writer {
      facet "TIFF" {
         convert referenced graphics;
         export to file "$(entity).tif" as "TIFF";
}}}
```

See also

Related rules	“anchored frame” on page 333 “convert referenced graphics” on page 340 “equation” on page 355
Rules mentioned in synopses	“element” on page 345 “is fm equation element” on page 392 “is fm graphic element” on page 394 “writer” on page 442
General information on this topic	Chapter 15, “Translating Graphics and Equations”

fm attribute

You use the `fm attribute` rule with the “drop” subrule to discard an attribute that you’ve defined for a FrameMaker+SGML element but that does not exist on the corresponding SGML element.

Synopsis and contexts

1. **fm attribute** "*attr*" drop;

2. element "*gi*" {. . .
   fm attribute "*attr*" drop;
   . . .}

Arguments

attr	A FrameMaker+SGML attribute name.
gi	An SGML generic identifier.

Examples

- Assume the element `chapter` exists in both the SGML and FrameMaker+SGML representations of your documents. In FrameMaker+SGML, you use the `XRefLabel` attribute in formatting cross-references to this element. Since this attribute exists only for formatting purposes, you don’t want it in the SGML document. To drop this attribute on export, use this rule:

```
element "chapter" {
   is fm element;
   fm attribute "XRefLabel" drop;
}
```

- If you use the `XRefLabel` attribute on many elements for the same purpose, you can discard it from all elements on export with this rule:

```
fm attribute "XRefLabel" drop;
```

- If you want to keep the `XRefLabel` attribute on the `appendix` element, but drop it from all others, use these rules:

```
element "appendix" {
   is fm element;
   attribute "xreflab" is fm attribute "XRefLabel";
}
fm attribute "XRefLabel" drop;
```

  Note that the order of these rules is not important. If you reversed them, the `XRefLabel` attribute would still be correctly interpreted for the `appendix` element, since that reference to the attribute is more specific. Note also that case is sensitive for fm attribute names.

See also

Related rules	"attribute" on page 335 "is fm attribute" on page 385
Rules mentioned in synopses	"element" on page 345 "drop" on page 342
General information on this topic	Chapter 12, "Translating Elements and Their Attributes"

fm element

Use the `fm element` rule to tell FrameMaker+SGML what to do on export with FrameMaker+SGML elements that do not correspond to SGML elements.

Synopsis and contexts

1. **fm element** "*fmtag*" drop;
2. **fm element** "*fmtag*" unwrap;

Arguments

fmtag	A FrameMaker+SGML element tag.

Details

- Use this rule when you export FrameMaker+SGML documents to SGML documents.
- If you use this rule, you may want to write an SGML API client to handle the export of the element or to create it on import.

- The first version of this rule discards the FrameMaker+SGML element on export. The second version inserts the contents of *fmtag* in the corresponding SGML document, but not *fmtag* itself.
- If you use this rule to unwrap FrameMaker+SGML cross-reference elements or system variable elements, those elements become text in the resulting SGML document.

Examples

- If `Chapter Number` is a FrameMaker+SGML element that you want to discard on export, use this rule:

  ```
  fm element "Chapter Number" drop;
  ```

  If you use this rule and want to create this element on import, you need to write an SGML API client.

- If `Modification Date` is a FrameMaker+SGML system variable element that you wish to translate to text on export to SGML, use this rule:

  ```
  fm element "Modification Date" unwrap;
  ```

See also

Related rules	"element" on page 345 "is fm element" on page 391
Rules mentioned in synopses	"drop" on page 342 "unwrap" on page 436
General information on this topic	Chapter 12, "Translating Elements and Their Attributes"

fm marker

On export, you use the `fm marker` rule to tell FrameMaker+SGML what to do with non-element markers other than markers of the type reserved for storing SGML processing instructions, PI entities, and external data entities. (By default, SGML PI markers are reserved for processing instructions, and SGML Entity Reference markers are reserved for external data entities.) In the absence of a rule to the contrary, the software creates SGML processing instructions for non-element markers. You can also choose to discard them.

Synopsis and contexts

```
fm marker ["type1", . . ., "typen"] drop;
fm marker ["type1", . . ., "typen"] is processing instruction;
```

Arguments

$type_i$	The name of a FrameMaker+SGML marker type.

Details

- If *type*$_i$ is specified, this rule applies only to markers of that type.

  If no *type*$_i$ is specified, this rule applies to all non-element markers other than markers of the reserved type. For information on what the software does with the reserved marker type, see Chapter 18, “Translating Markers.”

- You can have multiple occurrences of this rule in a rules file, to determine different treatment for different FrameMaker+SGML markers. You can have only one occurrence of the rule with no explicitly listed markers. A given marker type can be explicitly mentioned in only one occurrence of this rule.

- The order of `fm marker` rules is not important. A more specific occurrence of the rule always takes precedence over a more general occurrence. For example, the following rules:

  ```
  fm marker "Index" is processing instruction;
  fm marker drop;
  ```

  have the same effect, exporting only index markers as processing instructions, if they occur in this order:

  ```
  fm marker drop;
  fm marker "Index" is processing instruction;
  ```

Examples

- To discard all non-element markers, use this rule:

  ```
  fm marker drop;
  ```

- To discard non-element conditional text markers but retain all others as processing instructions, use this rule:

  ```
  fm marker "Conditional Text" drop;
  ```

- To retain only Index and Hypertext markers as processing instructions and drop all other non-element markers, use the following set of rules:

  ```
  fm marker "Index", "Hypertext" is processing instruction;
  fm marker drop;
  ```

See also

Related rules	“is fm marker element” on page 395
Rules mentioned in synopses	“drop” on page 342 “is processing instruction” on page 415
General information on this topic	Chapter 18, “Translating Markers”

fm property

You use the `fm property` rule to determine values for properties defined for certain types of FrameMaker+SGML constructs that you do not want to represent as SGML attributes.

Synopsis and contexts

```
1. fm property prop value is "val";

2. element "gi" {
      is fm type ["fmtag"];
      fm property prop value is val;.
   . . .}
```

Arguments

prop A FrameMaker+SGML property. Possible properties are:

- For cross-reference elements: `cross-reference format`.
- For graphic and equation elements: `entity`, `dpi`, `import size`, `import by reference or copy`, `sideways`, `import angle`, `horizontal offset`, `vertical offset`, `position`, `baseline offset`, `near-side offset`, `alignment`, `cropped`, `floating`, `angle`, `width`, or `height`.
- For marker elements: `marker type` or `marker text`.
- For table elements: `column ruling`, `column widths`, `columns`, `page wide`, `row ruling`, `table border ruling`, or `table format`.
- For table cell elements: `column name`, `column number`, `column ruling`, `end column name`, `horizontal straddle`, `more rows`, `rotate`, `row ruling`, `span name`, `start column name`, or `vertical straddle`.
- For table row elements: `maximum height`, `minimum height`, `row type`, or `row ruling`.
- For CALS table colspecs: `cell alignment character`, `cell alignment offset`, `cell alignment type`, `column name`, `column number`, `column ruling`, `column width`, `row ruling`, or `vertical alignment`.
- For CALS table spanspecs: `cell alignment character`, `cell alignment offset`, `cell alignment type`, `column ruling`, `end column name`, `row ruling`, `span name`, `start column name`, or `vertical alignment`.

gi An SGML generic identifier.

fmtag	A FrameMaker+SGML element tag.
type	One of the following: `cross-reference element`, `graphic element`, `equation element`, `marker element`, `table element`, `table row element`, `table cell element`, `colspec`, or `spanspec`.
val	The keyword `default` or a string containing a legal value for the formatting property.

Details

- This rule applies only to an element corresponding to a cross-reference, graphic, equation, marker, table, or table part element.
- Some FrameMaker+SGML properties have no natural SGML counterparts. If you choose to not translate such properties as SGML attributes, an SGML document will not contain information on appropriate values for these properties. In this situation, you can use the `fm property` rule to explicitly set property values when reading an SGML document.
- This rule can be used either at the highest level to set a default or within an `element` rule to specify the translation of a property for a particular element.
- If you use this rule to set a property value explicitly, you cannot also have an SGML attribute that corresponds to this property. For example, the following rule is erroneous:

```
element "tab2" {
   is fm table element;
   attribute "w" is fm property column widths;
   fm property column widths value is "1in 2in";
}
```

Examples

- To translate the SGML element `table` to a FrameMaker+SGML table with two columns:

```
element "table" {
   is fm table element;
   fm property columns value is "2";
}
```

  On import to FrameMaker+SGML, the software creates the table as a 2-column table in FrameMaker+SGML.

- Assume you have an SGML element `halfpage` that holds a 4.5 inch by 6.5 inch graphic object; it doesn't use an attribute to store the size information. You can translate this to a FrameMaker+SGML graphic as follows:

```
element "halfpage" {
   is fm graphic element;
   fm property width value is "6.5";
   fm property height value is "4.5";
}
```

See also

Related rules	"is fm property" on page 396
	"is fm property value" on page 398
General information on this topic	Chapter 15, "Translating Graphics and Equations"
	Chapter 14, "Translating Tables"
	Chapter 16, "Translating Cross-References"
	Chapter 18, "Translating Markers"

fm variable

On export, use the `fm variable` rule to tell FrameMaker+SGML what to do with certain variables. Use this rule if you do not want them translated to SGML entities.

Synopsis and contexts

fm variable ["var_1", . . ., "var_n"] drop;

Arguments

var_i	The name of a FrameMaker+SGML variable.

Details

- Use this rule when you export FrameMaker+SGML documents to SGML documents. It applies only to non-element variables, not to system variable elements.
- If var_i is specified, this rule applies only to that variable. If no var_i is specified, this rule applies to all variables.
- If you use this rule, you may want to write an SGML API client to handle the export of variables or to create variables on import.
- You can have multiple occurrences of this rule in a rules document to determine different treatment for different FrameMaker+SGML variables. You can have only one occurrence of the rule with no explicitly listed variables. A given variable can be explicitly mentioned in only one occurrence of this rule.

Examples

- To translate the FrameMaker+SGML variables `Licensor` and `Product` as entities and discard all other variables, use these rules:

```
entity "licensor" is fm variable;
entity "product" is fm variable;
fm variable drop;
```

See also

Related rules	"is fm system variable element" on page 407
General information on this topic	Chapter 13, "Translating Entities and Processing Instructions" Chapter 17, "Translating Variables and System Variable Elements" *SGML API Programmer's Guide*

fmsgml version

The `fmsgml version` rule specifies the version of the product being run. It is required and must be the first rule in all rules documents. If you create your rules document with the New Read/Write Rules command, this rule automatically appears in the document.

Synopsis and contexts

```
FMSGML version is "6.0";
```

Arguments

None.

Details

Note that you would use the string `"5.0"` in this rule even though the product version may be an incremental release above 5.0, such as 5.1.

See also

General information on this topic	Chapter 11, "SGML Read/Write Rules and Their Syntax"

generate book

Use the `generate book` subrule of a highest-level `reader` rule to specify whether FrameMaker+SGML should use elements or processing instructions to indicate where in an SGML document to start a book and its components in the corresponding FrameMaker+SGML book.

Synopsis and contexts

```
1. reader generate book
      use processing instructions;

2. reader generate book
   {
      put element "gi1" in file ["fname1"];
      . . .
      put element "giM" in file ["fnameM"];
   }

3. reader generate book [for doctype "dt1" [, . . . "dtN"]]
   {
      put element "gi1" in file ["fname1"];
      . . .
      put element "giM" in file ["fnameM"];
   }
```

Arguments

dt_i	An SGML document type name.
gi_j	An SGML generic identifier.
$fname_j$	A filename for the book component. FrameMaker+SGML adds a counter to the name (before the suffix if there is one) as needed, to generate a unique filename. You can use the `$(bookname)` variable to base the component's filename on the book filename (excluding any suffix). If you do not supply this argument, the filename is gi_j`.doc`.

Details

- By default, when reading an SGML document into FrameMaker+SGML, the software uses the `?FM: book` and `?FM: document` processing instructions to indicate the start of a book and of its components. The following rule confirms this default behavior:

  ```
  reader generate book
     use processing instructions;
  ```

- Your DTD may be defined so that you can use elements to indicate the start of a book and its components. When you use the second form of the `generate book` rule, FrameMaker+SGML creates a book for every SGML document you translate. When you use the third form of the `generate book` rule, it creates a book only for SGML documents whose DTD specifies the document type you've listed in the rule. If you have an SGML document with a different document type, FrameMaker+SGML translates that

document as a single FrameMaker+SGML document, even if it contains elements referenced in `put element` rules. For example, assume you have this rule:

```
reader generate book for doctype "manual"
   put element "chapter" in file;
```

If you translate an SGML document whose highest-level element is `report`, that document becomes a single FrameMaker+SGML document, even if it contains `chapter` descendant elements.

- When it encounters one of the gi_j elements specified in a `put element` subrule, FrameMaker+SGML starts a new book component. Since the software does not allow an element to be broken across files, it places the entire gi_j element in the same file, even if another element appears that you've said should start a new file. To illustrate, assume the `section` element can occur either within or outside of a `chapter` element and you have this rule:

  ```
  reader generate book {
     put element "chapter" in file;
     put element "section" in file;
  }
  ```

  When FrameMaker+SGML encounters a `chapter` element, it starts a new file. If it encounters a `section` element as a child of that `chapter` element, it does not start a new file. It continues with the file started by the `chapter` element. On the other hand, if the software encounters a `section` element outside a `chapter` element or within another `section` element, it does start a new file for it.

- There are a couple of points to consider when dividing an SGML document into book components:
  - Every FrameMaker+SGML document must contain exactly one highest-level element. That is, there cannot be two elements in a single file that do not have an ancestor element in the same file.
  - A book element can contain substructure but cannot directly contain text. That is, child elements that can contain text must occur in separate files.

  Assume you have this rule:

  ```
  reader generate book
     put element "chapter" in file;
  ```

And you have an SGML document with the following element structure:

```
<manual>
<chapter>
<head>Introduction</head>
. . .
</chapter>
<appendix>
<head>The final word</head>
. . .
</appendix>
</manual>
```

When FrameMaker+SGML translates this document, it creates a book with `manual` as the highest-level element in the book file. When it encounters the `chapter` element, the software starts a new file for that element. When it encounters the `appendix` element, FrameMaker+SGML flags an error, because your rules have not told it what to do with this element. It cannot put the element in the same file as the preceding `chapter` element, because that would create two highest-level elements in the same file. It also cannot put the `appendix` element in the book file, because it contains text.

- By default, when it writes a FrameMaker+SGML book to SGML, the software writes `?FM: book` and `?FM: document` processing instructions for the book and book components. It does this even if you use the `generate book` rule to have particular elements specify book components when reading an SGML document. If you do not want FrameMaker+SGML to output these processing instructions, use the `do not output book processing instructions`, which is described on page 422.

Examples

- If you know that an SGML document should always correspond to a FrameMaker+SGML book and that individual files in the book should start when the document reaches a `toc` or `chapter` element, you can use this rule:

```
reader generate book {
   put element "toc" in file;
   put element "chapter" in file "ch.doc";
}
```

With this rule, FrameMaker+SGML creates a book for each SGML document. In an SGML document, FrameMaker+SGML starts a new book component when it encounters a `toc` or `chapter` element. For the first `toc` element, FrameMaker+SGML uses the filename `toc1` unless a file of that name already exists in the directory it is using. It continues that book component until it encounters either another `toc` element or a `chapter` element. At that point, it starts a new book component. It tries to put the first `chapter` element in a file called `ch1.doc`.

- Assume that an SGML document whose highest-level element is either `manual` or `book` should correspond to a FrameMaker+SGML book and any other SGML document should correspond to an individual FrameMaker+SGML document. Further assume that the books created from `manual` and `book` elements should have new files for each instance of the elements `chapter`, `front`, or `toc`. To accomplish all this, you can use this rule:

```
reader generate book for doctype "manual", "book"
{
   put element "chapter" in file "ch.doc";
   put element "front" in file;
   put element "toc" in file "$(bookname).toc";
}
```

  With this rule, FrameMaker+SGML asks you for a name for the book file if you open an SGML document with `manual` as its document type. If you specify `myfile.bk` as its name, and the document contains two `chapter` elements, one `front` element, and one `toc` element, FrameMaker+SGML creates the following files: `myfile.bk`, `ch1.doc`, `ch2.doc`, `front`, and `myfile.toc`.

See also

Related rules	"output book processing instructions" on page 422
General information on this topic	Chapter 19, "Processing Multiple Files as Books"

implied value is

Use the `implied value is` rule to specify default attribute values in your EDD to correspond with imported elements that specify no value for the attribute. For example, assume your DTD declares an element named `list`, which has an attribute named `style` defined as `<!ATTLIST list style (bul | num) #IMPLIED>`. For importing the DTD, you can use this rule to set up a default value in the EDD for the `Style` attribute of the `List` element. Then, if you import a `list` element that has no value for style, this default attribute value will be used for formatting purposes. Also, when you export the EDD, the DTD will declare the `style` attribute for the `list` element as `#IMPLIED`.

Synopsis and contexts

```
1. attribute "attr" {. . .
      implied value is "val";
   . . .}

2. element "gi" {. . .
      attribute "attr" {. . .
         implied value is "val";
   . . .} . . .}
```

Arguments

attr	The name of an SGML impliable attribute.
val	A value to use for the *attr* attribute.
gi	An SGML generic identifier.

Details

- This rule is for importing DTDs and exporting EDDs. In FrameMaker+SGML, a default attribute value can only be specified in the EDD, so this rule has no effect when importing an SGML instance or exporting a FrameMaker+SGML document.
- This rule specifically does not supply an attribute value for an element that has no value in the SGML instance. It only sets up a default attribute value in the EDD. This default value can be used for formatting by attributes. When you export the document, FrameMaker+SGML will not add a value for the attribute to the element's start tag.
- The rule can be used in a highest-level `attribute` rule to specify the value to use for that attribute in any element. Alternatively, it can be used in an `attribute` rule within an `element` rule to specify the value for that element only.

Examples

- Assume you have these declarations for an SGML element used for cross-references:

  ```
  <!ELEMENT xref - o EMPTY>
  <!ATTLIST xref
      id IDREF #IMPLIED
      format CDATA #IMPLIED>
  ```

  And you have this rule:

  ```
  element "xref" {
     is fm cross-reference element;
     attribute "format" {
        is fm property cross-reference format;
        implied value is "Page";
  }}
  ```

  When FrameMaker+SGML encounters an instance of the `xref` element in an SGML document and that instance doesn't have a value for the `format` attribute, the software use the Page cross-reference format for the cross-reference in the FrameMaker+SGML document.

See also

Related rules	"value" on page 439
Rules mentioned in synopses	"attribute" on page 335 "element" on page 345
General information on this topic	Chapter 12, "Translating Elements and Their Attributes" "Default value" on page 164

include dtd

By default, when creating an SGML document, FrameMaker+SGML includes in the document type definition an external identifier that refers to the DTD file. Therefore, it does not include a copy of actual declarations in the document type declaration subset. The `include dtd` rule tells FrameMaker+SGML to do so.

Synopsis and contexts

```
writer [do not] include dtd;
```

Arguments

None.

Details

- You use this rule when you export FrameMaker+SGML documents to SGML documents. If this rule is specified, FrameMaker+SGML does not generate an external identifier in the `DOCTYPE` declaration.
- To confirm the default behavior, you can use the opposite rule:

  ```
  writer do not include dtd;
  ```

- The `include dtd` rule and the `external dtd` rule are mutually exclusive. That is, you cannot use both of these rules in the same read/write rules file. (If you try to put both of these rules in the same file, you will get an alert.) Also, the `include dtd` rule and the `write sgml document instance only` rule are mutually exclusive.
- To write an entire SGML document, including an SGML DTD and SGML declaration with the document instance, you must use the following rules:

  ```
  writer {
     include sgml declaration;
     include dtd;
  }
  ```

Examples

- If your document type declarations are in a file called `report.dtd`, then by default FrameMaker+SGML includes this document type declaration in the document it creates on export:

```
<!DOCTYPE report SYSTEM "report.dtd" [
. . . more declarations specific to this document instance . . .
]>
```

  If you specify the `include dtd` rule, then FrameMaker+SGML includes this document type declaration in the document it creates:

```
<!DOCTYPE report [
. . . contents of the file report.dtd . . .
. . . more declarations specific to this document instance . . .
]>
```

See also

Related rules

"external dtd" on page 363
"include sgml declaration," next
"write sgml document" on page 441
"write sgml document instance only" on page 441

include sgml declaration

By default, FrameMaker+SGML does not include an SGML declaration in a generated SGML document. The `sgml declaration` rule tells FrameMaker+SGML to include one. The SGML declaration is copied from the file in the associated application subset. To see the default SGML declaration used by FrameMaker+SGML, see Appendix D, "SGML Declaration."

Synopsis and contexts

```
writer [do not] include sgml declaration;
```

Arguments

None.

Details

- To confirm the default behavior, you can use the opposite rule:

```
writer do not include sgml declaration;
```

- You cannot use the `include sgml declaration` rule in the same read/write rules file as the `write sgml document instance only` rule. Note that using both rules in the same rules file doesn't give an error. Also, "write sgml document instance only" takes priority, regardless of order.

- To write an entire SGML document, including an SGML DTD and SGML declaration with the document instance, you must use the following rules:

```
writer {
   include sgml declaration;
   include dtd;
}
```

See also

Related rules

"external dtd" on page 363
"include dtd," (the previous section)
"write sgml document" on page 441
"write sgml document instance only" on page 441

insert table part element

You use the `insert table` *`part`* `element` rule when creating a FrameMaker+SGML table element on import of an SGML document. This rule tells FrameMaker+SGML to create a table part of the designated type, even if the SGML document does not contain content for that table part.

Synopsis and contexts

```
element "gi" {. . .
   is fm table element ["fmtag1"];
   reader insert table part element ["fmtag2"];
. . .}
```

Arguments

gi	An SGML generic identifier.
$fmtag_1$	A FrameMaker+SGML element tag for a table element.
part	One of the keywords: `title`, `heading`, or `footing`.
$fmtag_2$	A FrameMaker+SGML element tag for a table part element.

Details

- By default, as the last step in creating a table element when reading an SGML document, FrameMaker+SGML discards parts of the table that have no content, even if the general rule for the element requires that table part. (Your EDD may supply the content, for example, by using format rules that specify a prefix for the element.) If you do not want FrameMaker+SGML to remove the table part element with no content, OR if you want FrameMaker+SGML to create a table part element for you when the SGML instance does not contain this element, use the `insert table` *`part`* `element` rule.

Examples

- Assume you have an SGML element `statetab`, which you represent as a 3-column table in FrameMaker+SGML, with the same table headings everywhere it occurs. You use formatting rules in the EDD to specify the table headings. In this situation, the SGML document does not include information that corresponds to the table headings, so you want the software to add the table heading element when reading such an SGML instance and drop it when exporting a FrameMaker+SGML document to SGML. Suppose your DTD has these SGML declarations:

```
<!ELEMENT statetab - - ((state, pop, income)+)>
<!ELEMENT state - - (#PCDATA)>
<!ELEMENT pop - - (#PCDATA)>
<!ELEMENT income - - (#PCDATA)>
```

and your EDD has these FrameMaker+SGML element definitions:

Element (Table): State Table
General rule: State Head, State Body
Text format rules
1. In all contexts.
Use paragraph format: TableCell

Element (Table Heading): State Head
General rule: State Head Row
Text format rules
1. In all contexts.
Default font properties
Weight: Bold

Element (Table Row): State Head Row
General rule: Label

Element (Table Cell): Label
General rule: <EMPTY>
Text format rules
1. If context is: {first}
Numbering properties
Autonumber format: State
Else if context is: {last}
Numbering properties
Autonumber format: Household Income
Else
Numbering properties
Autonumber format: Population

Element (Table Body): State Body
General rule: State Row+

Element (Table Row): State Row
General rule: State, Income, Population

Element (Table Cell): State
General rule: <TEXT>

Element (Table Cell): Income
General rule: <TEXT>

Element (Table Cell): Population
General rule: <TEXT>

Note that the Label element provides the text for the column headings.

You could use these rules:

```
element "statetab" {
   is fm table element "State Table";
   fm property columns value is "3";
   reader insert table heading element "State Head";
}

element "state" {
   is fm table cell element;
   fm property column number value is "1";
   fm property row type value is "Body";
}

element "income" is fm table cell element;

element "pop" is fm table cell element "Population";

fm element "State Head" drop;
fm element "State Body" unwrap;
fm element "State Row" unwrap;
```

To convert the following SGML instance to the desired FrameMaker+SGML document:

```
<statetab>
<state>Georgia</state><pop>15,000,000</pop><income>25,000</
income>
<state>Mississippi</state><pop>8,000,000</pop><income>18,000</
income>
</statetab>
```

The first rule identifies `statetab` as a 3-column table element and tells it to always create a heading element for an occurrence of this `statetab`.

The second rule identifies `state` as a table cell that must always occur in the first column of a body row. This ensures that FrameMaker+SGML starts a new table row whenever it encounters a `state` element.

The other element rules identify other elements used as table cells. The `fm element drop` rule causes the software to drop the element that was created by the software per the `insert element` rule so that it does not appear in the SGML. Note also that it is necessary for the software to have a `tablerow` element and a `tablebody` element in its table structure. However, these do not appear in the SGML document. The software creates such necessary elements by default.Since they do not correspond to SGML elements, they are unwrapped on export to SGML--not dropped, because they would lose the contents of the entire table.

See also

General information on this topic Chapter 14, “Translating Tables”

is fm attribute

Use the `is fm attribute` rule to specify that an SGML attribute translates to a FrameMaker+SGML attribute. The optional parts of this rule allow you to have the software make several changes to the attribute during translation.

Synopsis and contexts

```
1. [sgmldv] attribute "sgmlattr" {. . .
      is fm [read-only] [fmtype] attribute
         ["fmattr"] [range from low to high];
   . . .}

2. element "gi" {. . .
      [sgmldv] attribute "sgmlattr"
         is fm [read-only] [fmtype] attribute
            ["fmattr"] [range from low to high];
   . . .}
```

Arguments

sgmldv	An optional SGML declared value. Legal values are: `cdata`, `name`, `names`, `nmtoken`, `nmtokens`, `number`, `numbers`, `nutoken`, `nutokens`, `entity`, `entities`, `notation`, `id`, `idref`, `idrefs`, and `group`.
sgmlattr	An SGML attribute name.
fmtype	A FrameMaker+SGML attribute type. Legal values are: String, Strings, Integer, Integers, Real, Reals, UniqueID, IDReference, IDReferences, and Choice.
fmattr	A FrameMaker+SGML attribute name.
low	A number, indicating the low end of a numeric range.
high	A number, indicating the high end of a numeric range.

Details

- You can use the `is fm attribute` rule in a highest-level `attribute` rule to specify the translation of that attribute in all elements for which it is defined. Or you can use it in an `attribute` subrule in an `element` rule to specify the translation of the `attribute` in only that element.
- You may want some SGML attributes to become FrameMaker+SGML properties. If so, you cannot also import them as FrameMaker+SGML attributes. For information on the defined FrameMaker+SGML properties, see "is fm property" on page 396.

- To specify only that the attribute is an attribute in both representations, use this version:

  ```
  attribute "sgmlattr" is fm attribute;
  ```

- To also rename it during translation, use this version:

  ```
  attribute "sgmlattr" is fm attribute "fmattr";
  ```

- To specify that the FrameMaker+SGML attribute is read-only—that is, that an end user cannot change the attribute's value—use this version:

  ```
  attribute "sgmlattr" is fm read-only attribute;
  ```

- To specify that an attribute that takes numeric values can have values only in a particular range, use this version:

  ```
  attribute "sgmlattr" is fm attribute range from low to high;
  ```

- To specify that an SGML attribute with a particular declared value translates to a FrameMaker+SGML attribute of a type other than the default translation, use this version:

  ```
  sgmldv attribute "sgmlattr" is fm fmtype attribute;
  ```

- Note that you can use more than one of the optional pieces of the `is fm attribute` rule at the same time. For example, you can both rename an attribute and state that it is read-only by using this version:

  ```
  attribute "sgmlattr" is fm read-only attribute "fmattr";
  ```

Examples

- To translate the SGML `sec` attribute to the FrameMaker+SGML `SecurityRanking` attribute in all elements in which it occurs, use this rule:

  ```
  attribute "sec" is fm attribute "SecurityRanking";
  ```

- To translate the SGML `sec` attribute to the FrameMaker+SGML `SecurityRanking` attribute in most elements in which it occurs, but to change it to the `Section` attribute in the `BookPart` element, use these rules:

  ```
  element "BookPart"
     attribute "sec" is fm attribute "Section";

  attribute "sec" is fm attribute "SecurityRanking";
  ```

- Assume you have an SGML attribute named `perc` with a declared value of `CDATA`, and assume you know that this attribute always has values that are integers in the range from 0 to 100. You can translate the `perc` attribute to the `Percentage` attribute with this rule:

  ```
  cdata attribute "perc"
     is fm integer attribute "Percentage" range from 0 to 100;
  ```

- Assume that an SGML element has an attribute with declared value `name` and that the attribute has a defined set of allowable values. You can translate that attribute and some of its possible values with the following rule:

```
element "fish" {
   name attribute "loc" {
      is fm choice attribute "CommonLocation";
      value "micro" is fm value "Micronesia";
      value "galap" is fm value "Galapagos Islands";
      value "png" is fm value "Papua New Guinea";
}}
```

See also

Related rules	"fm attribute" on page 366
Rules mentioned in synopses	"attribute" on page 335 "element" on page 345
General information on this topic	Chapter 12, "Translating Elements and Their Attributes"

is fm char

Use the `is fm char` rule to translate an SGML `SDATA` entity to a single character in FrameMaker+SGML.

Synopsis and contexts

1. `entity "`*`ename`*`"` **`is fm char`** *`ch`* `[in "`*`fmchartag`*`"];`
2. `reader entity "`*`ename`*`"` **`is fm char`** *`ch`* `[in "`*`fmchartag`*`"];`

Arguments

`ename`	An SGML entity name.
`ch`	A one-character string or a numeric character code (specified using the syntax for an octal, hexadecimal, or decimal number described in "Strings and constants" on page 200). Note that if the desired character is a digit or a white-space character, you must enter it as a numeric character code.
`fmchartag`	A FrameMaker+SGML character format tag. Note that the character format must use a non-standard font family such as Symbol or Zapf Dingbats for this argument to take effect.

Details

- Instead of using this rule to translate an SDATA entity, you can use a parameter literal of a particular form. For information on how to do so, see "Translating SDATA entities as special characters in FrameMaker+SGML" on page 237.
- You can use the `is fm char` rule within an `entity` rule at the highest level to have the translation occur in both directions. Or you can put the `entity` rule inside a `reader` rule to have the translation occur only when reading an SGML document into FrameMaker+SGML. For example, your SGML document might use a `period` entity for entering some instances of the period character in your SGML document. If you use this rule:

  ```
  entity "period" is fm char ".";
  ```

  then the entity references for period in the instance are translated correctly to the period character in FrameMaker+SGML. But on export, all periods in the document become references to the `period` entity (which is not likely what you had in mind). To have the period entities read correctly when importing an instance, but have periods remain the period character on export, use this version of the rule:

  ```
  reader
     entity "period" is fm char ".";
  ```

- Without the `in` clause, the software translates the entity using the default character format of the enclosing paragraph element. Frequently, however, special characters require a font change. In these cases, you use the `in` clause.
- 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+SGML includes copies of these entity sets and provides rules to handle them for your application. For information on how FrameMaker+SGML supports ISO public entities, see Appendix F, "ISO Public Entities."

Examples

- To translate the SDATA entity `sum` as the mathematical summation sign in the Symbol font (Σ), you could use either of these rules in your rules document:

  ```
  entity "sum" is fm char "S" in "Symbol";
  ```

  ```
  entity "sum" is fm char "\x53" in "Symbol";
  ```

  ```
  entity "sum" is fm char 0x53 in "Symbol";
  ```

  If FrameMaker+SGML encounters a reference to the `summation` entity when importing an SGML document, it replaces the reference with Σ (assuming your FrameMaker+SGML template defines the MathSymbol character format appropriately and the entity is declared in the DTD). If the software encounters Σ when exporting an document, it generates a reference to the `summation` entity (assuming the MathSymbol character

format is defined appropriately and applied to the character, and that the DTD for your application has an entity declaration for “sum”).

- To translate both the `thin` and `en SDATA` entity references in an SGML instance to en spaces in FrameMaker+SGML and to write all en spaces as an `en` entity reference, use these rules:

```
entity "en" is fm char 0x13;
reader entity "thin" is fm char 0x13;
```

See also

Rules mentioned in synopses	“entity” on page 349
General information on this topic	Chapter 13, “Translating Entities and Processing Instructions”

is fm colspec

Use the `is fm colspec` rule when you’re using a variant of the CALS table model. This rule indicates an SGML element that takes the place of the `colspec` element for a CALS table. In FrameMaker+SGML, that SGML element does not become a FrameMaker+SGML element. Rather, its attributes are used to specify properties of the corresponding table and table part elements.

Synopsis and contexts

```
element "gi" {. . .
   is fm colspec;
. . .}
```

Arguments

gi	An SGML generic identifier.

See also

Related rules	“is fm spanspec” on page 406
General information on this topic	Chapter 14, “Translating Tables” Appendix B, “The CALS Table Model” Appendix C, “SGML Read/Write Rules for CALS Table Model”

is fm cross-reference element

Use the `is fm cross-reference element` rule to identify an SGML element that translates to a cross-reference element in FrameMaker+SGML. You can choose either to

have the same name in both representations or to change the name during translation. The SGML element should have an attribute of type IDREF and declared content of EMPTY.

Synopsis and contexts

```
element "gi" {. . .
   is fm cross-reference element ["fmtag"];
. . .}
```

Arguments

gi	An SGML generic identifier.
fmtag	A FrameMaker+SGML element tag.

Details

- If you use the `is fm cross-reference element` rule, the other subrules of the `element` rule that you can use for that SGML element are as follows:

  `attribute`, page 335, specifies what to do with an SGML element's attributes.

  `fm attribute`, page 366, specifies what to do with attributes present in the FrameMaker+SGML representation of the element but not in the SGML representation.

  `fm property`, page 370, specifies what to do with FrameMaker+SGML properties associated with the element.

  `reader drop content`, page 344, specifies that the content but not the structure of an element should be discarded on import of an SGML document.

Examples

- To have the SGML element `xref` become the FrameMaker+SGML cross-reference element `Xref`, use this rule:

  ```
  element "xref" is fm cross-reference element;
  ```

- To have it become the FrameMaker+SGML cross-reference element `CrossRef`, use this rule:

  ```
  element "xref" is fm cross-reference element "CrossRef";
  ```

See also

Rules mentioned in synopses	"element" on page 345
General information on this topic	Chapter 16, "Translating Cross-References"

is fm element

If you do not specify a value for *fmtag*, the `is fm element` rule specifies only that an SGML element remains an element in FrameMaker+SGML. This is the default behavior. With a value for *fmtag*, this rule changes the element name when it is translated between SGML and FrameMaker+SGML.

Synopsis and contexts

```
element "gi" {. . .
   is fm element ["fmtag"];
. . .}
```

Arguments

gi	An SGML generic identifier.
fmtag	A FrameMaker+SGML element tag.

Details

- If you use the `is fm element` rule, the other subrules of the `element` rule that you can use for that SGML element are as follows:

  `attribute`, page 335, specifies what to do with an SGML element's attributes.

  `fm attribute`, page 366, specifies what to do with attributes present in the FrameMaker+SGML representation of the element but not in the SGML representation.

  `fm property`, page 370, specifies what to do with FrameMaker+SGML properties associated with the element.

  `reader drop content`, page 344, specifies that the content but not the structure of an element should be discarded on import of an SGML document.

  `writer drop content`, page 344, specifies that the content but not the structure of an element should be discarded on export of a FrameMaker+SGML document.

Examples

- To translate the SGML element `par` to the FrameMaker+SGML element `Paragraph`, use this rule:

  ```
  element "par" is fm element "Paragraph";
  ```

See also

Rules mentioned in synopses	"element" on page 345
General information on this topic	Chapter 12, "Translating Elements and Their Attributes"

is fm equation element

Use the `is fm equation element` rule to identify an SGML element that translates to an equation element in FrameMaker+SGML. You can choose either to have the same name in both representations or to change the name during translation.

Synopsis and contexts

```
element "gi" {. . .
   is fm equation element ["fmtag"];
. . .}
```

Arguments

gi	An SGML generic identifier.
fmtag	A FrameMaker+SGML element tag.

Details

- If you use this rule, the other subrules of the `element` rule that you can use for the same SGML element are as follows:

  `attribute`, page 335, specifies what to do with an SGML element's attributes.

  `fm attribute`, page 366, specifies what to do with attributes present in the FrameMaker+SGML representation of the element but not in the SGML representation.

  `fm property`, page 370, specifies what to do with FrameMaker+SGML properties associated with the element.

  `writer anchored frame`, page 333, tells FrameMaker+SGML what to do with graphic elements other than those with a single non-internal FrameMaker+SGML facet.

  `writer equation`, page 355, tells FrameMaker+SGML what to do with equation elements.

  `writer facet`, page 364, tells FrameMaker+SGML what to do with a graphic element that has a single non-internal FrameMaker+SGML facet.

Examples

- To have FrameMaker+SGML equation element `Eqn` become the SGML element `eqn`, use this rule:

  ```
  element "eqn" is fm equation element;
  ```

- To have FrameMaker+SGML equation element `Equation` become the SGML element `eqn`, use this rule:

  ```
  element "eqn" is fm equation element "Equation";
  ```

See also

Related rules	"is fm graphic element" on page 394
Rules mentioned in synopses	"element" on page 345
General information on this topic	Chapter 15, "Translating Graphics and Equations"

is fm footnote element

Use the `is fm footnote element` rule to identify an SGML element that translates to a footnote element in FrameMaker+SGML. You can choose either to have the same name in both representations or to change the name during translation.

Synopsis and contexts

```
element "gi" {. . .
   is fm footnote element ["fmtag"];
. . .}
```

Arguments

gi	An SGML generic identifier.
fmtag	A FrameMaker+SGML element tag.

Details

- If you use this rule, the other subrules of the `element` rule that you can use for the same SGML element are as follows:

  `attribute`, page 335, specifies what to do with an SGML element's attributes.

  `fm attribute`, page 366, specifies what to do with attributes present in the FrameMaker+SGML representation of the element but not in the SGML representation.

Examples

- To translate the SGML element `fn` to the `Fn` footnote element in FrameMaker+SGML, use this rule:

  ```
  element "fn" is fm footnote element;
  ```

- To translate it to the `Footnote` footnote element, use this rule:

  ```
  element "fn" is fm footnote element "Footnote";
  ```

See also

Rules mentioned in synopses	"element" on page 345
General information on this topic	Chapter 12, "Translating Elements and Their Attributes"

is fm graphic element

Use the `is fm graphic element` rule to identify an SGML element that translates to a graphic element in FrameMaker+SGML. You can choose either to have the same name in both representations or to change the name during translation.

Synopsis and contexts

```
element "gi" {. . .
   is fm graphic element ["fmtag"];
. . .}
```

Arguments

gi	An SGML generic identifier.
fmtag	A FrameMaker+SGML element tag.

Details

- If you use this rule, the other subrules of the `element` rule that you can use for the same SGML element are as follows:

  `attribute`, page 335, specifies what to do with an SGML element's attributes.

  `fm attribute`, page 366, specifies what to do with attributes present in the FrameMaker+SGML representation of the element but not in the SGML representation.

  `fm property`, page 370, specifies what to do with FrameMaker+SGML properties associated with the element.

  `writer anchored frame`, page 333, tells FrameMaker+SGML what to do with graphic elements other than those with a single non-internal FrameMaker+SGML facet.

  `writer equation`, page 355, tells FrameMaker+SGML what to do with equation elements.

  `writer facet`, page 364, tells FrameMaker+SGML what to do with an imported graphic element that has a single non-internal FrameMaker+SGML facet.

Examples

- To translate the SGML element `pict` to the `Pict` graphic element in FrameMaker+SGML, use this rule:

  ```
  element "pict" is fm graphic element;
  ```

- To translate it to the `Picture` graphic element, use this rule:

```
element "pict" is fm graphic element "Picture";
```

See also

Related rules	"is fm equation element" on page 392
Rules mentioned in synopses	"element" on page 345
General information on this topic	Chapter 15, "Translating Graphics and Equations"

is fm marker element

Use the `is fm marker element` rule to identify an SGML element that translates to a marker element in FrameMaker+SGML. You can choose either to have the same name in both representations or to change the name during translation.

Synopsis and contexts

```
element "gi" {. . .
   is fm marker element ["fmtag"];
. . .}
```

Arguments

gi	An SGML generic identifier.
fmtag	A FrameMaker+SGML element tag.

Details

- If you use this rule, the other subrules of the `element` rule that you can use for the same SGML element are as follows:

  `attribute`, page 335, specifies what to do with an SGML element's attributes.

  `fm attribute`, page 366, specifies what to do with attributes present in the FrameMaker+SGML representation of the element but not in the SGML representation.

  `fm property`, page 370, specifies what to do with FrameMaker+SGML properties associated with the element.

  `marker text is`, page 418, specifies whether the text of a FrameMaker+SGML marker element should be element content or an attribute value in SGML.

Examples

- To translate the SGML element `m` to the `M` marker element in FrameMaker+SGML, use this rule:

```
element "m" is fm marker element;
```

- To translate it to the `Marker` marker element, use this rule:

```
element "m" is fm marker element "Marker";
```

See also

Related rules	"marker text is" on page 418 "fm marker" on page 368
Rules mentioned in synopses	"element" on page 345
General information on this topic	Chapter 18, "Translating Markers"

is fm property

Use the `is fm property` rule to translate an SGML attribute to a FrameMaker+SGML property. This rule can apply in a highest-level `attribute` rule to set a default. Or it can apply within an `element` rule for a table, table part, marker, cross-reference, graphic, or equation element, to set the property only for that element.

Synopsis and contexts

```
1. attribute "attr" {. . .
      is fm property prop;
   . . .}

2. element "gi" {. . .
      attribute "attr" {. . .
         is fm property prop;
   . . .} . . .}
```

Arguments

attr	The name of an SGML attribute.
prop	A FrameMaker+SGML property. Possible properties are: • For cross-reference elements: `cross-reference format`, `cross-reference id`. • For graphic and equation elements: `entity`, `dpi`, `import size`, `import by reference or copy`, `sideways`, `import angle`, `horizontal offset`, `vertical offset`, `position`, `baseline offset`, `near-side offset`, `alignment`, `cropped`, `floating`, `angle`, `width`, or `height`. • For marker elements: `marker type` or `marker text`.

- For table elements: `column ruling`, `column widths`, `columns`, `page wide`, `row ruling`, `table border ruling`, or `table format`.
- For table cell elements: `column name`, `column number`, `column ruling`, `end column name`, `horizontal straddle`, `more rows`, `rotate`, `row ruling`, `span name`, `start column name`, or `vertical straddle`.
- For table row elements: `maximum height`, `minimum height`, `row type`, or `row ruling`.
- For CALS table colspecs: `cell alignment character`, `cell alignment offset`, `cell alignment type`, `column name`, `column number`, `column ruling`, `column width`, `row ruling`, or `vertical alignment`.
- For CALS table spanspecs: `cell alignment character`, `cell alignment offset`, `cell alignment type`, `column ruling`, `end column name`, `row ruling`, `span name`, `start column name`, or `vertical alignment`.

gi An SGML generic identifier.

Details

- In general, if you use the `is fm property` rule to translate an SGML attribute to a FrameMaker+SGML property, the SGML attribute does not also appear as a FrameMaker+SGML attribute.
- If you use this rule in a highest-level `attribute` rule, it applies only to elements that have that attribute and are of the appropriate type. For example, if you have these declarations:

  ```
  <!ATTLIST (graphic | table) w CDATA #IMPLIED>
  ```

  and these rules:

  ```
  attribute "w" is fm property width;
  element "graphic" is fm graphic element;
  element "table" is fm table element;
  ```

the `w` attribute becomes the `width` property of the `graphic` element but remains an attribute for the `table` element, since tables do not have a `width` property. If you intended `w` to be the column width for tables, you should use these rules:

```
element "graphic" {
   is fm graphic element;
   attribute "w" is fm property width;
}

element "table" {
   is fm table element;
   attribute "w" is fm property column width;
}
```

Examples

- The SGML attribute `w` may be used for multiple elements to represent the width of a table's columns. To translate it to the FrameMaker+SGML property `column width`:

```
attribute "w" is fm property column width;
```

- To translate the attribute `form` to the cross-reference formatting property `cross-reference format` for the element `xref`:

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

See also

Related rules	"fm property" on page 370 "is fm property value," next
Rules mentioned in synopses	"element" on page 345 "attribute" on page 335
General information on this topic	"Formatting properties for tables" on page 252 "Anchored frame properties" on page 278 "Other graphic properties" on page 280 Chapter 18, "Translating Markers" Chapter 16, "Translating Cross-References"

is fm property value

Use the `is fm property value` rule when an SGML attribute has a name token group as its declared value and you want to rename the individual name tokens when translating to and from FrameMaker+SGML property values.

Synopsis and contexts

```
1. value "token" is fm property value propval;

2. attribute "attr" {. . .
      value "token" is fm property value propval;
   . . .}

3. element "gi" {. . .
      attribute "attr" {. . .
         value "token" is fm property value propval;
   . . .} . . .}
```

Arguments

token	A token in a name token group.
propval	A defined FrameMaker+SGML property value.
attr	The name of an SGML attribute.
gi	An SGML generic identifier.

Details

- This rule can be used at the highest level to set a default, or within an `attribute` rule.
- Use this rule when the corresponding SGML attribute translates to a property in FrameMaker+SGML. If the SGML attribute translates to a choice attribute instead, you need to use the `is fm value` rule to specify the correspondence between SGML tokens and FrameMaker+SGML attribute choices.
- When using this rule, remember that SGML does not permit a token to appear in the declared value of more than one attribute of an element. For example, the following rule:

```
element "picture" {
   is fm graphic element;
   attribute "place" {
      is fm property position;
      value "left" is fm property value subcol left;
    }
   attribute "just" {
      is fm property alignment;
      value "left" is fm property value align left;
}}
```

corresponds to an erroneous SGML ATTLIST such as:

```
<!ATTLIST picture
   place (left, sright, snear, . . .)
   just (left, aright, acenter, . . .)
>
```

- The default declarations for graphic elements include three attributes that have a name token group as the declared value: `position`, `align`, and `impby`. These attributes correspond to the FrameMaker+SGML properties `position`, `alignment`, and `import by reference or copy`.

  By default, the defined values and corresponding property values for the `position` property are as follows:

Defined 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`

By default, the defined values and corresponding property values for the `alignment` property are as follows:

Defined value	Property value
`aleft`	`align left`
`aright`	`align right`
`acenter`	`align center`
`ainside`	`align inside`
`aoutside`	`align outside`

By default, the defined values and corresponding property values for the `import by reference or copy` property are as follows:

Defined value	Property value
`ref`	`reference`
`copy`	`copy`

- FrameMaker+SGML defines the `table border ruling` property for working with tables and the `alignment` and `vertical alignment` properties for working with colspecs and spanspecs.

  If you use the CALS table model for your tables, these properties automatically translate to the `frame`, `align`, and `valign` attributes on appropriate elements. There is also a default correspondence between the FrameMaker+SGML property values and the SGML defined values.

  If you do not use the CALS table model, you may still choose to translate these FrameMaker+SGML formatting properties to SGML attributes. In this case, you must also determine the translation from property value to defined value.

- If you use the CALS table model, the `frame` attribute has the following defined values: `all`, `top`, `bottom`, `topbot`, `sides`, and `none`. The values for the corresponding `table border ruling` property are the same as the defined values, except that the `topbot` defined value is the `top and bottom` property value.

  The `align` attribute and the corresponding `cell alignment type` property have the following values: `left`, `center`, `right`, `justify`, and `char`.

  The `valign` attribute and the corresponding `vertical alignment` property have the following values: `top`, `middle`, and `bottom`.

Examples

- To use the `table border ruling` property for a non-CALS table and to set its name tokens, use this rule:

```
element "tab" {
   is fm table element;
   attribute "frame" {
      is fm property table border ruling;
      value "all" is fm property value all;
      value "top" is fm property value top;
      value "bottom" is fm property value bottom;
      value "topbot" is fm property value top and bottom;
      value "sides" is fm property value sides;
      value "none" is fm property value none;
}}
```

  The DTD fragment for this element and attribute looks like this:

- To rename the FrameMaker+SGML `import by reference or copy` property as the `refcopy` attribute, and to also change the name tokens, use this rule:

```
attribute "refcopy" {
   is fm property import by reference or copy;
   value "r" is fm property value reference;
   value "c" is fm property value copy;
}
```

See also

Related rules	“fm property” on page 370
	“is fm property” on page 396
Rules mentioned in synopses	“attribute” on page 335
	“element” on page 345
	“value” on page 439

is fm reference element

Use the `is fm reference element` rule to translate an SGML `SDATA` entity to an element defined on a reference page in a FrameMaker+SGML document.

Synopsis and contexts

1. `entity "`*`ename`*`" is fm reference element ["`*`fmtag`*`"];`
2. `reader entity "`*`ename`*`" is fm reference element ["`*`fmtag`*`"];`

Arguments

ename	An SGML entity name.
fmtag	A FrameMaker+SGML element tag.

Details

- Instead of using this rule to translate an `SDATA` entity, you can use a parameter literal of a particular form. For information on how to do so, see "Translating SDATA entities as FrameMaker+SGML reference elements" on page 240.
- You can use the `is fm reference element` rule within an `entity` rule at the highest level to have the translation occur in both directions. Or you can put the `entity` rule inside a `reader` rule to have the translation occur only when reading an SGML document into FrameMaker+SGML. Remember that the SDATA entity must be declared in the DTD in order to use this rule.
- The FrameMaker+SGML element must occur in a flow named `Reference Elements`. That flow must be on a reference page of the application's template file 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 user's manual.
- When FrameMaker+SGML encounters references to the specified entity while translating an SGML document to FrameMaker+SGML, it copies the appropriate element from its reference page in the FrameMaker+SGML template associated with the SGML application. When it encounters an instance of an element associated with one of the reference pages while writing a FrameMaker+SGML document to SGML, it generates an entity reference.
- When you use this rule, the *fmtag* element must be defined for your FrameMaker+SGML documents and valid in the contexts in which *ename* occurs. If it is not, the resulting FrameMaker+SGML document is invalid.

Examples

- Assume you have an entity named `legalese` which contains text you need to include in many places. The entity is too long to be a FrameMaker+SGML variable, and you don't want to treat it as an entire paragraph. Instead, you can choose to have the entity correspond to a text range element called `LegaleseFragment`. To do so, add the following rule to your rules document:

  ```
  entity "legalese" is fm reference element "LegaleseFragment";
  ```

  The entity declaration to your DTD looks like this:

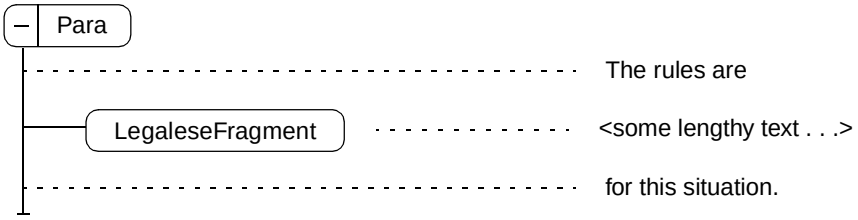
  ```
  <!ENTITY legalese SDATA "[]">
  ```

Create a reference frame on the reference page of your application which contains the element “LegaleseFragment” with your boilerplate text. In order for the element to be treated as a “text range” use the appropriate TextFormatRules for this element in the EDD.

When FrameMaker+SGML translates an SGML document that contains the following markup:

```
<para>The rules are &legalese; for this situation.</para>
```

It produces the following element structure:



See also

Rules mentioned in synopses	“entity” on page 349
General information on this topic	Chapter 13, “Translating Entities and Processing Instructions”

is fm rubi element

Use the `is fm rubi element` rule to identify an SGML element that translates to a Rubi element in FrameMaker+SGML. You can choose either to have the same name in both representations or to change the name during translation.

Synopsis and contexts

```
element "gi" {. . .
   is fm rubi element ["fmtag"];
. . .}
```

Arguments

gi	An SGML generic identifier.
fmtag	A FrameMaker+SGML element tag.

Details

- If you use this rule, the other subrules of the `element` rule that you can use for the same SGML element are as follows:

  `attribute`, page 335, specifies what to do with an SGML element's attributes.

  `fm attribute`, page 366, specifies what to do with attributes present in the FrameMaker+SGML representation of the element but not in the SGML representation.

Examples

- To translate the SGML element `rubitext` to the `Rubitext` element in FrameMaker+SGML, use this rule:

```
element "rubitext" is fm rubi group element;
```

- To translate it to the `MyRubiTextp` element, use this rule:

```
element "rubitext" is fm rubi element "MyRubiText";
```

See also

Rules mentioned in synopses	"element" on page 345
General information on this topic	Chapter 12, "Translating Elements and Their Attributes"

is fm rubi group element

Use the `is fm rubi group element` rule to identify an SGML element that translates to a Rubi group element in FrameMaker+SGML. You can choose either to have the same name in both representations or to change the name during translation.

Synopsis and contexts

```
element "gi" {. . .
   is fm rubi group element ["fmtag"];
. . .}
```

Arguments

gi	An SGML generic identifier.
fmtag	A FrameMaker+SGML element tag.

Details

- If you use this rule, the other subrules of the `element` rule that you can use for the same SGML element are as follows:

  `attribute`, page 335, specifies what to do with an SGML element's attributes.

  `fm attribute`, page 366, specifies what to do with attributes present in the FrameMaker+SGML representation of the element but not in the SGML representation.

Examples

- To translate the SGML element `rubigroup` to the `Rubigroup` element in FrameMaker+SGML, use this rule:

```
element "rubigroup" is fm rubi group element;
```

- To translate it to the `MyRubiGroup` element, use this rule:

```
element "rubigroup" is fm rubi group element "MyRubiGroup";
```

See also

Rules mentioned in synopses	"element" on page 345
General information on this topic	Chapter 12, "Translating Elements and Their Attributes"

is fm spanspec

Use the `is fm spanspec` when you're using a variant of the CALS table model. This rule indicates an SGML element that takes the place of the `spanspec` element for a CALS table. In FrameMaker+SGML, that SGML element does not become a FrameMaker+SGML element. Rather, its attributes are used to specify properties of the corresponding table and table part elements.

Synopsis and contexts

```
element "gi" {. . .
   is fm spanspec;
. . .}
```

Arguments

gi	An SGML generic identifier.

See also

Related rules	"is fm colspec" on page 389
General information on this topic	Chapter 14, "Translating Tables" Appendix B, "The CALS Table Model" Appendix C, "SGML Read/Write Rules for CALS Table Model"

is fm system variable element

Use the `is fm system variable element` rule to identify an SGML element that translates to a system variable element in FrameMaker+SGML. You can choose either to have the same name in both representations or to change the name during translation.

Synopsis and contexts

```
element "gi" {. . .
   is fm system variable element ["fmtag"];
. . .}
```

Arguments

gi	An SGML generic identifier.
fmtag	A FrameMaker+SGML element tag.

Details

- If you use this rule, the other subrules of the `element` rule that you can use for the same SGML element are:

  `attribute`, page 335, specifies what to do with an SGML element's attributes.

  `fm attribute`, page 366, specifies what to do with attributes present in the FrameMaker+SGML representation of the element but not in the SGML representation.

- This rule does not apply to translating non-element FrameMaker+SGML variables.

Examples

- To translate the SGML element `date` to the `Date` system variable element in FrameMaker+SGML, use this rule:

```
element "date" is fm system variable element;
```

You specify which system variable to use by adding a rule to the `Date` element's definition in the FrameMaker+SGML EDD. For example:

Element (System Variable): Date

System variable format rule

In all contexts.
Use system variable: Current Date (Long)

See also

Related rules	"is fm variable" on page 414 "fm variable" on page 372
Rules mentioned in synopses	"element" on page 345
General information on this topic	Chapter 17, "Translating Variables and System Variable Elements"

is fm table element

Use the `is fm table element` rule to identify an SGML element that translates to a table element in FrameMaker+SGML. You can choose either to have the same name in both representations or to change the name during translation.

Synopsis and contexts

```
element "gi" {. . .
   is fm table element ["fmtag"];
. . .}
```

Arguments

gi	An SGML generic identifier.
fmtag	A FrameMaker+SGML element tag.

Details

- If you use the CALS table model, you do not need to use this rule to translate the CALS `table` element properly.
- If your SGML element declarations for a table element do not include an attribute that corresponds to the `columns` property, you must use the `fm property` rule to specify a number of columns for the table.
- If you use this rule, the other subrules of the `element` rule that you can use for the same SGML element are as follows:

`attribute`, page 335, specifies what to do with an SGML element's attributes.

`fm attribute`, page 366, specifies what to do with attributes present in the FrameMaker+SGML representation of the element but not in the SGML representation.

`fm property`, page 370, specifies what to do with FrameMaker+SGML properties associated with the element.

`reader insert table` *`part`* `element`, page 381, indicates that the software should insert the indicated table part (table title, table heading, or table footing), even if the SGML element structure or instance does not contain the corresponding element.

Examples

- To translate the SGML element `gloss` to the `Gloss` table element in FrameMaker+SGML, use this rule:

```
element "gloss" is fm table element;
```

- To translate it to the `Glossary` table element, use this rule:

```
element "gloss" is fm table element "Glossary";
```

See also

Rules mentioned in synopses	"element" on page 345
General information on this topic	Chapter 14, "Translating Tables"

is fm table part element

Use the `is fm table` *`part`* `element` rule to identify an SGML element that translates to a table part element in FrameMaker+SGML, such as a table title element. You can choose either to have the same name in both representations or to change the name during translation.

Synopsis and contexts

```
element "gi" {. . .
   is fm table part element ["fmtag"];
. . .}
```

Arguments

`gi`	An SGML generic identifier.
`part`	A FrameMaker+SGML table part. One of the keywords: `title`, `body`, `heading`, `footing`, `row`, `cell`.
`fmtag`	A FrameMaker+SGML element tag.

Details

- If you use the CALS table model, you do not need to use this rule to translate elements representing parts of tables in CALS properly.
- If you map an SGML element to a FrameMaker+SGML table part element, then the element cannot be used anywhere in the instance except as that table part. For example, if you have a `“title”` element and you use the following rule:

  `element “title” is fm table title element;”`

  Then you would not be able to insert a `“title”` element in a `Chapter` element.
- If you use this rule, the other subrules of the `element` rule that you can use for the same SGML element are as follows:

  `attribute`, page 335, specifies what to do with an SGML element’s attributes.

  `fm attribute`, page 366, specifies what to do with attributes present in the FrameMaker+SGML representation of the element but not in the SGML representation.

  `fm property`, page 370, specifies what to do with FrameMaker+SGML properties associated with the element.

  `reader end vertical straddle`, page 348, indicates that the associated table row or cell element terminates a vertical table straddle. This subrule applies only if *`part`* is `row` or `cell`.

  `reader start new row`, page 433, indicates that the associated table cell element indicates the start of a new row in the table. This subrule applies only if *`part`* is `cell`.

  `reader start vertical straddle`, page 434, indicates that the associated table cell element starts a vertical table straddle. This subrule applies only if *`part`* is `cell`.

Examples

- To translate the SGML element `head` as the FrameMaker+SGML table heading element `Head`, use this rule:

```
element "head" is fm table heading element;
```

- To translate the SGML element `dfn` as the FrameMaker+SGML table cell element `Definition`, use this rule:

```
element "dfn" is fm table cell element;
```

See also

Rules mentioned in synopses	“element” on page 345
General information on this topic	Chapter 14, “Translating Tables”

is fm text inset

Use the `is fm text inset` rule to translate a declared SGML `SDATA` entity to a text inset in FrameMaker+SGML. In fact, you can translate any entity to a text inset, but we suggest you only do this with `SDATA` entities.

Synopsis and contexts

```
1. entity "ename" is fm text inset "fname"
      [in body_or_ref flow "flowname"];

2. reader entity "ename" is fm text inset "fname"
      [in body_or_ref flow "flowname"];
```

Arguments

ename	An SGML entity name.
fname	A filename containing the text to include. This file must be a FrameMaker+SGML document or a file of a type for which FrameMaker+SGML has a filter, for example, a MS-Word document.
body_or_ref	One of the keywords: `body` or `reference`, indicating the type of text flow in which to find the text to include. You can specify this option only if *fname* is a FrameMaker+SGML document.
flowname	The name of the FrameMaker+SGML text flow.

Details

- By default, external text entities in SGML are imported as text insets. For the SGML to be valid, the external text entities must be text or SGML files. In the FrameMaker+SGML document, the text insets use these files as their sources. It is probably most advantageous to retain these files for the text insets; you do not need to use the `is fm text inset` rule to import external text entities as text insets.
- The source file for the text inset must either be a FrameMaker+SGML file or a file of a format FrameMaker+SGML can filter automatically. You cannot use an SGML file as the source of the text inset.
- Instead of using this rule to translate an `SDATA` entity to a text inset, you can use a parameter literal of a particular form. For information on how to do so, see "Translating SDATA entities as FrameMaker+SGML text insets" on page 238.
- You can use the `is fm text inset` rule within an `entity` rule at the highest level to have the translation occur in both directions. Or you can put the `entity` rule inside a `reader` rule to have the translation occur only when reading an SGML document into FrameMaker+SGML.

- If *fname* is not a FrameMaker+SGML or FrameMaker document, you cannot specify the `in body flow` or `in reference flow` options. In this case, FrameMaker+SGML uses all of the text in the file specified by *fname* for the text inset.

  If *fname* is a FrameMaker+SGML document and you do not specify a flow to use, FrameMaker+SGML use the contents of the main body flow of the specified document.

- By default, the software reformats the text inset to conform to the format rules of the document containing the text inset. If the source for the text inset has element structure, FrameMaker+SGML also retains that element structure.

  You can confirm this behavior with the `reformat using target document catalogs` rule. You can change this behavior using the subrules `reformat as plain text` or `retain source document formatting`.

- FrameMaker+SGML requires that a structured flow have exactly one highest-level element. For this reason, you cannot use a single text inset to include multiple elements at the top level of the inset. You must use multiple text insets for this purpose.

- FrameMaker+SGML puts an end-of-paragraph symbol after a text inset. For this reason, you cannot use a text inset to insert a range of text inside a single paragraph. To do so, you can translate the entity either as a FrameMaker+SGML variable (with the `is fm variable` rule) or as a reference element (with the `is fm reference element` rule).

Examples

- Assume you have declared an SGML SDATA entity. You also have a single paragraph of boilerplate text to be used in your documents. You can place this text on a reference page in a text column with a flow called `BoilerPlate` in the FrameMaker+SGML template for your SGML application. If that template is the file `template.doc`, you could use this rule to translate occurrences of the `boiler` entity to a text inset in corresponding FrameMaker+SGML documents:

```
entity "boiler"
   is fm text inset "template.doc"
      in reference flow "BoilerPlate";
```

See also

Related rules	“reformat as plain text” on page 429
	“reformat using target document catalogs” on page 429
	“retain source document formatting” on page 430
	“is fm reference element” on page 402
	“is fm variable” on page 414
Rules mentioned in synopses	“entity” on page 349
	“reader” on page 427

General information on this topic — Chapter 13, “Translating Entities and Processing Instructions”

is fm value

Use the `is fm value` rule to translate the value of an SGML attribute to a particular choice for a FrameMaker+SGML choice attribute. The SGML attribute’s declared value must be a name token group or `NOTATION` and a name token group.

Synopsis and contexts

```
1. value "token" is fm value "val";

2. attribute "attr" {. . .
      value "token" is fm value "val";
   . . .}

3. element "gi" {. . .
      attribute "attr" {. . .
         value "token" is fm value "val";
   . . .} . . .}
```

Arguments

token	A token in a name token group.
val	An allowed value for a FrameMaker+SGML choice attribute.
attr	The name of an SGML attribute.
gi	An SGML generic identifier.

Details

- Use this rule when the corresponding SGML attribute translates to a choice attribute in FrameMaker+SGML. If the SGML attribute translates to a FrameMaker+SGML property, you need to use the `is fm property value` rule to specify the correspondence between SGML tokens and FrameMaker+SGML property values.

Examples

- If the token list `(r | b | g)` is used by multiple attributes, you can use these rules to translate the individual tokens consistently:

```
value "r" is fm value "Red";
value "b" is fm value "Blue";
value "g" is fm value "Green";
```

- If the token list `(r | b | g)` is used by several attributes as above but by the `bird` element differently, you can add this rule to the above rules:

```
element "bird" {is fm element;
   ] attribute "species" {
   value "r" is fm value "Robin";
   value "b" is fm value "Blue Jay";
   value "g" is fm value "Goldfinch";
}}]
```

See also

Related rules	"is fm property value" on page 398
Rules mentioned in synopses	"attribute" on page 335 "element" on page 345 "value" on page 439
General information on this topic	Chapter 12, "Translating Elements and Their Attributes"

is fm variable

Use the `is fm variable` rule to translate a declared SGML text entity to a FrameMaker+SGML non-element variable.

Synopsis and contexts

1. `entity "`*`ename`*`"` **`is fm variable`** `["`*`var`*`"];`
2. `reader entity "`*`ename`*`"` **`is fm variable`** `["`*`var`*`"];`

Arguments

ename	An SGML entity name.
var	A FrameMaker+SGML variable name.

Details

- You can use the `is fm variable` rule within an `entity` rule at the highest level to have the translation occur in both directions. Or you can put the `entity` rule inside a `reader` rule to have the translation occur only when reading an SGML document into FrameMaker+SGML.

Examples

- To translate the SGML element v to a non-element FrameMaker+SGML variable of the same name:

```
entity "v" is fm variable;
```

- To translate the FrameMaker+SGML variable `Licensor` to the SGML element `lic`, use this rule:

```
entity "lic" is fm variable "Licensor";
```

See also

Related rules	"fm variable" on page 372
	"is fm system variable element" on page 407
Rules mentioned in synopses	"entity" on page 349
General information on this topic	Chapter 17, "Translating Variables and System Variable Elements"

is processing instruction

On export, you use the `is processing instruction` rule to tell FrameMaker+SGML to create processing instructions for all non-element markers or for non-element markers of a particular type. By default, FrameMaker+SGML creates processing instructions for all non-element markers. You have the option of discarding non-element markers; you might use this rule in conjunction with the `drop` rule when you want to discard some but not all non-element markers.

Synopsis and contexts

fm marker ["$type_1$", . . ., "$type_n$"] **is processing instruction**;

Arguments

$type_i$	A FrameMaker+SGML marker type, such as Index or Type 22.

Details

- If you do not supply any $type_i$ arguments, this rule applies to all non-element markers other than markers of the type reserved by FrameMaker+SGML for storing SGML processing instructions, PI entities, and external data entities. (By default, the reserved marker types are `SGML PI` and `SGML ENTREF`.)

Examples

- To discard all nonelement markers other than Index markers, use these rules:

```
fm marker "Index" is processing instruction;
fm marker drop;
```

See also

Rules mentioned in synopses	“fm marker” on page 368
General information on this topic	Chapter 18, “Translating Markers”

line break

Use the `line break` rule to tell FrameMaker+SGML about any limits on the length of lines in an SGML file it generates. You also use it to tell the software whether or not to interpret line breaks in an SGML document as FrameMaker+SGML paragraph breaks within elements.

Synopsis and contexts

```
1. reader line break is mode;

2. writer line break is mode;

3. element "gi" {. . .
      reader {. . .
         line break is mode;
   . . .} . . .}

4. element "gi" {. . .
      writer {. . .
         line break is mode;
   . . .} . . .}
```

Arguments

mode	For `writer`: *n* `characters` (where *n* is a positive integer in C syntax). For `reader`: one of `forced return` or `space`.
gi	An SGML generic identifier.

Details

- This rule can be used at the highest level to set a default or within an `element` rule to set line breaks for only that element.
- On export, FrameMaker+SGML 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 and attempts to avoid generating lines longer than *n* characters (the default is 80). It maintains a counter indicating how many characters it has placed on a single line. After this counter reaches *n*`-10`, it changes the next data character space to a record end.

  It generates an SGML record end at the end of every paragraph and flow in the FrameMaker+SGML document.

  Within the content of an SGML element that can contain elements but no text, FrameMaker+SGML writes a record end after every start-tag and before every end-tag.

  Within a start-tag, it puts a record end followed by a tab character between every pair of attribute-value pairs.

If you want a start-tag for an element and its contents to appear on the same line in the SGML document, you must write an SGML API client.

- On import you have control over record ends not ignored by the underlying SGML parser. Within a `reader` rule, *`mode`* can be one of the following:

  `forced return` informs FrameMaker+SGML that a line break within a text segment should be converted to a forced return.

  `space` informs FrameMaker+SGML that a line break within a text segment should be treated as a space. This is the default.

Examples

- Line breaks may need to be treated differently within different elements. For example, a line break within an `example` element may need to be preserved on import, while a line break within a `par` element may be a word break:

```
element "example" reader line break is forced return;
element "par" reader line break is space;
```

marker text is

Use the `marker text is` rule to indicate whether the text of a marker element should become an attribute value or the content of the corresponding SGML element. Note that the SGML element must not be declared as empty if you want the marker text to be translated as content.

Synopsis and contexts

```
element "gi" {. . .
   is fm marker element ["fmtag"];
   marker text is attr_or_content;
. . .}
```

Arguments

`gi`	An SGML generic identifier.
`fmtag`	A FrameMaker+SGML element tag.
`attr_or_content`	One of the keywords: `attribute` or `content`.

Details

- By default, FrameMaker+SGML translates a marker element in FrameMaker+SGML to an SGML empty element. It writes the marker text as the value of the SGML element's `text` attribute.
- Instead of the default, you can have FrameMaker+SGML translate a marker element to an SGML element whose content model is `#PCDATA`. The marker text becomes the element's content.

Examples

- To state that the SGML element `mkr` corresponds to the FrameMaker+SGML element `Marker` and to confirm the default behavior, you can use this rule:

  ```
  element "mkr" {
     is fm marker element "Marker";
     marker text is attribute;
  }
  ```

  With this rule, the FrameMaker+SGML element definition:

  Element (Marker): Marker

  corresponds to the SGML declarations:

  ```
  <!ELEMENT mkr - O EMPTY>
  <!ATTLIST mkr text CDATA #IMPLIED
      type CDATA #IMPLIED>
  ```

  In this case, if FrameMaker+SGML writes as SGML a document containing an instance of the `Marker` element whose marker text is “Some marker text” and whose type is Type 22, the software outputs this information:

  ```
  <mkr text="Some marker text" type="Type 22">
  ```

- To state that the SGML element `mkr` corresponds to the FrameMaker+SGML element `Marker` but that the marker text should become element content in SGML, you can use this rule:

  ```
  element "mkr" {
     is fm marker element "Marker";
     marker text is content;
  }
  ```

  With this rule, the FrameMaker+SGML element definition:

  Element (Marker): Marker

  corresponds to the SGML declarations:

  ```
  <!ELEMENT mkr - - (#PCDATA)>
  <!ATTLIST mkr type CDATA #IMPLIED>
  ```

  In this case, if FrameMaker+SGML writes as SGML a document containing an instance of the `Marker` element whose marker text is “Some marker text” and whose type is Type 22, the software outputs this information:

  ```
  <mkr type="Type 22">
  Some marker text
  </mkr>
  ```

See also

Rules mentioned in synopses	"element" on page 345 "is fm marker element" on page 395
General information on this topic	Chapter 18, "Translating Markers"

notation is

Use the `notation is` rule only in an `element` rule for a graphic or equation element, to provide information the software needs when writing a document containing graphics and equations to SGML. FrameMaker+SGML uses this rule to determine the data content notation name to include in entity declarations it generates.

Synopsis and contexts

```
element "gi" {
   is fm graphic_or_eqn element ["fmtag"];
   writer type ["facetname"] notation is "notation";
. . .}}
```

Arguments

gi	An SGML generic identifier.
graphic_or_eqn	One of the keywords: `graphic` or `equation`.
type	One of the rules `anchored frame`, `facet`, or `equation`. If `facet`, you must also supply the *facetname* argument. If *type* is `equation`, the rule applies to equation elements. If *type* is `facet`, the rule applies to a graphic element that contains only a single facet with the name specified by *facetname*. This occurs when the graphic element is an anchored frame containing only a single imported graphic object whose original file was in the *facetname* graphic format. You can use this rule with *type* set to `facet` multiple times if you want the software to treat several file formats differently. If *type* is `anchored frame`, the rule applies to a graphic element under all other circumstances.
facetname	A facet name. You supply this argument if and only if *type* is `facet`.
notation	A string representing a data content notation name.

Details

- By default, FrameMaker+SGML uses the first eight characters of the name of the facet it exports as the data content notation. If the graphic or equation has only internal FrameMaker+SGML facets, the software uses CGM as the data content notation.

Examples

- Assume your end users use the `af` graphic element within FrameMaker+SGML, creating the graphics using FrameMaker+SGML tools, but want to store them in TIFF format on export. Furthermore, you want to name the files based on the FrameMaker+SGML document's name, but with an extension of `.gr`. You can accomplish this with the following rule:

```
element "af" {
   is fm graphic element;
   writer anchored frame {
      notation is "TIFF";
      export to file "$(docname).gr";
}}
```

  If you export the FrameMaker+SGML file `intro.doc`, the software writes the following entity declaration for the first instance of the `af` element that it finds:

```
<!ENTITY af1 SYSTEM "intro1.gr" NDATA TIFF>
```

See also

Related rules
"convert referenced graphics" on page 340
"entity name is" on page 352
"export to file" on page 359
"specify size in" on page 431

Rules mentioned in synopses
"element" on page 345
"is fm graphic element" on page 394
"is fm equation element" on page 392
"anchored frame" on page 333
"equation" on page 355
"facet" on page 364
"writer" on page 442

General information on this topic
Chapter 15, "Translating Graphics and Equations"

output book processing instructions

By default, when FrameMaker+SGML converts a FrameMaker+SGML book to SGML, it puts `?FM: book` and `?FM: document` processing instructions in the SGML document to indicate where the individual files in the FrameMaker+SGML documents began. You use the `output book processing instructions` rule to confirm or change this behavior.

Synopsis and contexts

```
writer [do not] output book processing instructions;
```

Arguments

None.

Details

- If you use the `generate book` rule to tell FrameMaker+SGML to use elements to identify book components when reading an SGML document, you might choose to not have it output processing instructions when writing the book to SGML. In this case, use this rule:

  ```
  writer do not output book processing instructions;
  ```

See also

Related rules	"generate book" on page 373
General information on this topic	Chapter 19, "Processing Multiple Files as Books"

preserve fm element definition

Use the `preserve fm element definition` rule to tell FrameMaker+SGML, when it is updating an EDD from a revised DTD, not to update the definition of a set of FrameMaker+SGML elements and their attributes on the basis of the DTD and other rules.

Synopsis and contexts

```
reader {. . .
   preserve fm element definition "fmtag1"[, . . ., "fmtagN"];
. . .}
```

Arguments

$fmtag_i$	A FrameMaker+SGML element tag.

Details

- FrameMaker+SGML uses the `preserve fm element definition` rule only when updating an EDD from a DTD. By default, when it updates an existing EDD, the software changes the definitions of FrameMaker+SGML elements to reflect the new DTD and all

SGML read/write rules. You may not want the definition of the FrameMaker+SGML element to change. For example, if one of your rules is to unwrap the element `body`, then any element with a definition that includes `body` will be modified directly include the contents of `body` instead of including `body`.

Examples

- Assume you have the rule:

  ```
  fm element "Body" unwrap;
  ```

  and the element definitions:

  Element (Container): Figure1
  General rule: Caption, Body

  Element (Container): Figure2
  General rule: Body, Footer

  Element (Container): Body
  General rule: Header, Line+

  The corresponding SGML declarations are:

  ```
  <!ELEMENT figure1 - - (caption, header, line+)>
  <!ELEMENT figure2 - - (header, line+, footer)>
  ```

  If you update the EDD containing the preceding definitions and use as input the DTD with the preceding declarations, FrameMaker+SGML replaces the definitions of `Figure1` and `Figure2` with:

  Element (Container): Figure1
  General rule: Caption, Header, Line+

  Element (Container): Figure2
  General rule: Header, Line+, Footer

  If you wish to retain the original definitions of `Figure1` and `Figure2` in the revised EDD, include this rule:

  ```
  reader preserve fm element definition "Figure1", "Figure2";
  ```

- Suppose you want to use an SGML API client to reverse the order of child elements in corresponding SGML and FrameMaker+SGML elements. For example, assume you have the SGML definition:

  ```
  <!ELEMENT ex - - (a, b)>
  ```

  and the FrameMaker+SGML element definition:

  Element (Container): Ex
  General rule: B, A

If you have no rules and update the EDD in this situation, FrameMaker+SGML updates the definition of `Ex` to correspond to the SGML definition. To suppress this change, use this rule:

```
reader preserve fm element definition "Ex";
```

See also

Related rules "drop" on page 342
"unwrap" on page 436

preserve line breaks

Use the `preserve line breaks` rule to tell FrameMaker+SGML to keep line breaks for an element when importing and exporting SGML documents. When importing SGML, it translates every RE in the element as a forced return. When exporting SGML, it translates forced returns as RE characters, and the line ends FrameMaker+SGML creates when automatically wrapping the text as non-RE line breaks in the SGML file. This is useful for elements that use RE characters to insert white space in an element's content.

Synopsis and contexts

```
element {. . .
   preserve line breaks ;
. . .}
```

Arguments

None

Details

- For an element using this rule, the software writes a an RE (line break) immediately after the open tag and immediately before the close tag.
- For an element using this rule, on export, FrameMaker+SGML writes a space character entity reference (&#SPACE) and an RE (line break) for each necessary line break in the SGML file. See the "line break" rule for information on how FrameMaker+SGML determines where to put these line breaks by default. Forced returns (shift-return) translate as RE characters (line breaks) in the SGML file.
- For export and import to have the same results, preserve line breaks must be specified for the same elements. For example, assume you use preserve line breaks on export for an element named Code. FrameMaker+SGML writes a space character entity reference and an RE (line break) when a line approaches the maximum line length, and it writes RE characters (line breaks) for forced returns. Now assume you remove preserve line breaks from the rules for the Code element. On import, FrameMaker+SGML will translate as spaces the space character entity reference/RE pairs, and as spaces any RE

characters (line breaks) not removed by the parser (default behavior). Thus the forced returns (shift-return) are lost and the imported file is not the same as the exported file.

- When importing SGML, `preserve line breaks` overrides the `line break is space` rule, if that rule is set. On import, `preserve line breaks` has the same effect for the specified element as the `line break is forced return` rule.

Examples

- The following rule preserves line breaks on import and export for the element named `code`:

```
fm element "code" {
   is fm element "Code";
   preserve line breaks;
}
```

See also

Rules mentioned in synopses	"element" on page 345
Related rules	"line break" on page 417

processing instruction

Use the `processing instruction` rule to drop SGML processing instructions that are not recognized by FrameMaker+SGML. By default, the software stores such processing instructions as the marker text in non-element markers of type `SGML PI`.

Synopsis and contexts

```
processing instruction drop;
```

Arguments

None

Details

- When you translate an SGML document to FrameMaker+SGML and the software encounters an unrecognized processing instruction such as:

  ```
  <?mypi>
  ```

  it stores the processing instruction as the text of a non-element `SGML PI` marker by default, with the following as the marker text:

  ```
  mypi
  ```

  When you translate a FrameMaker+SGML document to SGML, it outputs the corresponding processing instruction if it finds a non-element `SGML PI` marker with text in that format.

- This rule does not affect how FrameMaker+SGML treats the processing instructions it does recognize for books, book components, and other non-element markers.

Examples

- To discard all unrecognized processing instructions, use this rule:

```
processing instruction drop;
```

See also

Rules mentioned in synopses	"drop" on page 342
General information on this topic	Chapter 13, "Translating Entities and Processing Instructions"

proportional width resolution is

Use the `proportional width resolution is` rule to change the number used as the total for proportional column widths in tables. By default, if FrameMaker+SGML writes proportional columns widths, those widths add to 100.

Synopsis and contexts

```
writer proportional width resolution is "value";
```

Arguments

value	An integer indicating the total for proportional column width values.

Details

- Using this rule does not indicate that FrameMaker+SGML uses proportional widths, only that if the software writes proportional widths, then those widths add to *value* instead of to 100. To tell FrameMaker+SGML to use proportional widths, you must include the `use proportional widths rule`.

Examples

- Assume you do not use the `proportional width resolution is` rule, but have this rule:

```
writer use proportional widths;
```

  Further assume you have a 5-column table whose first two columns are 1 inch wide and whose last three columns are 2 inches wide. If the column widths are written to the `colwidth` attribute of the SGML `table` element, then FrameMaker+SGML creates this start-tag for that table:

```
<table colwidth="12.5* 12.5* 25* 25* 25*">
```

- Assume you have the same table as in the last example and you use this rule:

```
writer {
   use proportional widths;
   proportional width resolution is "8";
}
```

  FrameMaker+SGML writes this start-tag for the table:

```
<table colwidth="1* 1* 2* 2* 2*">
```

- Assume you have the same table as in the previous examples and you use this rule:

```
writer proportional width resolution is "8";
```

  That is, you do not also have the `use proportional widths` rule. In this case, FrameMaker+SGML ignores the "proportional width resolution" rule and writes this start-tag for the table:

```
<table colwidth="1in 1in 2in 2in 2in">
```

See also

Related rules	"use proportional widths" on page 438
General information on this topic	Chapter 14, "Translating Tables"

put element

See "generate book" on page 373.

reader

The `reader` rule indicates a rule that applies only on import to FrameMaker+SGML. It can be used at the highest level to set a default, or within an `element` rule to specify information particular to that element.

Synopsis and contexts

```
1. element "gi" {. . .
      reader {. . .
         subrule;
   . . .} . . .}

2. reader {. . .
      subrule;
   . . .}
```

Arguments

gi	An SGML generic identifier.

subrule Valid subrules:

`character map`, page 337, changes how FrameMaker+SGML translates between individual characters in the SGML and FrameMaker+SGML character sets. Allowed only at the highest level.

`drop content`, page 344, imports only the element itself, not its contents. Allowed only within an `element` rule.

`end vertical straddle`, page 348, specifies the end of a vertical straddle in a table. Allowed only within an `element` rule for a table cell or row element.

`entity`, page 349, specifies the treatment of an SGML entity in FrameMaker+SGML. Allowed only at the highest level.

`generate book`, page 373, specifies how to identify book components in an SGML document. Allowed only at the highest level.

`insert table` *`part`* `element`, page 381, specifies that FrameMaker+SGML should generate a table part (table title, table heading, or table footing) even if there is no content for that part. Allowed only within an `element` rule for a table element.

`line break`, page 417, changes the treatment of line breaks in the SGML instance which are not handled by the parser on import. Allowed at the highest level or within an `element` rule.

`preserve fm element definition`, page 422, instructs the software not to modify a FrameMaker+SGML element definition when updating an existing EDD. Allowed only at the highest level.

`start new row`, page 433, specifies that this table cell element starts a new row in the table. Allowed only within an `element` rule for a table row element.

`start vertical straddle`, page 434, specifies the start of a vertical straddle in a table. Allowed only within an `element` rule for a table cell element.

`table ruling style is`, page 435, specifies the ruling style to apply to all tables. Allowed only at the highest level.

Examples

- To change the default ruling style for tables:

```
reader table ruling style is "thick";
```

reformat as plain text

Use the `reformat as plain text` rule in an `entity` rule for an entity you want to translate as a text inset in FrameMaker+SGML. This specifies that the software should remove any element structure from the text inset and reformat the text using the format rules of the document into which the text inset is placed. You specify the other choices for formatting text insets with the rules `reformat using target document catalogs` and `retain source document formatting`.

Synopsis and contexts

```
1. entity "ename" {
      is fm text inset “fname”;
      reformat as plain text;
   . . .}

2. reader entity "ename" {
      is fm text inset “fname”;
      reformat as plain text;
   . . .}
```

Arguments

ename	An SGML entity name.

See also

Related rules	“reformat using target document catalogs,” next “retain source document formatting” on page 430
Rules mentioned in synopses	“entity” on page 349 “is fm text inset” on page 411
General information on this topic	Chapter 13, “Translating Entities and Processing Instructions”

reformat using target document catalogs

Use the `reformat using target document catalogs` rule in an `entity` rule for an entity you want to translate as a text inset in FrameMaker+SGML. This specifies that the software should retain any element structure from the text inset and reformat the text using the format rules of the document into which the text inset is placed. This is the default behavior for entities treated as text insets. You specify the other choices for formatting text insets with the rules `reformat as plain text` and `retain source document formatting`.

Synopsis and contexts

```
1. entity "ename" {
      is fm text inset “fname”;
      reformat using target document catalogs;
   . . .}

2. reader entity "ename" {
      is fm text inset “fname”;
      reformat using target document catalogs;
   . . .}
```

Arguments

ename	An SGML entity name.

See also

Related rules	“reformat as plain text,” (the previous section) “retain source document formatting” on page 430
Rules mentioned in synopses	“entity” on page 349 “is fm text inset” on page 411
General information on this topic	Chapter 13, “Translating Entities and Processing Instructions”

retain source document formatting

Use the `retain source document formatting` rule in an `entity` rule for an entity you want to translate as a text inset in FrameMaker+SGML. This specifies that the software should remove any element structure from the text inset, but keep the formatting of the source document, rather than reformatting it according to the rules of the document that contains the text inset. You specify the other choices for formatting text insets with the rules `reformat as plain text` and `reformat using target document catalogs`.

Synopsis and contexts

```
1. entity "ename" {
      is fm text inset “fname”;
      retain source document formatting;
   . . .}

2. reader entity "ename" {
      is fm text inset “fname”;
      retain source document formatting;
   . . .}
```

Arguments

ename	An SGML entity name.

See also

Related rules	"reformat as plain text" on page 429 "reformat using target document catalogs," (the previous section)
Rules mentioned in synopses	"entity" on page 349 "is fm text inset" on page 411
General information on this topic	Chapter 13, "Translating Entities and Processing Instructions"

specify size in

Use the `specify size in` rule only in an `element` rule for a graphic or equation element, to provide information the software needs when writing a document containing graphics and equations to SGML. This rule determines which of the `dpi` or the `impsize` attribute FrameMaker+SGML uses to indicate the size of a graphic or equation. The rule also indicates what units are used for `impsize` and the resolution in which sizes are reported is always 0.001. If there is no `specify size in` rule, FrameMaker+SGML uses the `dpi` attribute.

Synopsis and contexts

```
element "gi" {
   is fm graphic_or_eqn element ["fmtag"];
   writer type ["facetname"] specify size in units
. . .}
```

Arguments

gi	An SGML generic identifier.
graphic_or_eqn	One of the keywords: `graphic` or `equation`.
type	One of the rules `anchored frame`, `facet`, or `equation`. If `facet`, you must also supply the *facetname* argument. If *type* is `equation`, the rule applies to equation elements. If *type* is `facet`, the rule applies to a graphic element that contains only a single facet with the name specified by *facetname*. This occurs when the graphic element is an anchored frame containing only a single imported graphic object whose original file was in the *facetname* graphic format. You can use this rule with *type* set to

`facet` multiple times if you want the software to treat several file formats differently.

If *`type`* is `anchored frame`, the rule applies to a graphic element under all other circumstances.

`facetname` A facet name. You supply this argument if and only if *`type`* is `facet`.

`units` The units in which the size of the element is coded. Valid values: `cm`, `cc`, `dd`, `in`, `mm`, `pc`, `pi`, or `pt`.

Details

- Use this rule when you export FrameMaker+SGML documents to SGML documents.
- FrameMaker+SGML reports the size of the elements in the indicated units, at a fixed resolution of .001.

Examples

- Suppose your document has a graphic element, `graph`, containing an Anchored Frame sized to fit a FrameMaker+SGML-drawn circle with a diameter of 3.15 centimeters. Given the rule:

```
element "graph" {
   is fm graphic element;
   writer anchored frame specify size in cm;
}
```

  FrameMaker+SGML generates the attribute `height=”3.150cm” and attribute width=”3.150cm”`.

- However, with the same graphic, if the rule is:

```
element "graph" {
   is fm graphic element;
   writer anchored frame specify size in mm;
}
```

  FrameMaker+SGML generates `height=”31.500mm” and attribute width=”31.500mm”`.

- Given the rule:

```
element "graph" {
   is fm graphic element;
   writer anchored frame specify size in dpi with resolution 10;
}
```

  FrameMaker+SGML generates `dpi=80` for a graphic imported at 75 dpi.

See also

Related rules	“convert referenced graphics” on page 340 “entity name is” on page 352 “export to file” on page 359 “specify size in” on page 431
Rules mentioned in synopses	“element” on page 345 “is fm graphic element” on page 394 “is fm equation element” on page 392 “anchored frame” on page 333 “equation” on page 355 “facet” on page 364 “writer” on page 442
General information on this topic	Chapter 15, “Translating Graphics and Equations”

start new row

Use the `start new row` rule in the `element` rule for a table cell element to specify that an occurrence of the table cell element indicates that FrameMaker+SGML should start a new table row to contain that cell.

Synopsis and contexts

```
element "gi" {. . .
   is fm table cell element ["fmtag"];
   reader start new row ["name"];
. . .}
```

Arguments

gi	An SGML generic identifier.
fmtag	A FrameMaker+SGML element tag.
name	An optional name to identify this row

Details

- Your DTD may contain elements that you want to format as tables in FrameMaker+SGML even though the element hierarchy does not match that required by FrameMaker+SGML for tables. In such a situation, the nature of the element hierarchy may indicate where new rows should begin.

- In some cases, you can use a rule such as the following to indicate that a table cell starts a new row:

```
element "gi" {
   is fm table cell element;
   fm property column number value is "1";
}
```

With this rule, when FrameMaker+SGML encounters a *gi* element, it tries to place that element in the first column of the current table row. If there is already a cell in the first column of the current row, the software automatically creates a new row for *gi*. In this situation, you would not also need the `start new row` rule.

However, if there is not already a cell in the first column of the current row when the software encounters a *gi* element, it puts the *gi* cell in the current row and does not create a new row for it. This can happen if the table has a vertical straddle in the first column. When FrameMaker+SGML encounters a *gi* element on a row that should have a vertical straddle in the first column, with only the rule above, the software puts the *gi* element in the same row (because that cell isn't occupied). To guarantee a new row starts with the occurrence of *gi* instead, you should use this rule:

```
element "gi" {
   is fm table cell element;
   fm property column number value is "1";
   reader start new row;
}
```

Examples

- For a complete example using the `start new row` rule, see "Omitting explicit representation of table parts" on page 262.

See also

Related rules	"start vertical straddle," next
Rules mentioned in synopses	"element" on page 345 "is fm table part element" on page 409 "reader" on page 427
General information on this topic	Chapter 14, "Translating Tables"

start vertical straddle

Use the `start vertical straddle` rule inside the `element` rule for a table cell to specify that an occurrence of the cell element indicates the start of a vertical straddle.

Synopsis and contexts

```
element "gi" {. . .
   is fm table cell element ["fmtag"];
   reader start vertical straddle "name";
. . .}
```

Arguments

gi	An SGML generic identifier.
fmtag	A FrameMaker+SGML element tag.
name	A name associated with a table straddle. This name must occur in at least one corresponding `end vertical straddle` rule.

Details

- Your DTD may contain elements that you want to format as tables in FrameMaker+SGML even though the element hierarchy does not match that required by FrameMaker+SGML for tables. In such a situation, the nature of the element hierarchy may indicate where vertical straddles should begin and end. The `start vertical straddle` rule allows you to specify such elements.
- Use this rule in conjunction with the `end vertical straddle` rule. That rule specifies a table cell or row that indicates the end of the vertical straddle started by this rule.
- You give a name to the particular straddle started by *gi*. In the corresponding `end vertical straddle` rule or rules, you use the same name to specify that the element ends this vertical straddle.

Examples

- For an example of the use of this rule, see "Creating parts of a table even when those parts have no content" on page 264.

See also

Related rules	"start new row," (the previous section)
Rules mentioned in synopses	"element" on page 345 "is fm table part element" on page 409 "reader" on page 427
General information on this topic	Chapter 14, "Translating Tables"

table ruling style is

You use the `table ruling style is` rule to specify the ruling style for all tables.

Synopsis and contexts

```
reader table ruling style is "style";
```

Arguments

style	A ruling style for all tables. One of the keywords: None, Double, Medium, Thick, Thin, or Very Thin.

Details

- This rule specifies the ruling style applied to all tables. When working with the CALS table model, you can use the `frame`, `colsep`, and `rowsep` attributes to determine whether or not portions of a table have rulings. However, these attributes have boolean values. Consequently, you can only use them to say whether or not a table has a ruling, not what type of ruling to use if it does have one. In this situation, you could use the `table ruling style is` rule to set the ruling style for all tables.
- FrameMaker+SGML considers the ruling style set with this rule as custom ruling. If you re-import formats to the FrameMaker+SGML document and remove overrides, the ruling style set with this rule will remain. If possible, therefore, you should use table formats to specify ruling styles.

Examples

- To specify that all tables should use the Thick ruling style, use this rule:

```
reader table ruling style is "Thick";
```

See also

General information on this topic	Chapter 14, “Translating Tables”

unwrap

Use the `unwrap` rule when you do not want to preserve an element on translation from one representation to another. If you specify that FrameMaker+SGML should unwrap an element (*gi* or *fmtag*), the software places the element’s content as part of the content of the element’s parent element, but does not make an element for *gi* or *fmtag* itself.

Synopsis and contexts

```
1. element "gi" unwrap;
2. fm element "fmtag" unwrap;
```

Arguments

gi	An SGML generic identifier.
fmtag	A FrameMaker+SGML element tag.

Details

- When FrameMaker+SGML encounters an element to be unwrapped, it does not insert a corresponding element into the document it is creating. Instead, it inserts the content of an unwrapped element.
- If you use this rule to unwrap FrameMaker+SGML cross-reference elements or system variable elements, those elements become text in the resulting SGML document.
- When importing a DTD or exporting an EDD, FrameMaker+SGML does not generate an element definition or declaration corresponding to an element that is unwrapped. Furthermore, when an element uses the unwrapped element in its definition, the software replaces the name of the unwrapped element with its content model or general rule in the general rule or content model of the element that used it or replaces it with the list of its children in an exception. You can change this behavior by using the `preserve fm element definition` rule.
- You cannot use the `unwrap` rule with any other subrule of the `element` or `fm element` rules. For example, you cannot specify that an SGML element both be unwrapped and be translated to a FrameMaker+SGML element.

Examples

- An SGML document used to produce both the student's and teacher's edition of a textbook might include an `ANSWER` element used for answers to exercises. In producing the teacher's edition of the textbook, this element might be unwrapped into FrameMaker+SGML as text. An SGML API client could associate this element with the condition tag `Answer`.
- Suppose a DTD contains the following declarations:

  ```
  <!ELEMENT wrapper - - (a, b)>
  <!ELEMENT x - - (p, q, wrapper, r)>
  <!ELEMENT y - - (#PCDATA) +(wrapper)>
  ```

  and you have this rule:

  ```
  element "wrapper" unwrap;
  ```

  FrameMaker+SGML would generate the following element definitions:

  Element (Container): X
  General rule: P, Q, A, B, R

  Element (Container): Y
  General rule: <TEXT>
  Inclusions: A, B

See also

Related rules	“preserve fm element definition” on page 422 “drop” on page 342
Rules mentioned in synopses	“element” on page 345 “fm element” on page 367
General information on this topic	Chapter 12, “Translating Elements and Their Attributes”

use processing instructions

See “generate book” on page 373.

use proportional widths

Use the `use proportional widths` rule to indicate that when FrameMaker+SGML writes the width of table columns, it should use proportional measurements. By default, if the software writes the width of table columns, it uses absolute measurements.

Synopsis and contexts

```
writer use proportional widths;
```

Arguments

None.

Details

- If you use this rule when writing an attribute indicating the width of one or more columns in a table, FrameMaker+SGML writes values such as `"25*"`, where the asterisk `*` indicates a proportional measurement, instead of values such as `"0.25in"` which are absolute measurements.
- If you use this rule, you can also use the `proportional width resolution is` rule to determine what number the values add to. Without the `proportional width resolution is` rule, the proportional measurements add to 100.

Examples

- Assume you do not use the `proportional width resolution is` rule, but have this rule:

  ```
  writer use proportional widths;
  ```

  Further assume you have a 5-column table whose first two columns are 1 inch wide and whose last three columns are 2 inches wide. If the column widths are written to the

colwidth attribute of the SGML table element, then FrameMaker+SGML creates this start-tag for that table:

```
<table colwidth="12.5* 12.5* 25* 25* 25*">
```

- Assume you have the same table as in the last example and you use this rule:

```
writer {
   use proportional widths;
   proportional width resolution is "8";
}
```

FrameMaker+SGML writes this start-tag for the table:

```
<table colwidth="1* 1* 2* 2* 2*">
```

See also

Related rules	"proportional width resolution is" on page 426
General information on this topic	Chapter 14, "Translating Tables"

value

Use the value rule to translate the value of an SGML attribute to the value of a particular FrameMaker+SGML property or to a particular choice for a FrameMaker+SGML choice attribute. The SGML attribute's declared value must be a name token group or NOTATION and a name token group.

Synopsis and contexts

```
1. value "token" subrule;

2. attribute "attr" {. . .
      value "token" subrule;
      . . .}

3. element "gi" {. . .
      attribute "attr" {. . .
         value "token" subrule;
         . . .} . . .}
```

Arguments

token	A token in a name token group.
attr	The name of an SGML attribute.
gi	An SGML generic identifier.
subrule	One of the following:

`is fm value`, page 413, translates an SGML value to a particular choice for a FrameMaker+SGML choice attribute.

`is fm property value`, page 398, translates an SGML value to the value of a particular FrameMaker+SGML property.

Details

- The rule can be used at the highest level to set a default, within a highest-level attribute rule to set the default for all attributes that use that token, or within an `element` rule to set the default for a particular token within a particular attribute in that element.

Examples

- To rename the FrameMaker+SGML `import by reference or copy` property as the `refcopy` attribute, and to also change the name tokens, use this rule:

```
attribute "refcopy" {
   is fm property import by reference or copy;
   value "r" is fm property value reference;
   value "c" is fm property value copy;
}
```

- If the token list `(r | b | g)` is used by multiple attributes, you can use these rules to translate the individual tokens consistently:

```
value "r" is fm value "Red";
value "b" is fm value "Blue";
value "g" is fm value "Green";
```

- If the token list `(r | b | g)` is used by several attributes as above, but by the `bird` element differently, you can add this rule to the above rules:

```
element "bird" {is fm element;
] attribute "species" {
   value "r" is fm value "Robin";
   value "b" is fm value "Blue Jay";
   value "g" is fm value "Goldfinch";
   }}]
```

See also

Related rules	"is fm value" on page 413 "is fm element" on page 391
Rules mentioned in synopses	"attribute" on page 335 "element" on page 345
General information on this topic	Chapter 12, "Translating Elements and Their Attributes"

value is

See "fm property" on page 370.

write sgml document

By default, when you save a FrameMaker+SGML document to SGML, the software writes out the document instance, any declarations for the internal DTD subset, and a DOCTYPE statement which references the external DTD subset, but not an SGML declaration or the declaration within the external DTD subset. You can use this rule to confirm the default behavior.

Synopsis and contexts

```
writer write sgml document;
```

Arguments

None.

Details

- You cannot use the `write sgml document` rule and the `write sgml document instance only` rule in the same read/write rules file.

See also

Related rules

"external dtd" on page 363
"include dtd" on page 379
"include sgml declaration" on page 380
"write sgml document instance only," next

write sgml document instance only

By default, when you save a FrameMaker+SGML document to SGML, the software writes out the document instance, any declarations for the internal DTD subset, and a DOCTYPE statement which references a file for the external DTD subset. It doesn't write an SGML declaration. This rule causes the software to write the document instance only--no external or internal DTD and no SGML declarations.

Synopsis and contexts

```
writer write sgml document instance only;
```

Arguments

None.

Details

- By default, when you translate a FrameMaker+SGML document to SGML, as its last step the software runs the SGML parser on the SGML document to check its validity. If you use this rule, FrameMaker+SGML does not write a complete SGML document and so does not send the result through the parser.
- You cannot use the `write sgml document instance only` rule in the same read/write rules file as any of the `write sgml document`, `include dtd`, or `include sgml declaration` rules.

See also

Related rules

"external dtd" on page 363
"include dtd" on page 379
"include sgml declaration" on page 380
"write sgml document," (the previous section)

writer

The `writer` rule indicates a rule that applies only on export of a FrameMaker+SGML document to SGML. It can be used at the highest level to set a default or within an `element` rule to specify a subrule for that element.

Synopsis and contexts

```
1. writer {. . .
      subrule;
   . . .}

2. element "gi" {. . .
      writer {. . .
         subrule;
   . . .} . . .}
```

Arguments

gi — An SGML generic identifier.

subrule — Valid subrules:

`anchored frame`, page 333, tells FrameMaker+SGML what to do with graphic elements other than those with a single non-internal FrameMaker+SGML facet. Allowed only within an `element` rule for a graphic element.

`character map`, page 337, determines the correspondence between individual characters in the FrameMaker+SGML and SGML character sets. Allowed only at the highest level.

`convert referenced graphics`, page 340, tells the software to create new files for graphic files that were imported by reference.`drop content`, page 344, exports a FrameMaker+SGML element without its contents. Allowed only within an `element` rule.

`equation`, page 355, tells FrameMaker+SGML what to do with equation elements. Allowed only with an `element` rule for an equation element.

`external dtd`, page 363, specifies an external DTD to use. Allowed only at the highest level.

`facet`, page 364, tells FrameMaker+SGML what to do with a graphic element that has a single non-internal FrameMaker+SGML facet. Allowed only with an `element` rule for a graphic element.

`[do not] include dtd`, page 379, specifies information to exclude or include in the written document. Allowed only at the highest level.

`[do not] include sgml declaration`, page 380, specifies information to exclude or include in the written document. Allowed only at the highest level.

`line break`, page 417, specifies treatment of line breaks not handled by the parser on export. Allowed at the highest level or within an `element` rule.

`[do not] output book processing instructions`, page 422, specifies whether or not to create processing instructions that identify book components when writing a FrameMaker+SGML book as an SGML document. Allowed only at the highest level.

`proportional width resolution is`, page 426, specifies the total value to which proportional widths for table columns add up. Allowed only at the highest level.

`use proportional widths`, page 438, specifies that the software should use proportional values in describing the widths of table columns. Allowed only at the highest level.

`write sgml document`, page 441, specifies that an entire SGML document should be written, not just the document instance. This is the default. Note that the external DTD subset is not written to the file. Instead, a DOCTYPE statement with a reference to the external DTD file is written. Allowed only at the highest level.

`write sgml document instance only`, page 441, specifies that only the document instance should be written, not the DTD and SGML declaration. Allowed only at the highest level.

Examples

- To tell FrameMaker+SGML not to use processing instructions to identify book components when writing a FrameMaker+SGML book as an SGML document, use this rule:

```
writer do not output book processing instructions;
```

- Assume you want all graphics to be exported in TIFF format. Further assume that some of your graphic elements were imported from the TIFF format. For these elements you don't want to create a new external data entity. To accomplish this, use these rules:

```
element "graphic" {
   is fm graphic element;
 writer facet default{
 convert referenced graphics;
 export to file “$(entity) .tif as “TIFF”;
   writer anchored frame
      export to file "$(entity).tif" as "TIFF";
}
```

A Conversion Tables for Adding Structure to Documents

You can set up a conversion table to help end users automate the task of adding structure to documents. The conversion table uses paragraph and character formats to identify which document objects to wrap in elements, and element tags to identify which child elements to wrap in parent elements. A user wraps all of a document's contents in one move by applying a structure command to the document and referring to one of your conversion tables.

This appendix describes how to set up a conversion table and define object and element mapping in it. For information on the commands for adding structure to documents, see the FrameMaker+SGML user's manual

How a conversion table works

A conversion table contains rules for mapping between document objects and elements and between child elements and parent elements. The table is a regular FrameMaker+SGML table, with at least three columns and one body row. Each body row holds one rule.

The first column in a conversion table specifies a document object, a child element, or a sequence of child elements or paragraphs to wrap in an element. A *document object* is a paragraph, text range, table, table part (such as heading or row), equation, variable, footnote, Rubi group, Rubi text, marker, cross-reference, text inset, or graphic (anchored frame or imported graphic object).

The second column in the table specifies the element in which you want to wrap the object, child element, or sequence. The third column can specify an optional *qualifier* to use as a temporary label for the element in rules that are applied later. For example:

Wrap this object	In this element	With this qualifier
P:BulletItem	Item	Bullet
E:Item[Bullet]+	BulletList	

The first column uses a one-letter code and usually a tag to identify an object or element.

The second column specifies the element in which to wrap the object or element.

The third column can provide a label for the new element to be used in later rules.

To add structure to a document or book, an end user chooses the Structure Current Document, Structure Documents, or Structure Current Book command from the File>Utilities submenu and refers to one of the conversion tables.

When someone adds structure to a document manually, he or she typically begins with the lowest-level components and works up to the highest level. For example, to add structure to a chapter an end user might start by wrapping sub-paragraph objects like text ranges and tables, then wrap the contents of paragraphs together in `Paragraph` elements, then wrap sequences of `Head` and `Paragraph` elements in `Section` elements, and so on until the entire document is wrapped in a single highest-level `Chapter` element.

The process of adding structure with a conversion table is similar to adding structure manually. FrameMaker+SGML begins by applying rules to document objects below the paragraph level, then applies rules at the paragraph level, and proceeds through successively higher levels. The process stops when FrameMaker+SGML reaches a single highest-level element or when no more rules can be applied. To understand this process, it helps to have manually structured a document.

Using the sample table above, FrameMaker+SGML first wraps each paragraph with the paragraph format `BulletItem` in an element called `Item` and gives the element a qualifier called `Bullet`. Then it wraps each `Item` element with the qualifier `Bullet` in a parent element called `BulletList`.

FrameMaker+SGML tries to order the rules as much as possible. If a rule needs a building block that is generated by a later rule, the later rule is run first so that all of the building blocks in the first rule are available. To make a conversion table easy to interpret for a human reader, you may want to write the rules in the order they should be applied.

Setting up a conversion table

You can have FrameMaker+SGML generate an initial conversion table for you from an unstructured document or book, or you can create a conversion table entirely from scratch. If you already have a document that end users need to add structure to, or a document that is similar to one users will add structure to, you'll probably want to let FrameMaker+SGML generate the initial table. You can modify the rules in the table as necessary.

After creating a conversion table, you can update it from other unstructured documents. Updating a table adds rules for any objects in the document that are not yet in the table.

A conversion table document can have the conversion table itself (which may be split up into several tables) and text or graphics you want to include for documenting the rules. It cannot have any tables other than conversion tables. You need to save the document before it can be used for adding structure to other documents or books.

Each body row in a conversion table holds one mapping rule. FrameMaker+SGML reads only the information in the first thee columns of the body rows, so you can use additional columns and headings and footings for comments about rules.

For information on defining and modifying the rules in a table, see "Adding or modifying rules in a conversion table" on page 448.

Generating an initial conversion table

You can have FrameMaker+SGML generate a conversion table from an unstructured document. This is the easiest way to begin a new conversion table.

To generate an initial conversion table, choose Generate Conversion Table from the File>Developer Tools submenu in a document with objects you want to structure. Select Generate New Conversion Table in the dialog box and click Generate.

The software looks through the flows on body pages in the document and compiles a list of every object that can be structured. For each object, it gives the object type and the format tag used in the document (if the object has a format), and maps the object to an element. The element tag is the same as the format tag, or if the object does not have a format, the element tag is a default name such as `CELL` or `BODY`. If necessary, FrameMaker+SGML removes parentheses and other characters to create an element tag that is valid.

The initial conversion table gives you a first pass through the document, identifying objects to wrap in elements. It does not identify child elements to wrap in parent elements—you need to add those rules to the table yourself.

This is an example of an initial conversion table:

Wrap this object	In this element	With this qualifier
P:Head1	Head1	
P:Head2	Head2	
P:Body	Body	
P:Code	Code	
SV:Current Date \(Long\)	CurrentDateLong	
C:Code	cCode	
TC:	CELL	
TR:	ROW	

For details on the object type identifiers used in the table (such as `P:` and `TC:`), see "Identifying a document object to wrap" on page 450.

Note that if there are conflicts in a format tag from the unstructured document, an object type identifier in lowercase is prepended to any duplicate element tag. In the example above, the element tag for text ranges with the `Code` character format is `cCode` because the document also has a paragraph format called `Code`.

When you create an initial table, FrameMaker+SGML does not examine the document's format catalogs—it looks only at objects actually used in the document. For this reason, the table may not be as complete as you need. You may want to update the table from a set of documents that together provide all or most of the objects you need rules for. You can also add and modify rules manually.

Setting up a conversion table from scratch

You can set up a regular FrameMaker+SGML table to serve as a conversion table. The table must appear on a body page in its own document. The document and table can be structured or unstructured. Begin a conversion table this way if you do not yet have an unstructured document to use for generating the table.

To set up a conversion table from scratch, create a new document and insert a table with at least three columns and one body row. The table can have any number of heading or footing rows.

You can divide a conversion table into several smaller tables. This is helpful when you have many rules and want to organize the rules in groups. Each table must have at least three columns and one body row. You can add explanatory heads and paragraphs between the tables to document the rules. Do not include tables that are not conversion tables.

Updating a conversion table

After creating a conversion table, you may want to update the table from at least one other unstructured document to get a more complete list of objects. FrameMaker+SGML adds a rule for each object from the document that is not already listed in the table.

To update a conversion table, choose Generate Conversion Table from the File>Developer Tools submenu in a document with the objects you want to structure. Select the name of the conversion table document in the Update Conversion Table popup menu and click Generate.

When you update a conversion table, the process that FrameMaker+SGML goes through is similar to the process of generating an initial table. The software does not examine the document's format catalogs—it looks only at objects actually used in the document.

Adding or modifying rules in a conversion table

Each body row in a conversion table holds one mapping rule. Follow these steps to define a mapping rule:

1. **In the first column, identify a document object, a child element, or a sequence of child elements or paragraphs to wrap.**

   You use a one- or two-letter code to identify the type of item and, in most cases, a format or element tag to narrow the definition. See "Identifying a document object to wrap" on page 450, "Identifying an element to wrap" on page 451, or "Identifying a sequence to wrap" on page 452.

2. **In the second column, specify an element in which to wrap the object, child element, or sequence.**

   Type one valid element tag. If you are writing rules for a document that already has element definitions, use tags from the document's Element Catalog.

   If you are wrapping a table part, graphic, or inset, FrameMaker+SGML always wraps all instances of the object in the same kind of element. The element has a default tag, such as `CELL`, `BODY`, `GRAPHIC`, or `INSET`. Type a different tag in the second column only if you want to override the default tag.

   You can also give an element an attribute with a value. For details, see "Providing an attribute for an element" on page 453.

3. **(Optional) In the third column, add a qualifier for the new element tag.**

   A qualifier is a temporary label that you can attach to an element tag for the structuring process. If you wrap the element in a parent element in a later rule, you include the qualifier tag with the element tag. For details, see "Using a qualifier with an element" on page 454.

To make a conversion table easy to read and to help you think through the process, we recommend that you put the rules in order from the lowest level to the highest. In the first rows of the table, write rules that wrap individual document objects such as text ranges, tables, and paragraphs; next add rules that wrap child elements in parent elements; then add rules that wrap sequences in elements; and finally add rules that wrap elements in one highest-level element.

Every flow in a document must have a highest-level element, and the element can be different for each flow.

About tags in a conversion table

Format and element tags in a conversion table are case-sensitive and must be specified the way they are defined in their catalogs. Qualifier tags are also case-sensitive, and two occurrences of one qualifier must match exactly. The following characters are not allowed in an element tag, but can appear in a format or qualifier tag if you precede them with a backslash (\) in the table:

```
( ) & | , * + ? % [ ] : \
```

A space character does not need to be preceded with a backslash. For example, you can write the tag `Format A`.

You can use a percentage sign (%) as a wildcard character in a format or element tag to match zero, one, or more characters. For example, `P:%Body` matches paragraphs with the format tag `Body`, `FirstBody`, or `BulletBody`.

Identifying a document object to wrap

To identify a document object to wrap in an element, type an object type identifier and (optionally) a format tag in the first column of the table. Separate the identifier and format tag with a colon.

FrameMaker+SGML finds all the objects with that type and format and wraps them in the element you specify in the second column of the table. If you leave the format tag out of the rule, FrameMaker+SGML finds all the objects with the specified type that are not identified in other conversion rules.

For example:

Wrap this object	In this element
P:Body	Para
T:RulesTbl	RulesTbl
T:	StandardTbl
Q:Small	SmallEqns

This rule wraps all tables not named in other rules, regardless of format tag.

These are the object type identifiers and format tags you can use:

Object type	Identifier	Format tag
Paragraph	`P:`	Paragraph format tag
Text range	`C:`	Character format tag
Table	`T:`	Table format tag
Table title	`TT:`	(none)
Table heading	`TH:`	(none)
Table body	`TB:`	(none)
Table row	`TR:`	(none)
Table cell	`TC:`	(none)
System variable	`SV:`	Variable format name
User variable	`UV:`	Variable format name
Graphic (anchored frame or imported object)	`G:`	(none)
Footnote	`F:`	Location of footnote: `Table` or `Flow`
Rubi group	`RG:`	(none)
Rubi text	`R:`	(none)
Marker	`M:`	Marker type

Object type	Identifier	Format tag
Cross-reference	`X:`	Cross-reference format name
Text Inset	`TI:`	(none)
Equation	`Q:`	Size of equation: `Small`, `Medium`, or `Large`

Table parts, graphics, and text insets do not have any formatting information, so FrameMaker+SGML wraps all instances of those objects in the same kind of element. The element has a default tag, such as `CELL`, `BODY`, `GRAPHIC`, or `INSET`. (Specify a different tag in the second column to override the default tag.)

You can write identifiers and the keywords for footnote location or equation size in any combination of uppercase and lowercase letters. The names of formats and marker types are case-sensitive, however, and must be typed the way they are specified in their catalogs.

A system variable can be wrapped in a variable element but a user variable cannot. If you identify a user variable, FrameMaker+SGML wraps it in a container element with the tag specified in the second column.

FrameMaker+SGML wraps a text inset in a container.

Identifying an element to wrap

To identify a child element to wrap in a parent element, type the object type identifier `E:` followed by an element tag and (optionally) a qualifier in brackets in the first column of the table. The qualifier must already be defined for the element in a rule applied earlier.

FrameMaker+SGML finds all instances of the element and wraps them in the element you specify in the second column of the table. You can omit the element tag if you include a qualifier.

For example:

Wrap this object	In this element
E:Item[Bullet]	BulletItem
E:[1Head]	ChapHead

This rule wraps all elements with the qualifier 1Head not named in other rules.

You can type the `E:` identifier in either uppercase or lowercase. The element tags are case-sensitive, however, and must be typed the way they are specified in their catalog. You can even omit the `E:` identifier—when FrameMaker+SGML reads an object name with no identifier, it assumes the object is an element.

To identify a table child element to wrap in a table parent element, type the object identifier `TE:` followed by `E:`, an element tag, and (optionally) a qualifier in brackets in the first

column of the table. This allows you to name a table element from one or more child elements, rather than naming it from a table format tag (with the `T:` identifier).

For example:

Wrap this object	In this element
TB:RulesBody	RulesBody
TE:E:RulesBody	RulesTbl

This rule wraps RulesBody table child elements in a RulesTbl table element.

Most often, you wrap multiple elements together in one parent. You can use `E:` or `TE:` to identify a sequence of elements for this. For more information, see "Identifying a sequence to wrap," next. For more information on qualifiers, see "Using a qualifier with an element" on page 454.

Identifying a sequence to wrap

You can wrap a sequence of child elements in a parent element. For example, you might wrap a `Head` element followed by one or more `Paragraph` and `List` elements in a higher-level `Section`.

You can also wrap a sequence of unwrapped paragraphs in an element. For example, you might wrap a sequence of paragraphs with the format tag `Body` all in one `Note` element. (With other unwrapped document objects such as tables, graphics, and text ranges, you can wrap only one object in an element.)

To identify a sequence to wrap, specify object type identifiers and element tags or paragraph format tags, and use symbols to further describe the sequence. You can mix elements and unwrapped paragraphs together in one specification.

These are the symbols you can use:

Symbol	Meaning
Plus sign (+)	Item is required and can occur more than once.
Question mark (?)	Item is optional and can occur once.
Asterisk (*)	Item is optional and can occur more than once.
Comma (,)	Items must occur in the order given.
Ampersand (&)	Items can occur in any order.
Vertical bar (|)	Any one of the items in the sequence can occur.
Parentheses	Beginning and end of a sequence.

The symbols available are the same connectors, occurrence indicators, and parentheses used in general rules in an EDD. For more information on the symbols, see "Writing an EDD general rule" on page 99.

For example:

To identify this sequence	Use this specification
One or more `Item` elements	`Item+`
An element tagged `Item[Bullet]` followed by one or more unwrapped paragraphs tagged `Bullet`	`E:Item[Bullet], P:Bullet+`
A `ChapNum` element followed by a `ChapName` element	`ChapNum, ChapName`
A `Head` element followed by zero or more `Paragraph`, `BulletList`, or `NumberList` elements	`Head, (Paragraph|BulletList|NumberList)*`
An `Item[FirstNItem]` element followed by one or more `Item[NItem]` elements	`Item[FirstNItem], (Item[NItem])+` or `[FirstNItem], ([NItem])+`
A `RulesTitle` table title element followed by a `RulesBody` table body element	`TE:E:RulesTitle, E:RulesBody`

Providing an attribute for an element

When you specify an element in the second column of the table, you can provide an attribute for the element. In the structured document, all the element instances will have the attribute name and value.

To provide an attribute for an element, type the attribute name and value in brackets after the element tag in the second column of the table. Separate the name and value with an equal sign, and enclose the value in double quotation marks.

For example:

Wrap this object	In this element
P:Intro	Para[Security="Unclassified"]
P:Important	Note[Label="Important"]
E:Item+	List[Type="Numbered"]

If the unstructured document has an Element Catalog with an element and attribute matching the one you're providing, the attribute is the type specified in the catalog. If the attribute does not match an attribute already defined, the type is string.

If you need to use a double quotation mark in an attribute value, escape the quotation mark with a backslash (\). Other restrictions on characters are determined by the attribute's type. (The string type allows any arbitrary text string.) For information on these restrictions, see "Attribute type" on page 160.

To give an element more than one attribute, separate the attribute definitions with an ampersand (&). For example, this specification gives the element a `Type` attribute with the value `Numbered` and a `Content` attribute with the value `Procedure`:

```
List [Type="Numbered" & Content="Procedure"]
```

For an example of an attribute that maintains formatting information from a qualifier, see "Using a qualifier with an element," next.

Using a qualifier with an element

Qualifiers act as temporary labels that preserve formatting information from the unstructured document until all elements have been wrapped. Qualifiers are used only in the conversion table—they do not show up in a final structured document.

To use a qualifier with an element specified in the second column of the table, type the qualifier tag in the third column. Then when you wrap the element in a later rule, type the qualifier tag in brackets after the element tag in the first column. Spell and capitalize the qualifier the same way in the two places. FrameMaker+SGML keeps track of qualifiers separately from elements, so you can use the same tag for an element and its qualifier.

For example:

Wrap this object	In this element	With this qualifier
P:BulletItem	Item	bulleted
P:NumberItem	Item	numbered
E:Item[bulleted]+	BulletList	
E:Item[numbered]+	NumberList	

First specify the qualifier for the element.

Then include the qualifier with the element in later rules.

In the example above, an unstructured document has both bulleted items and numbered items, with paragraph formats called `BulletItem` and `NumberItem`. When adding structure to the document, you want to wrap all the items in an `Item` element with a parent element of either `BulletList` or `NumberList`. To do this, you need to keep the

`BulletItem` and `NumberItem` formatting designations long enough to determine in which list to wrap the items. The conversion table first associates qualifiers called `bulleted` and `numbered` with new `Item` elements. Then it wraps each `Item` element in either a `BulletList` or a `NumberList`, as specified by its qualifier.

Note that if you specify an attribute for formatting information in the second column, you cannot use the attribute as a label for preserving formatting during the conversion process. You still need to use the qualifier. For example:

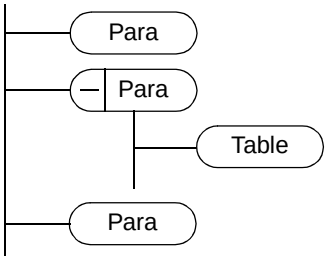
Wrap this object	In this element	With this qualifier
P:BulletItem	Item	bulleted
P:NumberItem	Item	numbered
E:Item[bulleted]+	List[Type="Bulleted"]	
E:Item[numbered]+	List[Type="Numbered"]	

Handling special cases

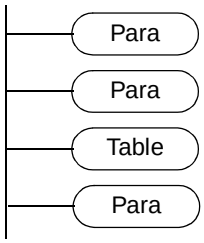
You may need to accommodate a few special circumstances or requirements in a conversion table.

Promoting an anchored object

In an unstructured FrameMaker+SGML document, a table or an anchored graphic must be anchored in a paragraph. The anchor specifies which paragraph to keep the object with as an author continues to edit the document. When a user adds structure to the document, the table or graphic normally becomes a child of the paragraph with the anchor, like this:



In a structured document, you often want a table or graphic element to be at the same level as its surrounding paragraph elements. FrameMaker+SGML can break the table or graphic out of its paragraph and promote the element to be a sibling of the paragraphs, like this:



To break a table or graphic out of its paragraph and promote it one level, add the keyword `promote` in parentheses after the element tag for the table or graphic. (The keyword is not case-sensitive.) For example:

Wrap this object	In this element
T:Table	Table (promote)

Note that FrameMaker+SGML promotes the object at the location of the anchor symbol in the paragraph. If the symbol is in the middle of the paragraph, the structured document will have half of the paragraph, then the table, and then the other half of the paragraph. Typically, you want the symbol to be at the end of the paragraph.

Flagging format overrides

An unstructured document may have *format overrides*. This happens when someone uses the Paragraph or Character Designer to make formatting changes to a paragraph or text range but does not save the changes in the catalog format.

When an end user adds structure to a document, FrameMaker+SGML does not normally identify format overrides. You can have FrameMaker+SGML flag all element instances in the document that have overrides so that the user can find the overrides and decide how to handle them in a structured context.

To flag format overrides, add the rule `flag paragraph format overrides` or `flag character format overrides` to the first column of the table. (The rule is case-insensitive.) This is a general instruction for the table, so you do not add anything to the second and third columns. For example:

Wrap this object	In this element
flag paragraph format overrides	
flag character format overrides	

At each element instance that has an override in the document, FrameMaker+SGML adds an attribute called `Override` with the value `Yes`.

Wrapping untagged formatted text

It is possible for someone to format a text range by applying commands from the Font, Size, and Style submenus in the Format menu—and not use a character format at all. This leaves the text formatted but without a tag that you can refer to in your conversion table.

You can have FrameMaker+SGML find text that has been formatted with the submenu commands and wrap it in a “catch-all” element. After adding structure to a document, the end user will probably want to look at these instances and change them to other elements (such as `Emphasis`) that more specifically describe the type of formatting.

To wrap untagged formatted text, add the rule `untagged character formatting` to the first column of the table and add an element to the second column. (The rule is case-insensitive.) For example:

Wrap this object	In this element
untagged character formatting	UntaggedText

This might also be useful while you are developing a conversion table. You can add structure to a sample document with this rule to see if the document has any untagged formatting.

Nesting object elements

Typically, a non-paragraph object such as a table or graphic is wrapped in an object element and then wrapped in a paragraph element. You can also wrap the object in more than one level below the paragraph. Sometimes you need to do this to conform to a DTD that requires more hierarchy, or you may just want to be able to use two objects together.

To nest object elements in a paragraph, define each mapping in a separate rule in the table. For example:

Wrap this object	In this element
M:Index	Index
G:	Graphic
E:Index & E:Figure	Figure

In the example above, the rules wrap an index marker in an `Index` element and a graphic anchor in a `Graphic` element, and then they wrap the two elements together in a `Figure` text range element. This way, the graphics in a structured document will automatically have a marker identifying a location to be included in an index.

Building table structure from paragraph format tags

When FrameMaker+SGML adds structure to tables, it normally wraps all instances of a table part in the same kind of element and uses a default name for the element, such as `CELL`, `ROW`, `HEADING`, or `BODY`. You can override the default name by providing a different element tag in the second column of the conversion table.

If you want to have more than one kind of element for a particular table part, you can build the structure up from the format tags used in the cells or titles. This allows you to distinguish between different formatting used in different instances of a single table part. For example, a table may have a few special body rows with italicized text that marks divisions in the table. Or a table may have two titles, one of them a subtitle in a different font size.

To build table structure from paragraph format tags, for each cell or title rule use the `TC:` or `TT:` type identifier followed by the `P:` identifier and a format tag in the first column of the table. For example:

Wrap this object	In this element
TC: P:DividerCell	DividerCell
TC: P:BodyCell	BodyCell
TR:DividerCell+	ROW
TR:BodyCell+	ROW
TB:Row+	BODY

In the example above, the rules map cells that use a `DividerCell` paragraph format in an element called `DividerCell` and map cells that use a `BodyCell` paragraph format in

an element called `BodyCell`. Then they wrap both kinds of cell elements in the same default `ROW` element and continue the wrapping normally.

Testing and correcting a conversion table

You should test and correct a conversion table as you develop it. To do this, prepare a sample document that represents the type of documents the table will apply to, and use the conversion table to add structure to the sample. Make sure your sample document has all of the document objects that the final documents may contain.

When a structure command reads a conversion table, it identifies any syntax errors in the rules and displays the errors in a log file. Correct the table and test it again until no more errors are found.

You may find it helpful to wrap only document objects for your first testing pass, without wrapping in higher levels of hierarchy. When you're sure that the rules for wrapping individual objects are correct, start writing and testing the rules to wrap elements and sequences in parent elements.

B The CALS Table Model

The CALS table model is a specific set of element and attribute declarations for defining tables, defined in “Markup Requirements and Generic Style Specification for Electronic Printed Output and Exchange of Text,” MIL-M-28001B. If your SGML documents use these elements and attributes or some simple variations of them, FrameMaker+SGML can translate them to tables and table parts without the assistance of read/write rules. The CALS model can be interpreted in various ways. This appendix describes the CALS elements and attributes as they are interpreted by FrameMaker+SGML.

Some attributes are common to several elements in the description of the table. In these cases, attribute values are inherited in the element hierarchy. The values of attributes associated with `<colspec>` and `<spanspec>` elements act as though they were on the parent element for inheritance purposes. This is, if a `<tgroup>` element has two `<colspec>` child elements and a `<thead>` child element, the attributes of the `<colspec>` elements apply to the `<thead>` element unless that element has its own `<colspec>` elements with attribute values that override the inherited ones. If you want to change how FrameMaker+SGML processes any attribute of a `<colspec>` or `<spanspec>` element, you refer to the attribute as a formatting property.

In the CALS model, the `<table>` element has an `<orient>` attribute. This attribute is not supported in FrameMaker+SGML, because there is no way in a FrameMaker+SGML table to specify orientation on the page.lk

FrameMaker+SGML properties that DO NOT have corresponding CALS attributes

FrameMaker+SGML Property	For FrameMaker+SGML Elements of Type	Corresponding CALS Attribute
column widths	table (CALS: tgroup)	(none)

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 CALS -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 (in FrameMaker+SGML dialog boxes, “ is also used, but not for “column width” formatting property)
millimeter	mm
pica	pc (or pi)
point	pt

FrameMaker+SGML Property	For FrameMaker+SGML Elements of Type	Corresponding CALS Attribute
maximum height	row	(none)

Maximum height of a row in a table.

minimum height	row	(none)

Minimum height of a row in a table.

row type	row	(none)

Whether the associated table row is a heading, footing, or body row, or the associated table cell occurs in a row of that type.

horizontal straddle	cell	(none)

How many columns this straddle cell spans

vertical straddle	cell	(none)

How many rows this straddled cell spans

Element and attribute definition list declarations

The element and attribute declarations as used by FrameMaker+SGML are as follows:

```
<!ENTITY % yesorno "NUMBER">

<!ELEMENT table - - (title?, tgroup+)>
<!ATTLIST table
 colsep   %yesorno; #IMPLIED
   frame    (all|top|bottom|topbot|sides|none) #IMPLIED
   pgwide   %yesorno; #IMPLIED
   rowsep   %yesorno; #IMPLIED
   tabstyle NMTOKEN   #IMPLIED
>

<!ELEMENT title - - (#PCDATA)>

<!ELEMENT tgroup - O (colspec*, spanspec*, thead?, tfoot?, tbody)>
<!ATTLIST tgroup
   align       (left|center|right|justify|char) #IMPLIED
   char        CDATA     #IMPLIED
   charoff     NUTOKEN   #IMPLIED
   colsep      %yesorno; #IMPLIED
   cols     NUMBER    #REQUIRED
   rowsep      %yesorno; #IMPLIED
   tgroupstyle NMTOKEN   #IMPLIED
>

<!ELEMENT colspec - O EMPTY>
<!ATTLIST colspec
   align    (left|center|right|justify|char) #IMPLIED
   char     CDATA     #IMPLIED
   charoff  NUTOKEN   #IMPLIED
   colname  NMTOKEN   #IMPLIED
   colnum   NUMBER    #IMPLIED
   colsep   %yesorno; #IMPLIED
   colwidth CDATA     #IMPLIED
   rowsep   %yesorno; #IMPLIED
>
```

```
<!ELEMENT spanspec - O EMPTY>
<!ATTLIST spanspec
   align    (left|center|right|justify|char) #IMPLIED
   char     CDATA     #IMPLIED
   charoff  NUTOKEN   #IMPLIED
   colsep   %yesorno; #IMPLIED
   nameend  NMTOKEN   #REQUIRED
   namest   NMTOKEN   #REQUIRED
   rowsep   %yesorno; #IMPLIED
   spanname NMTOKEN   #REQUIRED
>

<!ELEMENT thead - O (colspec*, row+)>
<!ATTLIST thead
   valign (top|middle|bottom) "bottom"
>

<!ELEMENT tfoot - O (colspec*, row+)>
<!ATTLIST tfoot
   valign (top|middle|bottom) "top"
>

<!ELEMENT tbody - O (row+)>
<!ATTLIST tbody
   valign (top|middle|bottom) "top"
>

<!ELEMENT row - O (entry+)>
<!ATTLIST row
   rowsep %yesorno;           #IMPLIED
   valign (top|middle|bottom) "top"
>
```

```
<!ELEMENT entry - O (#PCDATA)>
<!ATTLIST entry
   align    (left|center|right|justify|char) #IMPLIED
   char     CDATA     #IMPLIED
   charoff  NUTOKEN   #IMPLIED
   colname  NMTOKEN   #IMPLIED
   colsep   %yesorno; #IMPLIED
   morerows NUMBER    #IMPLIED
   nameend  NMTOKEN   #IMPLIED
   namest   NMTOKEN   #IMPLIED
   rotate   %yesorno; #IMPLIED
   rowsep   %yesorno; #IMPLIED
   spanname NMTOKEN   #IMPLIED
   valign   (top|middle|bottom) #IMPLIED
>
```

Element structure

A CALS table has an optional title followed by one or more `tgroup` elements. This allows, for example, different portions of one table to have different numbers of columns. In practice, most CALS tables have a single `tgroup` element. The `tgroup` element is the major portion of the table. It has several optional parts: multiple `colspec` and `spanspec` elements followed by (at most) one heading and one footing element. The only required sub-element of a `tgroup` element is its body. Unlike the FrameMaker+SGML model of table structure, the CALS model has its `tgroup` element appearing after the footing element.

The `colspec` empty element has attributes describing characteristics of a table column. The `spanspec` empty element has attributes describing straddling characteristics of a portion of a table. These elements have no counterpart in FrameMaker+SGML. They exist only to have their attribute values specify information about other elements in the table.

The `thead` and `tfoot` heading and footing elements contain their own optional `colspec` elements followed by one or more rows.

The `tbody` element contains one or more rows.

As supported by FrameMaker+SGML, a table row consists of a set of cells in `entry` elements, each of which can contain only text. Readers familiar with the CALS model may notice that these declarations do not include the `entrytbl` element which supports creating tables within tables. FrameMaker+SGML does not allow tables within tables, so does not support this element.

Attribute structure

Elements in the CALS table model use attributes to describe properties of the table such as cell alignment or straddling behavior. For information on the meaning of the CALS attributes, see “Formatting properties for tables” on page 252.

Inheriting attribute values

Some attributes are common to several elements in the description of a table. In these cases, attribute values are inherited in the element hierarchy. The values of attributes associated with `colspec` and `spanspec` elements act as though they were on the parent element for inheritance purposes. That is, if a `tgroup` element has two `colspec` child elements and a `thead` child element, the attributes of the `colspec` elements apply to the `thead` element unless that element has its own `colspec` elements with attribute values that override the inherited ones.

Orient attribute

In the CALS model, the `table` element has an `orient` attribute. This attribute is not supported in FrameMaker+SGML, because there is no way in a FrameMaker+SGML table to specify orientation on the page.

Straddling attributes

A `spanspec` element describes a column range so that a straddle cell can describe which columns it spans by referencing a `spanspec` through its `spanname` attribute.

An `entry` element specifies which columns it occupies by one of three methods:

- Using the `namest` and `nameend` attributes to reference columns explicitly. The `namest` attribute indicates the first column in the straddle; the `nameend` attribute indicates the last column.
- Using the `spanname` attribute as an indirect reference to the columns.
- Using the `colname` attribute (for a non-straddled cell).

C SGML Read/Write Rules for CALS Table Model

By default, FrameMaker+SGML can read and write CALS tables without your intervention. For information on what it does by default and how you can change that behavior with read/write rules, see Chapter 14, “Translating Tables.” FrameMaker+SGML does not use read/write rules to implement its default interpretation of CALS tables. However, to help your understanding of the default interpretation, this appendix contains a set of rules that encapsulate the software’s default behavior for CALS tables.

As described in Chapter 14, “Translating Tables,” the software’s default behavior is different depending on whether the `table` element is a container element or a table element in FrameMaker+SGML. The only difference is what type of element `table` becomes and what happens to the `tgroup` element. All other elements and attributes always translate in the same way.

```
element "table" {
 /* If table is a container element, use this subrule: */
   is fm element;

 /* If table is a table element, use this subrule: */
   is fm table element;

 /* The rest of the subrules for table are always applicable. */
   attribute "tabstyle" is fm property table format;
   attribute "tocentry" is fm attribute;
   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 "orient" is fm attribute;
   attribute "pgwide" is fm property page wide;
}
```

```
element "tgroup"
{
   /* If table is a container element, use this subrule: */
   is fm table element;

   /* If table is a table element, use this subrule: */
   unwrap;

   /*The rest of the subrules for tgroup are always applicable.*/
   attribute "cols"        is fm property columns;
   attribute "tgroupstyle" is fm property table format;
   attribute "colsep"      is fm property column ruling;
   attribute "rowsep"      is fm property row ruling;
   attribute "align"       is fm attribute;
   attribute "charoff"     is fm attribute;
   attribute "char"        is fm attribute;
}

element "colspec"
{
   is fm colspec;
   attribute "colnum"   is fm property column number;
   attribute "colname"  is fm property column name;
   attribute "align"    is fm property cell alignment type;
   attribute "charoff"  is fm property cell alignment offset;
   attribute "char"     is fm property cell alignment character;
   attribute "colwidth" is fm property column width;
   attribute "colsep"   is fm property column ruling;
   attribute "rowsep"   is fm property row ruling;
}

element "spanspec"
{
   is fm spanspec;
   attribute "spanname" is fm property span name;
   attribute "namest"   is fm property start column name;
   attribute "nameend"  is fm property end column name;
   attribute "align"    is fm property cell alignment type;
   attribute "charoff"  is fm property cell alignment offset;
   attribute "char"     is fm property cell alignment character;
   attribute "colsep"   is fm property column ruling;
   attribute "rowsep"   is fm property row ruling;
}
```

```
element "thead"
{
   is fm table heading element;
   attribute "valign" is fm attribute;
}

element "tfoot"
{
   is fm table footing element;
   attribute "valign" is fm attribute;
}

element "tbody"
{
   is fm table body element;
   attribute "valign" is fm attribute;
}

element "row"
{
   is fm table row element;
   attribute "valign" is fm attribute;
   attribute "rowsep" is fm property row ruling;
}

element "entry"
{
   is fm table cell element;
   attribute "colname"  is fm property column name;
   attribute "namest"   is fm property start column name;
   attribute "nameend"  is fm property end column name;
   attribute "spanname" is fm property span name;
   attribute "morerows" is fm property more rows;
   attribute "colsep"   is fm property column ruling;
   attribute "rowsep"   is fm property row ruling;
   attribute "rotate"   is fm property rotate;
   attribute "valign"   is fm attribute;
   attribute "align"    is fm attribute;
   attribute "charoff"  is fm attribute;
   attribute "char"     is fm attribute;
}
```

C

D SGML Declaration

To be complete, an SGML document must start with an SGML declaration. This appendix contains the text of the SGML declaration used by FrameMaker+SGML when you do not supply one. It also describes the variants of the concrete syntax that you can use in your SGML declaration and unsupported optional SGML features.

When you import an SGML document, FrameMaker+SGML first searches for the declaration in the SGML document. If the software doesn't find the declaration there, it looks for an SGML declaration specified by your SGML application definition. If your definition does not specify an SGML declaration, then the software uses the declaration described below.

When you export a FrameMaker+SGML document to SGML, FrameMaker+SGML first tries to use an SGML declaration you specified by your application. If you haven't specified one, it uses the SGML declaration described below.

For information on how to specify an SGML declaration as part of an application, see "Application definition file" on page 42.

Text of the default SGML declaration

The SGML declaration provided by FrameMaker+SGML uses ISO Latin-1 as the character set, the reference concrete syntax, and the reference capacity set. The declaration enables the optional features `OMITTAG`, `SHORTTAG`, and `FORMAL`.

For information on the default translation between the FrameMaker+SGML and ISO Latin-1 character sets, see Appendix E, "Character Set Mapping." For information on using other ISO character sets, see Appendix F, "ISO Public Entities."

The text of the default SGML declaration is as follows:

```
<!SGML "ISO 8879:1986"

CHARSET

   BASESET "ISO Registration Number 100//CHARSET ECMA-94 Right
Part of Latin Alphabet Nr. 1//ESC 2/13 4/1"
```

```
   DESCSET
      0 9 UNUSED
      9 2 9
      11 2 UNUSED
      13 1 13
      14 18 UNUSED
      32 95 32
      127 1 UNUSED
      128 127 128
      255 1 UNUSED

CAPACITY
      PUBLIC "ISO 8879:1986//CAPACITY Reference//EN"

SCOPE DOCUMENT

SYNTAX

   SHUNCHAR         0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 20
            21 22 23 24 25 26 27 28 29 30 31 127 255

   BASESET "ISO Registration Number 100//CHARSET ECMA-94 Right
Part of Latin Alphabet Nr. 1//ESC 2/13 4/1"

    DESCSET 0 256 0

   FUNCTION RE 13
            RS 10
            SPACE 32
            TAB SEPCHAR 9

   NAMING         LCNMSTRT ""
            UCNMSTRT ""
            LCNMCHAR "&#45;&#46;"
            UCNMCHAR "&#45;&#46;"
            NAMECASE
               GENERAL YES
               ENTITY NO

   DELIM      GENERAL SGMLREF
            SHORTREF SGMLREF

   NAMES SGMLREF

   QUANTITY SGMLREF

FEATURES
```

```
    MINIMIZE DATATAG NO
             OMITTAG YES
             RANK NO
             SHORTTAG YES

    LINK      SIMPLE NO
          IMPLICIT NO
          EXPLICIT NO

    OTHER CONCUR NO
          SUBDOC NO
          FORMAL YES

APPINFO NONE

>
```

SGML concrete syntax variants

The SGML parser used by FrameMaker+SGML allows these modifications to the SGML reference concrete syntax:

- The `NAMECASE` parameter of the SGML declaration can be changed. The default settings below specify that general names are *not* case sensitive (`YES`), and entity names *are* case sensitive (`NO`):

  ```
  NAMECASEGENERALYES
        ENTITY   NO
  ```

- Reserved names can be changed.
- Short references can, but need not, be used. If they are used, the only possible short reference delimiter set is that of the reference concrete syntax.
- The value for the `NAMELEN` quantity can be increased up to 239.
- The values for the following quantities can be increased, but not to more than 30 times their value in the reference concrete syntax:

  ```
  ATTCNT
  ATTSPLEN
  BSEQLEN
  ENTLVL
  LITLEN
  PILEN
  TAGLEN
  TAGLVL
  ```

- The following quantities can be increased up to 253:

  GRPCNT
  GRPGTCNT
  GRPLVL

No SGML read/write rules are needed to provide for variant concrete syntaxes. FrameMaker+SGML obtains the information from the SGML declaration.

The concrete syntax declared in the SGML declaration must be used for the entire document; if a variant concrete syntax is declared, the reference concrete syntax cannot be used in the prolog. Thus, the concrete syntax scope parameter must be:

```
SCOPE DOCUMENT
```

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+SGML:

- DATATAG
- RANK
- LINK
- SUBDOC
- CONCUR

Your DTD and SGML documents cannot use any of these features. If they do, the FrameMaker+SGML signals an error and terminates processing. You cannot change this behavior by providing an SGML API client.

E Character Set Mapping

FrameMaker+SGML writes SGML documents using the ISO Latin-1 character set. This character set differs from FrameMaker+SGML's character set. Consequently, the software uses a default character set mapping to translate between the character sets.

FrameMaker+SGML includes copies of other ISO public entity sets and provides rules to handle them for your application. For information on how FrameMaker+SGML supports ISO public entities, see Appendix F, "ISO Public Entities."

This appendix describes the default mapping between the FrameMaker+SGML character set and the ISO Latin-1 character set. You can change this mapping by using the `character map` rule as described in "character map" on page 337.

To determine the mapping for a particular character, use the table on the next page as follows:

- For a character in the ISO Latin-1 character set, find the hexadecimal character code for the character of interest in the leftmost column. Read the mapping in the column labelled "Mapping from ISO Latin-1 to FrameMaker+SGML." The entry on the left side of the equal sign is the ISO Latin-1 character code. The entry on the right side of the equal sign is the character's translation in FrameMaker+SGML. For example, the character code `\xA1` has the entry:

  ```
  \xA1 = \xC1
  ```

  This means that the ISO Latin-1 character `\xA1` translates to the FrameMaker+SGML character `\xC1`.

- For a character in the FrameMaker+SGML character set, find the hexadecimal character code for the character of interest in the leftmost column. Read the mapping in the column labelled "Mapping from FrameMaker+SGML to ISO Latin-1." The entry on the right side of the equal sign is the FrameMaker+SGML character code. The entry on the left side of the equal sign is the character's translation in ISO Latin-1. For example, the character code `\x10` has the entry:

  ```
  \x20 = \x10
  ```

  This means that the FrameMaker+SGML character `\x10` translates to the ISO Latin-1 character `\x20`.

- If there is no row corresponding to a character code, then that character code is the same in both character sets.

Character code	Mapping from ISO Latin-1 to FrameMaker+SGML	Mapping from FrameMaker+SGML to ISO Latin-1
\x00	\x00 = trap	trap = \x00
\x01	\x01 = trap	trap = \x01
\x02	\x02 = trap	trap = \x02
\x03	\x03 = trap	trap = \x03
\x04	\x04 = trap	trap = \x04
\x05	\x05 = trap	trap = \x05
\x06	\x06 = trap	trap = \x06
\x07	\x07 = trap	trap = \x07
\x08	\x08 = trap	\x09 = \x08
\x09	\x09 = \x08	\x0A = \x09
\x0A	\x0A = \x0A	\x0A = \x0A
\x0B	\x0B = trap	trap = \x0B
\x0C	\x0C = trap	trap = \x0C
\x0D	\x0D = trap	trap = \x0D
\x0E	\x0E = trap	trap = \x0E
\x0F	\x0F = trap	trap = \x0F
\x10	\x10 = trap	\x20 = \x10
\x11	\x11 = trap	\x20 = \x11
\x12	\x12 = trap	\x20 = \x12
\x13	\x13 = trap	\x20 = \x13
\x14	\x14 = trap	\x20 = \x14
\x15	\x15 = trap	\x2D = \x15
\x16	\x16 = trap	trap = \x16
\x17	\x17 = trap	trap = \x17
\x18	\x18 = trap	trap = \x18
\x19	\x19 = trap	trap = \x19
\x1A	\x1A = trap	trap = \x1A
\x1B	\x1B = trap	trap = \x1B
\x1C	\x1C = trap	trap = \x1C
\x1D	\x1D = trap	trap = \x1D
\x1E	\x1E = trap	trap = \x1E

Character code	Mapping from ISO Latin-1 to FrameMaker+SGML	Mapping from FrameMaker+SGML to ISO Latin-1
\x1F	\x1F = trap	trap = \x1F
\x7F	\x7F = trap	trap = \x7F
\x80	\x80 = trap	\xC4 = \x80
\x81	\x81 = trap	\xC5 = \x81
\x82	\x82 = trap	\xC7 = \x82
\x83	\x83 = trap	\xC9 = \x83
\x84	\x84 = trap	\xD1 = \x84
\x85	\x85 = trap	\xD6 = \x85
\x86	\x86 = trap	\xDC = \x86
\x87	\x87 = trap	\xE1 = \x87
\x88	\x88 = trap	\xE0 = \x88
\x89	\x89 = trap	\xE2 = \x89
\x8A	\x8A = trap	\xE4 = \x8A
\x8B	\x8B = trap	\xE3 = \x8B
\x8C	\x8C = trap	\xE5 = \x8C
\x8D	\x8D = trap	\xE7 = \x8D
\x8E	\x8E = trap	\xE9 = \x8E
\x8F	\x8F = trap	\xE8 = \x8F
\x90	\x90 = trap	\xEA = \x90
\x91	\x91 = trap	\xEB = \x91
\x92	\x92 = trap	\xED = \x92
\x93	\x93 = trap	\xEC = \x93
\x94	\x94 = trap	\xEE = \x94
\x95	\x95 = trap	\xEF = \x95
\x96	\x96 = trap	\xF1 = \x96
\x97	\x97 = trap	\xF3 = \x97
\x98	\x98 = trap	\xF2 = \x98
\x99	\x99 = trap	\xF4 = \x99
\x9A	\x9A = trap	\xF6 = \x9A
\x9B	\x9B = trap	\xF5 = \x9B

Character code	Mapping from ISO Latin-1 to FrameMaker+SGML	Mapping from FrameMaker+SGML to ISO Latin-1
\x9C	\x9C = trap	\xFA = \x9C
\x9D	\x9D = trap	\xF9 = \x9D
\x9E	\x9E = trap	\xFB = \x9E
\x9F	\x9F = trap	\xFC = \x9F
\xA0	\xA0 = trap	trap = \xA0
\xA1	\xA1 = \xC1	trap = \xA1
\xA2	\xA2 = \xA2	\xA2 = \xA2
\xA3	\xA3 = \xA3	\xA3 = \xA3
\xA4	\xA4 = \xDB	\xA7 = \xA4
\xA5	\xA5 = \xB4	\xB7 = \xA5
\xA6	\xA6 = \x7C	\xB6 = \xA6
\xA7	\xA7 = \xA4	\xDF = \xA7
\xA8	\xA8 = \xAC	\xAE = \xA8
\xA9	\xA9 = \xA9	\xA9 = \xA9
\xAA	\xAA = \xBB	trap = \xAA
\xAB	\xAB = \xC7	\xB4 = \xAB
\xAC	\xAC = \xC2	\xA8 = \xAC
\xAD	\xAD = \x2D	trap = \xAD
\xAE	\xAE = \xA8	\xC6 = \xAE
\xAF	\xAF = \xF8	\xD8 = \xAF
\xB0	\xB0 = \xFB	trap = \xB0
\xB1	\xB1 = trap	trap = \xB1
\xB2	\xB2 = trap	trap = \xB2
\xB3	\xB3 = trap	trap = \xB3
\xB4	\xB4 = \xAB	\xA5 = \xB4
\xB5	\xB5 = trap	trap = \xB5
\xB6	\xB6 = \xA6	trap = \xB6
\xB7	\xB7 = \xA5	trap = \xB7
\xB8	\xB8 = \xFC	trap = \xB8
\xB9	\xB9 = trap	trap = \xB9
\xBA	\xBA = \xBC	trap = \xBA

Character code	Mapping from ISO Latin-1 to FrameMaker+SGML	Mapping from FrameMaker+SGML to ISO Latin-1
\xBB	\xBB = \xC8	\xAA = \xBB
\xBC	\xBC = trap	\xBA = \xBC
\xBD	\xBD = trap	trap = \xBD
\xBE	\xBE = trap	\xE6 = \xBE
\xBF	\xBF = \xC0	\xF8 = \xBF
\xC0	\xC0 = \xCB	\xBF = \xC0
\xC1	\xC1 = \xE7	\xA1 = \xC1
\xC2	\xC2 = \xE5	\xAC = \xC2
\xC3	\xC3 = \xCC	trap = \xC3
\xC4	\xC4 = \x80	trap = \xC4
\xC5	\xC5 = \x81	trap = \xC5
\xC6	\xC6 = \xAE	trap = \xC6
\xC7	\xC7 = \x82	\xAB = \xC7
\xC8	\xC8 = \xE9	\xBB = \xC8
\xC9	\xC9 = \x83	trap = \xC9
\xCA	\xCA = \xE6	trap = \xCA
\xCB	\xCB = \xE8	\xC0 = \xCB
\xCC	\xCC = \xED	\xC3 = \xCC
\xCD	\xCD = \xEA	\xD5 = \xCD
\xCE	\xCE = \xEB	trap = \xCE
\xCF	\xCF = \xEC	trap = \xCF
\xD0	\xD0 = trap	\x2D = \xD0
\xD1	\xD1 = \x84	\x2D = \xD1
\xD2	\xD2 = \xF1	\x22 = \xD2
\xD3	\xD3 = \xEE	\x22 = \xD3
\xD4	\xD4 = \xEF	\x60 = \xD4
\xD5	\xD5 = \xCD	\x27 = \xD5
\xD6	\xD6 = \x85	trap = \xD6
\xD7	\xD7 = trap	trap = \xD7
\xD8	\xD8 = \xAF	\xFF = \xD8
\xD9	\xD9 = \xF4	trap = \xD9

Character code	Mapping from ISO Latin-1 to FrameMaker+SGML	Mapping from FrameMaker+SGML to ISO Latin-1
\xDA	\xDA = \xF2	\x2F = \xDA
\xDB	\xDB = \xF3	\xA4 = \xDB
\xDC	\xDC = \x86	trap = \xDC
\xDD	\xDD = trap	trap = \xDD
\xDE	\xDE = trap	trap = \xDE
\xDF	\xDF = \xA7	trap = \xDF
\xE0	\xE0 = \x88	trap = \xE0
\xE1	\xE1 = \x87	\xB7 = \xE1
\xE2	\xE2 = \x89	\x2C = \xE2
\xE3	\xE3 = \x8B	trap = \xE3
\xE4	\xE4 = \x8A	trap = \xE4
\xE5	\xE5 = \x8C	\xC2 = \xE5
\xE6	\xE6 = \xBE	\xCA = \xE6
\xE7	\xE7 = \x8D	\xC1 = \xE7
\xE8	\xE8 = \x8F	\xCB = \xE8
\xE9	\xE9 = \x8E	\xC8 = \xE9
\xEA	\xEA = \x90	\xCD = \xEA
\xEB	\xEB = \x91	\xCE = \xEB
\xEC	\xEC = \x93	\xCF = \xEC
\xED	\xED = \x92	\xCC = \xED
\xEE	\xEE = \x94	\xD3 = \xEE
\xEF	\xEF = \x95	\xD4 = \xEF
\xF0	\xF0 = trap	trap = \xF0
\xF1	\xF1 = \x96	\xD2 = \xF1
\xF2	\xF2 = \x98	\xDA = \xF2
\xF3	\xF3 = \x97	\xDB = \xF3
\xF4	\xF4 = \x99	\xD9 = \xF4
\xF5	\xF5 = \x9B	trap = \xF5
\xF6	\xF6 = \x9A	\x5E = \xF6
\xF7	\xF7 = trap	\x7E = \xF7
\xF8	\xF8 = \xBF	\xAF = \xF8

Character code	Mapping from ISO Latin-1 to FrameMaker+SGML	Mapping from FrameMaker+SGML to ISO Latin-1
\xF9	\xF9 = \x9D	trap = \xF9
\xFA	\xFA = \x9C	trap = \xFA
\xFB	\xFB = \x9E	\xB0 = \xFB
\xFC	\xFC = \x9F	\xB8 = \xFC
\xFD	\xFD = trap	trap = \xFD
\xFE	\xFE = trap	trap = \xFE
\xFF	\xFF = \xD8	trap = \xFF

E

F ISO Public Entities

Annex D of the SGML standard defines several sets of internal SDATA entities. Each entity represents a character; each entity set is a logical grouping of these entities. DTDs frequently include these entity sets by using parameter entity references to external entities accessed with a public identifier. People in the SGML community frequently interchange DTDs and SGML documents with such entity references and assume that the recipient can interpret the public identifiers. FrameMaker+SGML includes copies of these entity sets and makes them available using the default handling of public identifiers.

Because these entity sets are defined in an ISO standard and are accessed with public identifiers, they are commonly known as *ISO public entity sets*. The public entity sets fall into the following categories:

Entity set	Description
Latin alphabetic characters	Latin alphabetic characters used in Western European languages
Greek alphabetic characters	Letters of the Greek alphabet
Greek symbols	Greek character names for use as variable names in technical applications
Cyrillic alphabetic characters	Cyrillic characters used in the Russian language
Numeric and special graphic characters	Minimum data characters and reference concrete syntax characters
Diacritical mark characters	Diacritical marks
Publishing characters	Well-known publishing characters
Technical symbols	Technical symbols
Added math symbols	Mathematical symbols

If your application uses FrameMaker+SGML's support of ISO entity sets, you may want to create palettes your end user can use to enter these entities in a FrameMaker+SGML document. For information on creating these palettes, see "Facilitating entry of special characters that translate as SGML entities" on page 244.

What you need to use ISO public entities

For your end users to use characters from the ISO public entity sets, your application needs two pieces of information for each character entity: the entity's declaration, and an SGML read/write rule that tells FrameMaker+SGML how to translate a reference to that entity in an SGML document to a character or variable in a FrameMaker+SGML document. FrameMaker+SGML provides this information in two files for each entity set.

All files used for ISO public entity sets are in the directory `$SGMLDIR/isoents`. For information on the location of this directory on your system, see "Location of SGML files" on page 41. The files for each entity set are as follows:

Entity set	Entity declaration files	Read/write rules files
Latin alphabetic characters	`isolat1.ent` `isolat2.ent`	`isolat1.rw` `isolat2.rw`
Greek alphabetic characters	`isogrk1.ent` `isogrk2.ent`	`isogrk1.rw` `isogrk2.rw`
Greek symbols	`isogrk3.ent` `isogrk4.ent`	`isogrk3.rw` `isogrk4.rw`
Cyrillic alphabetic characters	`isocyr1.ent` `isocyr2.ent`	`isocyr1.rw` `isocyr2.rw`
Numeric and special graphic characters	`isonum.ent`	`isonum.rw`
Diacritical mark characters	`isodia.ent`	`isodia.rw`
Publishing characters	`isopub.ent`	`isopub.rw`
Technical symbols	`isobox.ent` `isotech.ent`	`isobox.rw` `isotech.rw`

Added math symbols	`isoamso.ent`
	`isoamsb.ent`
	`isoamsr.ent`
	`isoamsn.ent`
	`isoamsa.ent`
	`isoamsc.ent`
		`isoamso.rw`
		`isoamsb.rw`
		`isoamsr.rw`
		`isoamsn.rw`
		`isoamsa.rw`
		`isoamsc.rw`

Entity declaration files

Each entity declaration file starts with two comment declarations that suggest both the public identifier and the entity name by which to identify the entity set. For the ISO Latin-1 entity set, these comments are:

```
<!-- (C) International Organization for Standardization 1986
      Permission to copy in any form is granted for use with
      conforming SGML systems and applications as defined in
      ISO 8879, provided this notice is included in all copies.
-->
<!-- Character entity set. Typical invocation:
      <!ENTITY % ISOlat1 PUBLIC
         "ISO 8879-1986//ENTITIES Added Latin 1//EN">
      %ISOlat1;
-->
```

After the initial comments, an entity declaration file consists of a sequence of entity declarations. For example, the first few entity declarations for ISO Latin-1 are as follows:

```
<!ENTITY aacute SDATA "[aacute]"--=small a, acute accent-->
<!ENTITY Aacute SDATA "[Aacute]"--=capital A, acute accent-->
<!ENTITY acirc SDATA "[acirc ]"--=small a, circumflex accent-->
<!ENTITY Acirc SDATA "[Acirc ]"--=capital A, circumflex accent-->
<!ENTITY agrave SDATA "[agrave]"--=small a, grave accent-->
<!ENTITY Agrave SDATA "[Agrave]"--=capital A, grave accent-->
<!ENTITY aring SDATA "[aring ]"--=small a, ring-->
<!ENTITY Aring SDATA "[Aring ]"--=capital A, ring-->
```

You should never modify these files, because they provide the standard ISO public entity declarations.

If your SGML documents use the standard invocations for ISO public entity sets, you do not have to provide any information in your application definition on where to find these entities; FrameMaker+SGML finds them in the default directory. If necessary, you can provide explicit `public` statements to substitute alternative versions of the entity sets. For information on working with application definitions, see “Application definition file” on page 42.

Entity read/write rules files

FrameMaker+SGML provides read/write rules for many of the entities in the ISO public entity sets. The rules are organized in files for each public entity set. These files are not complete rules documents. Instead, they are simply lists of rules or comments explaining which entities do not have default correspondences.

You can include individual files in your application's read/write rules document by using the `#include` statement. To include the rules for all of the ISO public entity sets, use this single statement:

```
#include isoall.rw
```

To include only the ISO Latin-1 entity set, use these statements:

```
#include isolat1.rw
#include isolat2.rw
```

For more information on read/write rules files, see Chapter 11, "SGML Read/Write Rules and Their Syntax."

Format of entity rules

By default, FrameMaker+SGML has rules for each entity that can be represented in FrameMaker+SGML using the standard FrameMaker+SGML character set, the Symbol font, or the Zapf Dingbat font and for a few (such as the fractions in `isonum`) that can be represented with a FrameMaker+SGML user variable. Entities that cannot be represented in this way do not have a default translation. Users of your application may have more fonts available. If so, you can modify these rules files to include translations for other entities.

The default rules for entities available in the default character sets or through variables differ depending on how FrameMaker+SGML translates the entity.

- If the character appears in FrameMaker+SGML's standard character set and requires no special formatting, the rule has the following form:

  ```
  entity "ename" is fm char code;
  ```

  where *ename* is the entity name and *code* is the character code. For example, the following rule is for the small letter "a" with an acute accent:

  ```
  entity "aacute" is fm char 0x87;
  ```

- If the character appears in FrameMaker+SGML's Symbol or Zapf Dingbat character set or appears in FrameMaker+SGML's standard character set, but requires special formatting, the rule has the following form:

  ```
  entity "ename" is fm char code in "fmtag";
  ```

  where *ename* is the entity name, *code* is the character code, and *fmtag* is one of the character tags defined below. For example, the following rule is for the plus-or-minus sign:

  ```
  entity "plusnm" is fm char 0xb1 in "FmSymbol";
  ```

- If the character can be represented by an FrameMaker+SGML variable, the rule has the following form:

  ```
  entity "ename" is fm variable "var";
  ```

where *ename* is the entity name and *var* is one of the FrameMaker+SGML variables defined below. For example, the following rule is for the fraction one-half:

```
entity "frac12" is fm variable "FmFrac12";
```

For details on how each entity translates into a FrameMaker+SGML document, see the individual rules files.

Character formats

As mentioned above, the rules for some character entities refer to FrameMaker+SGML character formats or variable names. FrameMaker+SGML has default definitions for these character formats:

Character format	Defined as
FmDenominator	Default font, subscripted; other characteristics As Is
FmDingbats	Zapf Dingbat font; other characteristics As Is
FmNumerator	Default font, superscripted; other characteristics As Is
FmSdata	Default font, underlined and in green; other characteristics As Is
FmSuperscript	Default font superscripted; other characteristics As Is
FmSymbol	Symbol font; other characteristics As Is
FmUnderlineSymbol	Symbol font, underlined; other characteristics As Is

Variables

FrameMaker+SGML also has default definitions for these variables:

Variable	Defined as
FmCare-of	$^{c}/_{o}$
FmEmsp13	an em space
FmFrac12	$^{1}/_{2}$
FmFrac13	$^{1}/_{3}$
FmFrac14	$^{1}/_{4}$
FmFrac15	$^{1}/_{5}$
FmFrac16	$^{1}/_{6}$
FmFrac18	$^{1}/_{8}$
FmFrac23	$^{2}/_{3}$
FmFrac25	$^{2}/_{5}$
FmFrac34	$^{3}/_{4}$
FmFrac35	$^{3}/_{5}$
FmFrac38	$^{3}/_{8}$
FmFrac45	$^{4}/_{5}$

Variable	Defined as
FmFrac56	$^{5}/_{6}$
FmFrac58	$^{5}/_{8}$
FmFrac78	$^{7}/_{8}$

Your end user's documents may not have these character formats or variables defined. When FrameMaker+SGML imports an SGML document with an entity reference that needs one of these formats or variables, it checks whether the template defined in the SGML application provides the definition. If so, it uses the information from the template. If not, it uses its own definitions, copying the definition to the appropriate catalog of the document being processed and using it to process the entity.

What happens with the declarations and rules

Your application may use some or all of the entity declarations and read/write rules provided with FrameMaker+SGML. In addition, you may choose to have different declarations or rules for some or all of the entities.

If you want to use the translations provided by FrameMaker+SGML with no changes, you do so in one of two ways.

- If your application has no other read/write rules, then you do not have to explicitly mention the rules for these entity sets. That is, if the definition of your application does not include a read/write rules file, FrameMaker+SGML behaves as though it had a rules file that included only the ISO public entity rules.
- On the other hand, if your application does have a read/write rules file, then that file must explicitly include the rules for the ISO public entity sets in which you're interested. If you want all of them, add the following line to your file:

  ```
  #include isoall.rw
  ```

  When you create a new read/write rules file, this line is automatically included.

If you want to use only the rules that FrameMaker+SGML provides, be sure that your rules file has no additional `entity` rules referring to these entities. However, you may want to have FrameMaker+SGML translate most but not all of these entities in the way it provides, while you change the behavior for some of them with rules or entity declarations. To do this, include an extra entity declaration or rule for the appropriate entities.

For example, assume the DTD for your application is called `myapp.dtd` and includes the following lines:

```
<!ENTITY % ISOlat1 PUBLIC
         "ISO 8879-1986//ENTITIES Added Latin 1//EN">
%ISOlat1;
```

Further, assume the application has no rules or has a rules document that contains the following lines:

```
#include "isolat1.rw"
#include "isolat2.rw"
```

The default version of `isolat1.rw` contains the rule:

```
entity "aacute" is fm char 0x87;
```

This translates references to the `aacute` entity as the small letter a with an acute accent. Suppose, however, that your application needs this entity, instead, to translate as a particular bitmap that you store as a reference element in the FrameMaker+SGML document template. You can accomplish this by adding either a new entity declaration or a new rule.

To continue the example, assume you import an SGML document that begins as follows:

```
<!DOCTYPE myapp SYSTEM "myapp.dtd" [
   <!ENTITY aacute SDATA "fm ref: acute-a">
]>
```

This SGML document has two declarations for `aacute`. The parser uses the first one it encounters. Since the parser processes the external DTD subset after it processes the internal DTD subset, it finds the declaration that uses the reference element first and this is the entity declaration FrameMaker+SGML uses. Since FrameMaker+SGML recognizes the `fm ref` in the parameter literal, it uses that parameter literal to process the entity reference and ignores any rules that refer to the entity. The resulting document includes the reference element for the entity reference.

Instead of including the declaration for `aacute` that uses the `fm ref` parameter literal, you can add the following rule to your rules file:

```
entity "aacute" is fm reference element "acute-a";
```

Remember that FrameMaker+SGML uses the first rule in a rules file that applies to a particular situation. Therefore, if you use this rule, then the line in the example that includes `isolat1.rw` must occur after this rule. That is, your rules file must look like:

```
entity "aacute" is fm reference element "acute-a";
. . .
#include isolat1.rw
. . .
```

If, instead, it looks like:

```
#include isolat1.rw
. . .
entity "aacute" is fm reference element "acute-a";
. . .
```

FrameMaker+SGML finds the rule in `isolat1.rw` before your rule and use that to process references to the `aacute` entity.

FrameMaker+SGML has rules for entities in the ISO public entity sets for which there is a direct correspondence in one of its standard character sets or which can be created using a character from those character sets. It does not provide rules for entities that would require a different character set or a graphic.

If you reference an ISO public entity for which there is not a rule, the software treats it as it does any other entity that doesn't have a corresponding rule. You can change this behavior with the `entity` rule. For more information on FrameMaker+SGML's translation of entities in the absence of rules and for information on how you can modify this, see Chapter 13, "Translating Entities and Processing Instructions."

G SGML Batch Utilities for UNIX

The UNIX version of FrameMaker+SGML provides two utility programs, `fmimportsgml` and `fmexportsgml`, in addition to the utility programs defined for FrameMaker+SGML and described in the FrameMaker+SGML user's manual. You use `fmimportsgml` for batch importing of SGML files and `fmexportsgml` for batch exporting of FrameMaker+SGML files to SGML.

Some points to keep in mind about these programs:

- The utility programs start FrameMaker+SGML. For this reason, they can run only on a system capable of running FrameMaker+SGML.
- The programs overwrite existing files. If your user preferences specify creation of backup files, the programs create these files.
- The programs recognize wildcards in filenames. It is an error if a wildcard refers to multiple files where a single file is expected.
- Labeled parameters are those of the form:

  `-parameter_name` *`parameter_value`*

  All labeled parameters are optional. They can be entered in any order preceding the names of the documents to be converted.

Importing SGML documents in batch mode

The `fmimportsgml` program imports SGML documents as FrameMaker+SGML documents. You call it as follows:

```
fmimportsgml
   [-dir dir]
   [-suffix suffix]
   [-flow flow]
   [-sgmlapps fname]
   [-app app_name | -noapp]
   [-log log_file]
   [-i]
   [-v]
   [-h]
sgml_documents
```

where:

`-dir` *`dir`*	names the directory where `fmimportsgml` stores output files. If this parameter is not specified, `fmimportsgml` uses the current directory.
`-suffix` *`suffix`*	specifies the suffix for FrameMaker+SGML documents created by `fmimportsgml`. This option does not apply when you are importing a book; in this case, the suffix is always `.book`. If this parameter is not specified, the suffix is `.doc` for a single document.
`-flow` *`flow`*	specifies a flow tag for the output FrameMaker+SGML document. If this parameter is present, the application definition must specify a FrameMaker+SGML template. `fmimportsgml` writes its output to the first flow with the indicated tag that occurs on a body page of the template. FrameMaker+SGML appends the translation of the SGML document to the end of the content (if any) of the specified flow. It is an error if the flow doesn't exist. If you want to use `fmimportsgml` to import SGML documents into multiple flows of the same FrameMaker+SGML file, you invoke `fmimportsgml` multiple times, specifying a different flow tag each time. In the subsequent calls to `fmimportsgml`, you specify the output file of the previous call to `fmimportsgml` as the input template for the next call.
`-sgmlapps` *`fname`*	specifies an SGML file to use in place of `$SGMLDIR/sgmlapps.doc`. If this parameter is not specified, `fmimportsgml` uses the standard search path to find `sgmlapps.doc`—for details, see "Location of SGML files" on page 41.
`-app` *`app_name`* `-noapp`	specifies the application definition to use or specifies using no application at all. If this parameter is not specified, `fmimportsgml` uses no application.
`-log` *`log_file`*	specifies the name of a FrameMaker+SGML document to use as the log file. If this parameter is not present and if there are errors during conversion, `fmimportsgml` creates the log file `sgmllog.doc` in the current directory.
`-i`	causes `fmimportsgml` to use the `imakersgml` binary.
`-v`	displays the software version. When `-v` is specified, all other parameters are ignored.

`-h`	displays a brief description of `fmimportsgml` and its parameters. When `-h` is specified, the software ignores all other parameters. Omitting all parameters is equivalent to specifying `-h`.
`sgml_documents`	names one or more SGML documents to be imported.

For example, assume you the `/sgmlbook` directory contains the files `a.sgml`, `b.sgml`, and `c.txt` and you use this command:

```
fmimportsgml -app MyApp -dir /fmbook -suffix doc /mydocs/*.sgml
```

The software uses the `MyApp` application defined in `sgmlapps.doc` to create the files `a.doc` and `b.doc` in the `/fmbook` directory.

Exporting documents as SGML in batch mode

The `fmexportsgml` program exports FrameMaker+SGML documents as SGML documents. You call it as follows:

```
fmexportsgml
   [-dir dir]
   [-suffix suffix]
   [-flow flow]
   [-sgmlapps fname]
   [-app app_name | -noapp]
   [-log log_file]
   [-i]
   [-v]
   [-h]
fm_documents
```

where:

`-dir` *`dir`*	names the directory where `fmexportsgml` stores output files. If this parameter is not specified, `fmexportsgml` uses the current directory.
`-suffix` *`suffix`*	specifies the suffix for SGML documents created by `fmexportsgml`. If this parameter is not specified, the suffix is `.sgm`.
`-flow` *`flow`*	specifies a flow tag for the FrameMaker+SGML input document. `fmexportsgml` exports the first flow with the indicated tag that occurs in the FrameMaker+SGML input document; it is an error if there is none. If this parameter is not specified, FrameMaker+SGML exports the main flow of the document.

`-sgmlapps` *`fname`*	specifies an SGML file to use in place of `$SGMLDIR/sgmlapps.doc`. If this parameter is not specified, `fmexportsgml` uses the standard search path to find `sgmlapps.doc`—for details, see "Location of SGML files" on page 41.
`-app` *`app_name`* `-noapp`	specifies the application definition to use or specifies using no application at all. If this parameter is not specified, `fmexportsgml` uses no application.
`-log` *`log_file`*	specifies the name of a FrameMaker+SGML document to use as the log file. If this parameter is not present and if there are errors during conversion, `fmexportsgml` creates the log file `sgmllog.doc` in the current directory.
`-i`	causes `fmexportsgml` to use the `imakersgml` binary.
`-v`	displays the software version. When `-v` is specified, all other parameters are ignored.
`-h`	displays a brief description of `fmexportsgml` and its parameters. When `-h` is specified, the software ignores all other parameters. Omitting all parameters is equivalent to specifying `-h`.
`fm_documents`	names one or more FrameMaker+SGML documents to be exported.

For example, assume you the `/fmbook` directory contains the files `a.doc`, `b.doc`, and `c.text` and you use this command:

```
fmexportsgml -noapp dir /sgmlbook -suffix sgml /mydocs/*.doc
```

The software does not use an SGML application. It creates the files `a.sgml` and `b.sgml` in the `/sgmlbook` directory.

H Using the XML Export Feature

FrameMaker and FrameMaker+SGML include a feature to export both structured and unstructured files to XML. Extensible Markup Language (XML) is intended for data exchange and publishing on the World Wide Web, and shares characteristics of both HTML and SGML. Like HTML and SGML, XML uses elements and structure. XML is different from SGML in that the Document Type Declaration (DTD), the definition of elements and the hierarchy that they may form in the document, is optional. XML differs from HTML in that it is extensible; the user may define his own set of elements and is not restricted to a pre-defined, “hard coded” set of tags. Therefore, XML provides users with capabilities more powerful than HTML but less complex than SGML.

The XML export feature is a new option under the File>Save As... menu in both FrameMaker and FrameMaker+SGML. The feature works differently for unstructured files than in structured files.

For exporting XML from unstructured documents (either a FrameMaker file, or from a FrameMaker+SGML file that does not use structure), the process is the same as that used for export to HTML: the mapping used by the export to specify what element to create for each paragraph tag in the source FrameMaker file is defined on the reference page, and the export function creates not only an XML but also a Cascading Style Sheet (CSS) that can be used with the document. Use the HTML conversion documentation found in Chapter 19 of the FrameMaker User Guide or the FrameMaker online help.

For exporting XML from structured documents (from a FrameMaker+SGML file that uses structure), the process is the same as that used for export to SGML: a mapping from elements in the source FrameMaker+SGML file to elements in the output XML file may be specified in a Write Rules file. If this mapping is not specified then the export function will use a default one-to-one mapping. See Chapter 10, “Introduction to Translating between SGML and FrameMaker+SGML,” and Chapter 21, “Read/Write Rules Reference,” of this manual for instructions on setting up the mappings. This appendix covers issues specific to XML export.

The following topics are discussed in this appendix:

Topic	Discussed in
New terminology	, “Terms used in this appendix.”
Restrictions of the XML export feature	, “Restrictions.”
XML syntax	, “XML syntax.”

Topic	Discussed in
How empty elements are written	, “Empty elements.”
Generating a style sheet	, “CSS Style sheets.”
XML character encoding	, “XML character encoding.”
Read/write rules	, “XML read/write rules.”
Handling invalid element names	, “Handling invalid element names.”
Default attribute value	, “Default attribute value.”

Terms used in this appendix

Attribute	Additional information about an element. Attribute values must conform to the rules specified for that attribute in the DTD.
CSS	Cascading Style Sheet. A style sheet used by HTML and XML to specify the presentation or format of the document in a viewer or in print. Enables you to specify fonts, colors, whitespace, positioning, backgrounds, and so forth. Several documents can share the same stye sheet, allowing consistent presentation over a collection of documents. A CSS style sheet is written out when exporting both structured and unstructured flows to XML.
Entity	A named piece of data; somewhat like a variable.
Valid	Document markup that conforms to the structure defined by the DTD.
Well-formed	Document markup that follows simple rules such as all begin tags must be matched with end tags; the document has one and only one root element; and elements in the root element can follow one another, or appear inside another element, but may never overlap.
XLink	The Extensible Linking specification, the link syntax to be used with XML files. XLink can link between multiple resources and links between read-only resources.
XSL	The Extensible Style Language specification, the style sheet language used with XML. A different language than CSS, though similar in some respects.

Restrictions

XML support is limited to export only; the import of existing XML documents is not supported.

A DTD is not written out upon XML export. Even if a DTD is specified, the instance will not be validated against the DTD upon XML export; that is, the exported XML file will be well-formed only, and while probably valid will not be validated.

Document splitting is supported for unstructured flows, similar to the HTML export feature, but not for structured flows.

Hypertext links in a structured document do not convert to links during export to XML.

XML syntax

There are a number of syntax issues to consider when exporting to XML.

Empty elements

Empty elements are written out using the following syntax:

```
<SomeEmptyElement/>
```

This applies to export from both structured and unstructured flows. For export from unstructured flows, anchored frames and their contents, i.e., graphics and equations, are mapped to empty elements.

Typically in structured flows, cross-references and variables are empty elements. In XML export, FrameMaker+SGML treats these two objects as elements with content. For example, a cross-reference element `xref` that contains the text “See the User’s Guide for more info”, will be written out as:

```
<xref xml:link = "locator" href = "/usr/info.fm" show = "user"
actuate = "auto">"See the User’s Guide for more info"</xref>
```

If a variable element named `var` is used, and it has content “June 5th, 1973", the software will write out:

```
<var>"June 5th, 1973"</var>
```

CSS Style sheets

For generating a CSS style sheet for XML export of structured flows, FrameMaker+SGML looks for formatting rules which apply only to `All Contexts`. It does not handle context formatting and level formatting rules. Each time you export to XML, a CSS style sheet is generated and linked to the XML instance. The filename of the CSS generated style sheet will be `<XML_filename>.css`.

For unstructured flows, FrameMaker looks at the paragraph and character formats and generates CSS. Note that for both the structured and unstructured cases, there will be more work for the user to do.

Below is a mapping of FrameMaker+SGML character and paragraph properties to their CSS equivalent.

Mapping from FrameMaker Character Properties to CSS Properties

FrameMaker Character Property	CSS Property	Comments
Capitialization	font-style & text-transform	`FV_CAPITAL_CASE_SMALL` gets mapped into the small-caps value for the font-style property. `FV_CASE_LOWER` maps to the lowercase value for the text-transform property. `FV_CASE_UPPER` maps to the uppercase value for the text-transform property.
ChangeBar	None	
Color	color	The CMYK property is converted algorithmically to RGB.
FontAngle	font-style	Italic and Oblique convert without modification. Otherwise ignored.
FontFamily	font-family	Converted without modification. No attempt is made to determine fallback fonts or to support web fonts.
FontPlatformName	Ignored	
FontPostScript Name	Ignored	
FontSize	font-size	The FrameMaker unit of measure (MetricT) is converted to points and pt is appended.
FontVariation	Ignored	
FontWeight	font-weight	The properties extra-light, light, demi-light, medium, demi-bold, bold, extra-bold are converted without modification. Other font weights are mapped to values in this list.
KernX	Ignored	
KernY	Ignored	

Mapping from FrameMaker Character Properties to CSS Properties

FrameMaker Character Property	CSS Property	Comments
Outline	Ignored	
Overline	text-decoration	If true, sets text-decoration to overline.
PairKern	Ignored	
Position	vertical-align	`Plain` maps to baseline, Subscript maps to sub, and Superscript maps to super
Shadow	Ignored	
Spread	Ignored	
Strikethrough	text-decoration	If true, sets text-decoration to line-through.
Underlining	text-decoration	Underline, Double Underline, and Numeric Underline all map to the underline value of the text-decoration property.
Use*		If any of the use properties are false, the corresponding CSS properties are omitted.

Mapping from FrameMaker Paragraph Properties to CSS Properties

FrameMaker Paragraph Format Properties	CSS Property	Comments
AcrobatLevel	Ignored	
AutoNumChar	Ignored	
AutoNumString	Ignored	
PgfIsAutoNum	Ignored	
AdjHypens	Ignored	
Hyphenate	Ignored	
HyphMinPrefix	Ignored	

Mapping from FrameMaker Paragraph Properties to CSS Properties

FrameMaker Paragraph Format Properties	CSS Property	Comments
HyphMinSuffix	Ignored	
HyphMinWord	Ignored	
Language	LANG attribute	
FirstIndent	text-indent	
LeftIndent	margin-left	
RightIndent	margin-right	
Leading	Ignored	
LineSpacing	Ignored	
BlockLines	Ignored	
KeepWithNext	Ignored	
KeepWithPrev	Ignored	
PgfAlignment	text-align	
Placement	Ignored	
RunInSeparator	Ignored	
SpaceAbove	margin-top	
SpaceBelow	margin-bottom	
Start	Ignored	
BottomSeparator	border-bottom	The specified frame is converted into a graphic and the URL is supplied as the value of to border-bottom.
TopSeparator	border-bottom	The specified frame is converted into a graphic and the URL is supplied as the value of to border-bottom.
Cell*	Ignored	

Mapping from FrameMaker Paragraph Properties to CSS Properties

FrameMaker Paragraph Format Properties	CSS Property	Comments
NumTabs	Ignored	
Tabs	Ignored	
Name	CLASS attribute	
LetterSpace	Ignored	
MaxSpace	Ignored	
MinSpace	Ignored	
OptSpace	Ignored	

The base paragraph format is “body”. If there is paragraph format that uses all the same properties of body but has different font weight, only the font weight will be explicitly specified. For example, if there is a paragraph format called FOO which has two differing properties of body (PgfAlignment is centered and SpaceAbove is 12pt.), the following would be written out in CSS.

```
FOO { text-align: center;
      margin-top: 12pt; }
```

For character properties, “Default Font” is the baseline. Thus, if we have a character format called BAR which uses FontSize of 16pt, we would write out in CSS:

```
BAR { font-size: 16pt;}
```

XML character encoding

An element in the SGML application file called `XmlCharacterEncoding` specifies the character encoding for XML export. If none is specified, then UTF-8 (the default character encoding for XML export) is used. This value is placed in the default section of the `sgmlapps` file.

The new `XmlCharacterEncoding` element is nearly identical to the SGML character encoding element, but has the child element `UTF-8` while the SGML character encoding element does not. Character encoding UTF-8 is used only for XML and cannot be used for exporting SGML.

This element is not processed during SGML export. If the `XmlCharacterEncoding` element is present, and the SGML character element is not, it will be treated as if there was

no character encoding specified at all. Likewise, for XML export, the SGML character encoding element is completely ignored.

XML read/write rules

An element in the SGML application file called `XmlWriteRules` specifies the rules file to use for XML export. If no `XmlWriteRules` element is specified, export will default to the SGML behavior of using the default rules file.

For example, if there is a `Read/WriteRules` element that points to `sgml.rw` and there is an `XmlWriteRules` element that points to `xml.rw`, the `XmlWriteRules` element will override the SGML `Read/WriteRules` element for XML export. But, if there is only the SGML `Read/WriteRules` element, when exporting to XML, `sgml.rw` is used. If neither element existed, the software defaults to whatever is in the “default” application.

The following table shows read/write rules that are interpreted differently for XML export than SGML export.

`Convert referenced graphics`	Changed	The `href` attribute in XLink points to the XML file.
`entity name is`	Ignored	This rule is generally ignored. However, with graphics, if you originally specify that a certain graphic be written out to entity "foo", the software will save out the graphic using "foo".
`equation entity name is`	Ignored	This rule is ignored for XML export.
`external DTD`	Ignored	This rule is ignored for XML export.
`fm variable`	Ignored	This rule is ignored for XML export.
`include DTD`		This rule is ignored for XML export. The software always uses the rule `write sgml document instance only`. The DTD is not exported with XML export.
`include sgml declaration`	Ignored	This rule is ignored for XML export. The software uses its own internal SGML declaration.
`is fm char`	Ignored	This rule is ignored for XML export.
`is fm property`	Exception	There is no change to this rule except for when it involves graphics. For graphics, the software only writes out certain attributes, so when dealing with graphics this rule is ignored.

`is fm property value`	Exception	This rule is ignored for XML export if it is dealing with graphics.
`is fm reference element`	Ignored	This rule is ignored for XML export.
`is fm variable`	Ignored	This rule is ignored for XML export.
`preserve line breaks`	Change	Because spaces cannot be written out, the software writes out an extra attribute `xml:space=preserve` for elements which apply this r/w rule. This is also reflected in the CSS style sheet for elements in which this property applies.
`specify size in`	Ignored	This rule is ignored for XML export.
`write sgml document`	Ignored	This rule is ignored for XML export.
`write sgml document instance only`	Ignored	This rule is ignored for XML export. The software always writes out the document instance without the doctype subset.

Handling invalid element names

The rules for characters allowed in XML element names is more restrictive than for SGML. XML element names must start with a letter, and spaces, colons, periods, or under-scores are not allowed. If used in the source FrameMaker element name, one or more consecutive invalid characters are transformed on export into a single dash (-). If an element starts with one or more invalid characters that are not a space, the letters “FM” are prepended to the beginning of the name, and leading and trailing spaces are stripped. For example:

```
Front Matter-> Front-Matter
3Chapter -> FM3Chapter
_body -> FM-body
```

Notice that in the last example, the first character is not only an invalid first character, but an invalid character in general. FM is prepended and the invalid character is changed to a dash.

Default attribute value

Default attributes -- those attribute values that are supplied by the DTD and not specified in the document instance -- only apply to structured flows. Since a DTD is not written out on XML export, default attribute values must be otherwise specified. Default attributes and their values are always written out. The software first looks to see if the DTD specifies a default value; if so, it is used. Otherwise, the default value from the EDD is used.

I Developing SGML Publishing Applications

Implementing an SGML application in the FrameMaker+SGML publishing environment

This paper describes the relationship between SGML and FrameMaker+SGML, and the task of developing FrameMaker+SGML applications. It has three major sections:

- An explanation of the FrameMaker+SGML application philosophy
- An introduction to the technical steps in application development
- A description of some typical application development scenarios

Overview of FrameMaker+SGML Application Development

This section examines the relationship between SGML and various types of SGML applications. It also presents an overview of developing applications for FrameMaker+SGML.

SGML Applications

The growing need to reuse complete or partial documents is one result of today's constant increase in the quantity of business information and the cost of maintaining it. The way SGML separates a document's content from its appearance and intended processing makes it a natural tool for enabling reusability. However, the ability to use text in different ways requires that each use be defined. In SGML 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." All SGML tools, including FrameMaker+SGML, require such SGML applications.

Understanding SGML Applications The need for SGML applications is easily understood by comparing general-purpose SGML tools to traditional database packages. Just as database software is used to maintain and access collections of numeric data and fixed-length text strings, an SGML 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	SGML
Depends on a schema to define record layout	Requires a document type definition (DTD) to define structural elements and attributes
Input forms facilitate data entry	Configured 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 an SGML application.

These ideas are easily illustrated with an example: While SGML 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 publishing in additional media, such as CD-ROM and the Internet, as well as other applications, such as database distribution.

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 SGML prepares information for multiple presentations without making assumptions about its rendering, additional information is needed to control each rendering. This formatting information, pertinent to an application, is nevertheless outside the scope of SGML itself.

Consider, for instance, a repair manual stored in SGML. 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>
```

Because of the lack of formatting as well as the appearance of the markup codes, mechanics would doubtless find this procedure difficult to read in this form. They would be much more comfortable—and accordingly more efficient in performing the maintenance job—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 taskmodule, steps, and so on of the SGML example are to appear. For example, these rules:

- Put the heading in a large, bold font.
- Generate the sentence, “This procedure requires advanced certification” from the value of the skill attribute.
- 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.

SGML Editing and Publishing Applications Two important classes of SGML applications involve creating (editing) and publishing documents. Tools from different vendors take different approaches to the two applications. Editing tools are used to create SGML documents; publishing tools produce formatted results from existing SGML documents. For example, Datalogics' Composer and Arbortext's Adept Publisher are publishing tools that operate on completed SGML documents using FOSIs (Formatting Output Specification Instances) to specify page design and the visual characteristics applied to various structural elements. James Clark's JADE is a non-commercial publishing tool that uses DSSL (Document Style and Semantic Specification Language) to specify the formatting rules of an application.

Since users can best work with an editor if they have some visual indication of the document's structure, applications for SGML text editors, such as Adobe FrameMaker+SGML, Softquad's Author/Editor, and Arbortext's Adept Editor, also require some formatting specifications. Some editors also provide for rules to determine:

- How closely documents under development must conform to a DTD
- 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+SGML is a single tool that addresses both. Although FrameMaker+SGML can be used for either one in isolation, 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+SGML than to configure separate editing and publishing tools.

Adobe FrameMaker+SGML Application Development

Developing a FrameMaker+SGML application shares many steps with developing an SGML application for any other editing or publishing tool. This section summarizes the process, covering the steps common to other editing and publishing tools, and those unique to FrameMaker+SGML. Itemization of the various steps is preceded by a brief description of the approach to document processing that FrameMaker+SGML uses.

FrameMaker+SGML Editing Philosophy As the preceding formatted `taskmodule` example illustrates, visual clues—font variation, indentation, and numbering—help readers understand and use written information. One recent trend in electronic publishing is that of WYSIWYG editing, which allows authors to use the same formatting clues during a document's development. FrameMaker+SGML is a WYSIWYG editor. While most WYSIWYG editors require authors to indicate how each part of the document is formatted,

FrameMaker+SGML uses the rules in the SGML application to format document components automatically in a consistent manner.

This combination of WYSIWYG techniques with the structured principles of SGML is unique to FrameMaker+SGML. In fact, not too many years ago, some members of the SGML community believed that WYSIWYG editing was incompatible with SGML. The perceived conflict derived from the focus on SGML markup that underlies the typical SGML editor. Since the goal of such tools is the creation of SGML documents, authors must understand SGML syntax and conventions as they are editing. The goal in FrameMaker+SGML, however, is to improve the process of creating and publishing documents through the use of SGML. While authors manipulate elements and their attributes to create native SGML documents, they work in a WYSIWYG fashion instead of directly with SGML syntax.

Since the SGML document model underlies edited material, the SGML 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 SGML from word processing formats.

WYSIWYG techniques are especially valuable for authors who work with visually oriented constructs such as graphics and tables. In addition to its WYSIWYG user interface for editing these structures, FrameMaker+SGML offers numerous SGML representations for them. One component of a FrameMaker+SGML application includes rules for choosing the desired SGML representation.

Tasks in FrameMaker+SGML application development The major tasks involved in the development of a FrameMaker+SGML 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 an SGML 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 of the 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 SGML and FrameMaker+SGML representations of a document. For most elements, this correspondence is straightforward and automatic, but FrameMaker+SGML allows for some variations. For example, terse SGML names can be mapped into longer, more descriptive FrameMaker+SGML names, and variant representations of tables and graphics can be chosen.

Each of these tasks requires planning and design before implementation, as well as testing afterward. The rest of this paper describes these steps, and the skills needed to perform them, in more detail.

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

- Will users create new content, or will the application simply format existing SGML documents?
- Whether or not they require further editing, are existing SGML documents to be brought into the system? What about non-SGML documents (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 SGML? Will authors themselves output SGML, 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 SGML 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+SGML. The bulk of this effort is often the creation of a DTD. When the project begins with an existing SGML DTD, the DTD provides a foundation for the structure definition. FrameMaker+SGML, 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 generic identifiers, 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 name `Book` during editing, but automatically convert `Book` to and from `CiteTitle` when reading or writing the SGML form of the document.

- Omitting unnecessary elements and attributes.

  DocBook defines about 300 elements. Many organizations actually use only a small fraction of the possibilities. There is no need to make the remainder available in an interactive application. FrameMaker+SGML 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 alternate 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, the editing DTD can eliminate the unnecessary ones.

In addition to providing an editing DTD, the structure definition 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 509. The formatted version did not display timing information for the procedure steps. This alternate version formats the procedure in a three-column table, using

the first column for the step number, the second column for the expected time to complete the step, and the third column to describe the step:

> Tuning a 5450A Widget
>
> *This procedure requires advanced certification.*
>
> 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.(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+SGML—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 SGML projects. Thus, SGML itself has acquired an undeserved reputation in some circles for being complex and difficult to use. In fact, poorly designed SGML applications are often at fault. Many things can be done to make SGML easier and more appealing to the user. For example, FrameMaker+SGML menus can be customized, both by adding new application-specific commands defined through the FDK and by simplifying the menus in 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+SGML Application Development" on page 516 describes the steps involved in implementing a FrameMaker+SGML 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 SGML application (for FrameMaker+SGML 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, SGML import, and SGML export. Test documents must include FrameMaker+SGML documents and—if SGML import is intended—SGML documents. Testing must also include any processing of legacy documents, including the use of the FrameMaker+SGML conversion utility.

Some documents can be created specifically for testing. Others may derive from typical SGML documents that conform to an existing DTD. Still others may be legacy documents in hard copy, a word processor format, or unstructured FrameMaker form. They can be scanned, filtered, or converted to SGML as appropriate. 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+SGML 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 SGML 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 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

- SGML
- FrameMaker+SGML
- Any other SGML 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. With the current popularity of the Internet, for example, many documents initially prepared with FrameMaker+SGML are distributed on the World Wide Web, and the organization's Web advocate may be able to identify some requirements that do not affect the rest of the team.

While project managers can inject a note of realism in scheduling requirements, they must also be prevented from making unfeasible demands. For a large project, some disagreements among team members are to be expected, and the skills of an unbiased facilitator can keep the team focused and productive. In many organizations, executive sponsorship is required to make sure the required resources are available in a timely fashion, even in the analysis phase.

In the implementation phase, the developers require general expertise in graphic design and structured documents. The skill sets required within the implementation team include document design, SGML knowledge, setting up FrameMaker+SGML formatting templates, and setting up the formatting rules that control automatic application of the desired graphic design to structured documents. If the FDK is used, programming skills are also needed. These various abilities can be shared by a group of people; there is no need for one individual to master them all.

Finally, technical writing skills in documenting the finished application are a valuable contribution to the completed effort.

Technical Steps in FrameMaker+SGML Application Development

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

Element Definition Documents

At the heart of every FrameMaker+SGML 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 the DTD, if there is one.
- 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.

Since an EDD is a structured document, the FrameMaker+SGML Guided Editing feature assists developers in creating proper EDDs. In particular, developers need not remember where to insert element definitions, context specifications, or format rules. The Structure View and Element Catalog guide them in creating these constructs correctly. The formatting of the EDD increases its readability: Different fields are clearly labeled, and spacing, indentation, and different fonts emphasize its organization. Developers can increase 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 Element Definition Document 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

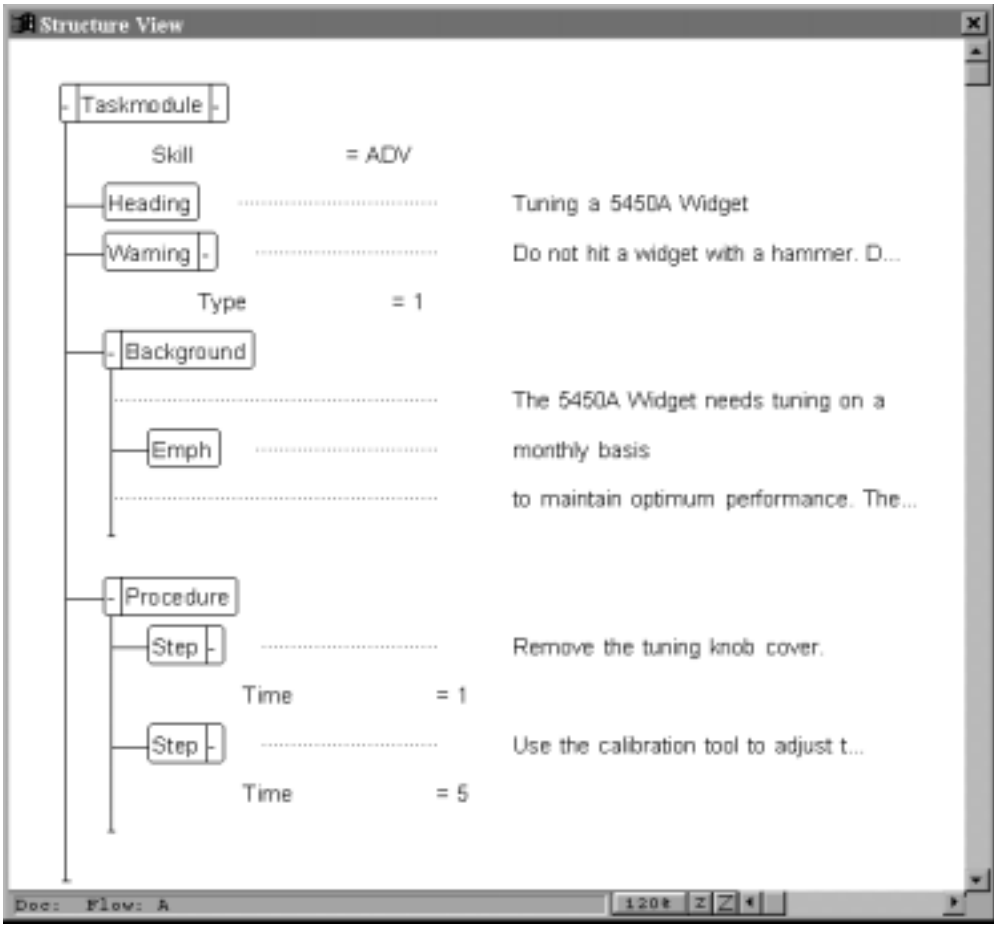
Structured Templates

End users do not use EDDs directly. Instead, the definitions made in an EDD are extracted and stored in a template. A template is simply a FrameMaker+SGML 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. The template also includes page layout information and formatting catalogs that define paragraph, character, cross-reference, and table styles. It may also contain sample text.

In some applications, information may be formatted in several ways. For example, the same text may need to be formatted for books with two page sizes. FrameMaker+SGML 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 SGML and FrameMaker+SGML

FrameMaker+SGML maintains the native element and attribute structure of SGML in a WYSIWYG environment. The FrameMaker+SGML 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+SGML rendition and the SGML text file include content and element structure, data can move between the two forms automatically. The user interface is straightforward. The FrameMaker+SGML File > Open command recognizes SGML documents. When a user opens one, FrameMaker+SGML automatically converts the document instance to a structured WYSIWYG document and applies the format rules of the appropriate SGML application. To write a structured WYSIWYG document to SGML, the user simply specifies SGML when executing the File > Save As command.

Just as the form of a structured document parallels the form of an SGML document instance, an EDD parallels a DTD. FrameMaker+SGML can also move definitions of

possible structures between the two forms. Thus, if a new project is based on an existing SGML DTD, FrameMaker+SGML can automatically create an EDD from the DTD. For example, given DTD declarations such as

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

FrameMaker+SGML 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. The developer can manually edit this information into the EDD.

Adding format rules to an EDD needs to be done only once. It is common for a DTD to undergo several revisions during its life cycle. When the developer receives an updated DTD, FrameMaker+SGML can automatically update an existing EDD to reflect the revision. While the update process preserves existing formatting information and comments in the EDD, it incorporates changes to content models and attribute definitions as well as inserting new element types and removing discarded ones. Furthermore, it generates a short report summarizing the changes so that the application developer may review them.

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

Customizing SGML Import/Export

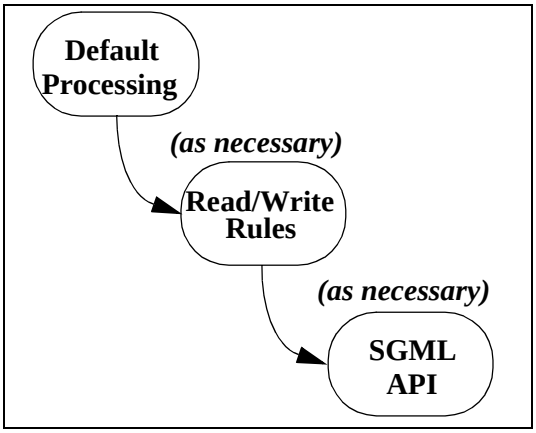
As mentioned, the developer may wish to customize the correspondence between a document's SGML and FrameMaker+SGML representations for several reasons. Possibilities include

- Basing the FrameMaker+SGML application on an authoring version of an interchange DTD
- Integrating FrameMaker+SGML with database or document management tools
- Calculating portions of a document or displaying predefined text when certain elements or attribute values occur or particular entities are used
- Interpreting certain elements, such as graphics or tables, as special objects

To support such requirements, FrameMaker+SGML offers three strategies for specifying how abstract SGML structures are represented in the WYSIWYG publishing environment. Most projects combine two or more of these approaches:

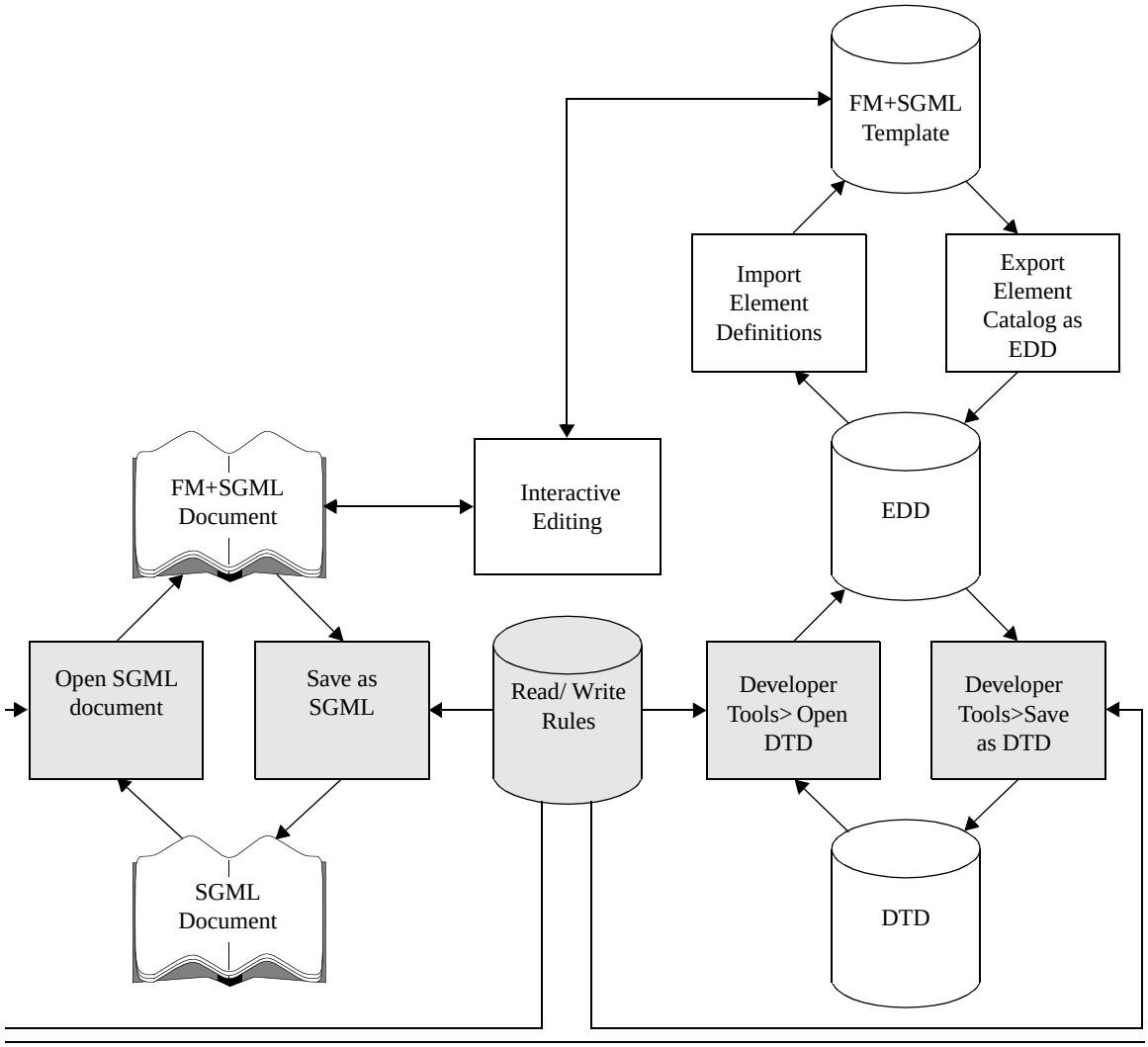
- By default, FrameMaker+SGML automatically translates an SGML element into a FrameMaker+SGML element with the same name (and translates a FrameMaker+SGML element to an SGML element with the same name); similar processing occurs for attributes. For many elements, the default processing is adequate.
- Some customizing can be accomplished through read/write rules. FrameMaker+SGML provides read/write rules for customizing that is common to many applications. For example, there are read/write rules for mapping terse `NAMELEN`-bounded SGML names to longer FrameMaker+SGML equivalents; for mapping SGML elements to special-purpose FrameMaker+SGML elements such as tables, graphics, or markers; and for associating `SDATA` entities with special characters. The developer enters 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 SGML Api, which enables more extensive customizing. The SGML api can be used, for instance, to extract part of a document's text from a database using attribute values as database keys.

These three approaches are diagrammed following diagram:



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

Note that in the diagram there is no designated starting point or direction for the flow of information; different projects enter the flow at different stages and follow the arrows in the appropriate direction.



SGML application files

As explained in the preceding section, moving documents and DTDs between their SGML and FrameMaker+SGML 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 an application definition. Definitions for all SGML applications available on a particular computer system are grouped in an application file. One of the FrameMaker+SGML configuration files, `sgmlapps.fm`, is an SGML application file that FrameMaker+SGML reads whenever it starts. The following example is a fragment of a typical SGML application file:

Application Definition Version 5.0

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

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

Since FrameMaker+SGML automatically reads `sgmlapps.fm`, end users do not need to know about the SGML application file. To import or export SGML documents, however, they may need to know the names of available applications. But because an application can be stored in a FrameMaker+SGML template or automatically selected from the document type name of an SGML document, sometimes even knowing the names is not always necessary. For example, given the preceding SGML application file example, when reading an SGML document that begins

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

FrameMaker+SGML uses the Maintenance application. Since 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 will be used if the document is later exported back to SGML.

The information in an application definition can include:

- 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 SGML 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 SGML 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.

- One or more document type names that trigger use of the application on import.
- Identification of an SGML Api client to use during import or export.
- The name of the file containing the application's read/write rules.
- Files containing external entities (identified by entity name or public identifier).
- Search paths for subfiles of read/write rules and external entities.
- An SGML Open entity catalog

Legacy Documents

In general, the conversion of unstructured documents to structured documents (or to SGML documents) cannot be completely automated. FrameMaker+SGML 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+SGML can read several 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 SGML. Thus, FrameMaker+SGML provides a path for converting word processor documents to SGML.

The rules for mapping between styles and elements are specified in a conversion table. Techniques for defining the mapping are discussed below. Consider a very simple unstructured document that consists of a paragraph tagged `MainTitle` followed by a paragraph tagged `ChapterTitle` and one or more paragraphs tagged `Body`. Each sequence of `Body` paragraphs may be followed by another `ChapterTitle` and more `Body` paragraphs. The `Body` paragraphs contain plain text and may also have some characters with the character format `Emphasis`. A conversion table for such a document appears below:

Wrap this object or objects	In this element	With this qualifier
P:MainTitle	Title	Main
P:ChapterTitle	Title	Chapter
P:Body	Para	
C:Emphasis	EmphPhrase	
E:Title[Chapter], E:Para+	Chapter	
E:Title[Main], E:Chapter+	Doc	

Each row in the table is a separate rule. The first row maps the paragraph tagged `MainTitle` to an element called `Title` and, during the structuring process, associates the qualifier `Main` with this particular `Title` element. The next row specifies that each `ChapterTitle` paragraph is also a `Title`, but qualifies these `Titles` by `Chapter`. The next row says that `Body` paragraphs become `Para` elements, and the following row wraps text ranges with the character format `Emphasis` to `EmphPhrase` elements.

The rules in the next two rows combine these low-level elements to define higher levels in the hierarchy. The penultimate row says that a `Title` element followed by one or more `Para` elements is a `Chapter`, provided that the `Title` is qualified by `Chapter`—in other words, a `Chapter` is a chapter `Title` followed by some `Paras`. The last row says that a `Title` qualified by `Main` followed by some `Chapters` is a `Doc`.

Note that there is no need to specify that an `EmphPhrase` element can appear within a `Para`. FrameMaker+SGML automatically places text range elements within the element for the paragraph in which they appear.

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. The FDK clients must be installed according to platform-dependent conventions. 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 SGML documents, their SGML application files must be modified for the new application. They need access to the read/write rules, entity catalog, DTD for export, and SGML declaration.

Typical Application Development Scenarios

Although creation of each component of a FrameMaker+SGML application is straightforward, novice developers may need some guidance as to the order in which to create these components. Suggestions are especially appropriate, because the strategy for creating an application can vary considerably with the requirements of a particular project. This section therefore describes three scenarios and how developers may proceed. The 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 SGML Documents

In the first scenario, there is an existing SGML DTD and sample SGML documents. The developer selects a short sample document and starts development by supporting the element types that occur in it. Later, the developer adds more complicated documents. Application development proceeds as follows:

1. The developer creates a new file of read/write rules and enters rules that reflect any decisions made during the analysis phase of the project. In particular, the developer enters 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 SGML names with longer, more readable FrameMaker+SGML versions (unless the SGML declaration specifies a `NAMELEN` quantity that permits long names in the SGML version)

   -Specify FrameMaker+SGML names with multiple capital letters. For instance, the developer may map the SGML generic identifier `BLOCKQUOTE` to the FrameMaker+SGML element tag `BlockQuote`. Such rules will not be needed if the SGML declaration does not force general names to uppercase (that is, if case is significant in the SGML names).

2. The developer defines a new SGML application in the SGML application file that invokes the read/write rules. If appropriate, the application definition specifies an SGML application and maps any public identifiers used in the DTD to filenames.

3. Using the new application, the developer creates an EDD from the DTD with the File > Developer Tools > Open DTD command. The resulting element declarations may be grouped into sections, and some added comments in the process.

4. The developer opens the selected sample document. The resulting FrameMaker+SGML document serves as the basis of a template. In this document, the developer defines page layouts, specifying the number of columns on the page, the width of any side-heads, page numbers, and so forth.

5. The application developer returns to the EDD and begins to define formatting rules. In general, formatting specifications are unnecessary for higher level elements, such as those for chapters and sections. The application developer concentrates 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, the application developer tests 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+SGML document. The developer defines all reported formats and glances through the sample document to verify that the formatted elements appear as intended.

   The developer then returns 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 the developer tests some other documents. After creating a base

for further development, the developer continues systematically through the DTD, making and testing context-sensitive format rules for all necessary elements.

7. The developer may add more read/write rules. If the developer changes 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, the developer uses the File > Developer Tools > 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, the developer deletes all content and saves the result.

9. The developer can then 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+SGML should automatically insert an `Item` element within it. The developer also considers whether an FDK client can provide input accelerators, such as automating frequent sequences of editing steps. The developer may also customize the FrameMaker+SGML 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 SGML import, the developer adds the template to the application definition in the SGML application file, and adds read/write rules for special characters and other entities. Once again, the developer opens the SGML version of the sample document, tests the result, and modifies the application until the results are correct.

11. If read/write rules are insufficient to import any elements, the developer uses the SGML api to develop a client with the requisite functionality.

12. If SGML export is also required, the developer tests it, adding other read/write rules or SGML api functions, if necessary.

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

Building a New Application

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

1. Instead of starting with a DTD, the developer begins to develop an EDD. Guided Editing makes the mechanics of editing the EDD simple. It is the developer's personal preference whether to define both content rules and attribute definitions for one element at a time, or to complete all content rules before defining any attributes. The developer may also define content rules for numerous elements before adding any format rules.

2. The developer creates a template and then uses the File > Import Element Definitions command to interpret the EDD and store the resulting element catalog in the template.
3. The developer uses the template to test sample documents. As in the scenario of the previous section, the developer 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, the developer uses the File > Developer Tools > Save As DTD command to create an equivalent DTD.
5. The developer creates an SGML application that uses the generated DTD for export.
6. The developer tests the export of the sample document and adds read/write rules and an SGML api client as necessary. If the developer does not want to use the FrameMaker+SGML default representation for special object elements, editing the corresponding element and attribute definition list declarations is the thing to do, in addition to providing the associated rules.
7. The developer may edit the generated DTD by allowing for tag omission and providing appropriate short reference mapping, short reference use, and entity declarations. Since FrameMaker+SGML exports all comments in the EDD to the generated DTD, the developer reviews comments carefully and retains comments pertaining to the structures being defined, but discards comments specific to FrameMaker+SGML formatting, since they do not necessarily apply to all uses of the DTD.
8. If the application requires SGML import as well as export, the developer tests conversion in the import direction as well, modifying the application to correct any errors.
9. Finally, the developer delivers the application to end users and prepares 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. The developer selects a typical unstructured document and uses the File > Developer Tools > 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	With this qualifier
P:MainTitle	MainTitle	
ChapterTitle	P:ChapterTitle	
P:Body	Body	
C:Emphasis	Emphasis	
G:	GRAPHIC	
F:flow	FOOTNOTE	
T:Table	Table	

2. The developer edits the conversion table, changing entries in the second column, adding entries in the third column, and adding new rows for higher level structures.
3. The developer uses 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 File > Developer Tools > Export Element Catalog as EDD command, the developer extracts 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 the developer can further develop.
5. The exported EDD lists the element definitions alphabetically, according to their tags. The developer 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. The developer then continues 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.

Conclusions

This paper has described how all general-purpose SGML systems require application-specific information. It has enumerated the robust and flexible set of tools that FrameMaker+SGML provides for creating SGML editing and publishing applications. Application development can be divided into separate steps, each of which is straightforward. The beginning of “Overview of FrameMaker+SGML Application

Development" on page 507 listed the major parts of an application. The following table reviews the list, along with the application components in which the information is specified:

Application Component	File
Element and attribute definitions	Edd (can be derived automatically from DTD)
Format rules	Edd
Page layouts	Template
SGML representation of elements and attributes	Read/write rules and SGML Api client

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

Adobe, the Adobe logo, and FrameMaker are trademarks of Adobe Systems Incorporated.

©1997 Adobe Systems Incorporated. All rights reserved. Printed in the USA. 000000 9/97.

Glossary

This glossary contains common terms used by FrameMaker+SGML and SGML. For references to more information about the terms, see the index.

ancestor

An element that contains a given element in a document's structure. For example, if a `Section` element contains a `Head` element followed by a `Paragraph` element, and the `Paragraph` contains a `Variable` element, the `Paragraph` and `Section` elements are both ancestors of the `Variable` element, but the `Head` element is not an ancestor of the `Variable` element. *See also* descendant, child element, parent element, and sibling.

API

Application Programming Interface. Enables developers to create API clients with other applications, such as databases, document management systems, CAD tools, and user interfaces, for automation, database publishing, HTML conversion and other purposes.

application definition

A data structure (and the associated files) describing part of a complete SGML application assembled with FrameMaker+SGML. You store application definitions in the `sgmlapps.doc` file.

attribute

A place to supply information about an element other than its hierarchical position and structure. An attribute value does not add content to a document.

attribute definition

The construct used to define a single attribute in a FrameMaker+SGML EDD or an SGML DTD.

attribute definition list declaration

In SGML, the declaration that provides the list of attribute definitions for one or more elements. Also called an ATTLIST. *See also* element declaration.

book

A grouping of FrameMaker+SGML documents that lets you work with them as a single unit. Lets you generate a single table of contents or other file from the documents, and simplifies printing, numbering, cross-referencing, and formatting.

CALS

Continuous Acquisition and Life Cycle Support. The US Department of Defense standard for the electronic delivery of documents.

catalog

A floating palette that stores predefined paragraph, character, or table formats.

CDATA

In SGML, character data. In character data, no markup is recognized, other than the delimiters that end the character data. *See also* NDATA, #PCDATA, RCDATA, and SDATA.

child element

An element that is contained in a given element and that is one level below the given element. For example, if a `Section` element contains a `Head`

element followed by a `Paragraph` element, and the `Paragraph` element contains a `Variable` element, the `Head` and `Paragraph` elements are both child elements of the `Section` element, but the `Variable` element is not. *See also* parent element, ancestor, descendant, and sibling.

concrete syntax In SGML, a set of choices on the markup a document will use. Since SGML does not require any particular values for these choices, an SGML document requires a concrete syntax so a parser can correctly interpret it. *See also* reference concrete syntax.

container element In FrameMaker+SGML, an element that can contain text, other elements, or both. Contrasts with certain specific element types—for example, a cross-reference element, which can contain nothing other than the cross-reference.

content model In SGML, the part of an element declaration that specifies both a model group and exceptions that define the allowed content of the element. Each SGML element declaration has either a content model or declared content. *See also* content rules, declared content, general rule, and model group.

content rules In FrameMaker+SGML, the part of an element declaration that specifies both the element's type and the kind of contents the element can have. *See also* format rules, content model, and general rule.

conversion table In FrameMaker+SGML, a table associating parts of an unstructured document with their structured counterparts, used in converting an unstructured document to a structured document.

cross-reference A passage in one place in a document that refers to another place, its cross-reference source, in the same or a different document.

cross-reference source The place referred to by a cross-reference.

data In SGML, the characters of a document that represent the inherent information content. Such characters are not recognized as markup. *See also* markup.

data content notation In SGML, an application-specific interpretation of an element's data content, or of a data entity, that usually extends or differs from the normal meaning of the document character set. Frequently used to identify the format of an external entity containing a graphic.

declaration In SGML, markup that controls how other markup of a document is to be interpreted.

declared content In an SGML element declaration, specifies that the defined element's content is one of the reserved types `CDATA`, `RCDATA`, or `EMPTY`.

declared value In an SGML attribute definition, determines the type of attribute value, such as `ID` or `NUTOKEN`, that is valid when the attribute is specified. Although SGML does not define the term *attribute type*, you can loosely think of an attribute's declared value as its type.

default value In SGML, the portion of an attribute definition that indicates whether an attribute is required and what value to use if the user doesn't specify one. In FrameMaker+SGML, refers only to the value to use if a user doesn't supply a value for an attribute.

delimiter In SGML, a character string used to identify a piece of markup or to distinguish markup from data. For example, > (greater-than sign) is the default closing delimiter for element tags.

descendant Any element that is below a given element in a document's structure. For example, if a `Section` element contains a `Head` element followed by a `Paragraph` element, and the `Paragraph` element contains a `Variable` element, the `Variable` element is a descendant of both the `Paragraph` and the `Section` elements, but not of the `Head` element. *See also* ancestor, child element, parent element, and sibling.

DOCTYPE In SGML, the reserved name that follows the opening delimiter of a DTD. Informally used to refer to the document element.

document A collection of information that is processed as a unit. A FrameMaker+SGML document is any file in FrameMaker+SGML format. An SGML document includes an SGML declaration, prologue, and document instance set.

document element In SGML, the highest-level element in a document. The generic identifier of this element is specified immediately after the `DOCTYPE` reserved name in the DTD.

document instance In SGML, the portion of a document that contains markup and data for a particular project such as a memo or book.

document type A class of documents having similar characteristics, such as technical manual or internal memo.

document type declaration In SGML, a document type declaration (DTD) associates a document element with a set of declarations (the document type declaration subset).

document type declaration subset In SGML, a set of declarations determining such things as the markup to allow in a document and the elements and attributes for a document set. *See also* external DTD subset and internal DTD subset.

DTD *See* document type declaration subset.

EDD *See* element definition document.

element A structural unit of a document. Holds and organizes the contents of the document.

Element Catalog In FrameMaker+SGML, the information extracted from an EDD and stored within each structured FrameMaker+SGML document. Makes an external element definition document unnecessary. *See also* element definition document.

element declaration In SGML, information describing a particular element. Includes both a name (generic identifier) for the element and content rules. An SGML document has an element declaration for each allowed element.

element definition In FrameMaker+SGML, a set of rules describing an element. Includes a name (tag) for the element, content rules, and (optionally) context-sensitive format rules. A structured document has an element definition for each element allowed. *See also* content rules and format rules.

element definition document A FrameMaker+SGML document that contains a set of element definitions for a class of documents. Can also include information on system defaults and on an SGML application with which to associate this information. Also called an EDD.

element tag In FrameMaker+SGML, the name assigned to an element and stored in the Element Catalog. *See also* generic identifier.

EMPTY Keyword in an element definition indicating that the element cannot have content. In SGML, `EMPTY` is a declared content.

end-tag In SGML, the markup that indicates the end of an element.

entity In SGML, a collection of characters that can be referenced as a unit. Used for many purposes in SGML, such as graphics or frequently used sets of characters.

exclusion An exception to the general rule or content model of an element. Specifies other elements that cannot appear anywhere in the element or in its descendants.

external cross-reference In FrameMaker+SGML, a cross-reference to a source in a different file. SGML does not define this concept.

external DTD subset In SGML, an informal term for an external entity for which an external identifier appears at the beginning of a document type declaration and that is automatically referenced at the end of the document type declaration subset.

external entity In SGML, an entity that specifies an external object, such as a file.

facet A pictorial representation of graphical data.

FDK client In FrameMaker+SGML, any application created using the Frame Developer's Kit. *See also* SGML API client.

flow *See* , "text flow."

format rules In FrameMaker+SGML, the part of an element definition that specifies which predefined format to apply to an element. Format rules can use different formats for different contexts in a document. *See also* content rules.

general entity In SGML, an entity that can be referenced from within the content of an element or an attribute value literal.

general rule In FrameMaker+SGML, a rule that specifies valid contents for an element and the order in which the contents can appear. Equivalent to the declared content of an element or the model group part of the content model of an element in SGML. *See also* content rules.

generic identifier In SGML, the name identifying an element. *See also* element definition and element tag.

highest-level rule In FrameMaker+SGML, an SGML read/write rule that is not a subrule of another read/write rule.

HTML Hypertext Markup Language. An encoding system used to describe the content and organization of an electronic document published on the World Wide Web.

ID attribute An attribute of type `ID`, frequently used as an identifier to mark the source of a cross-reference. In a single document, a particular value for an `ID` attribute can be used only once.

IDREF attribute An attribute whose value must be that of an `ID` attribute in the same SGML document or FrameMaker+SGML document or book. Frequently used for cross-references.

impliable attribute In SGML, an attribute whose value does not have to be supplied. If a document doesn't supply a value, it is up to the processing software to correctly interpret the attribute. Such attributes use the default value `#IMPLIED`.

inclusion An exception to the general rule or content model of an element. Specifies other elements that can appear anywhere in the element or in its descendants.

invalid element An element with contents that do not conform to content rules. May be missing required child elements, may not have a definition in the EDD or DTD, or may have text or child elements in a position not allowed by its content rules or by the exclusion and inclusion rules of its ancestors.

internal cross-reference In FrameMaker+SGML, a cross-reference to a source in the same file.

internal DTD subset In SGML, an informal term for the declarations in a document type declaration that occur within brackets (dso and dsc delimiters) in the SGML document entity, rather than being in an external entity.

internal entity In SGML, an entity whose replacement text is determined solely by information in its declaration.

ISO public entity In SGML, an entity that occurs in one of the entity sets defined in Annex D of the SGML Standard. These entities provide commonly used special characters.

marker In FrameMaker+SGML, a nonprinting character an end user inserts (such as an index entry) to indicate various types of information.

markup In SGML, text added to the data of a document in order to convey information about it, such as hierarchical structure or formatting.

markup minimization In SGML, any of various conventions for omitting markup in a document, including shortening or omitting tags and shortening entity references.

model group In SGML, an ordered list that specifies valid contents for an element (such as child elements) and the order in which the contents can appear. An SGML model group is similar to a FrameMaker+SGML general rule.

NAMECASE parameter In SGML, the part of the SGML declaration that determines case-sensitivity of markup.

NDATA In SGML, non-SGML data. `NDATA` is data that needs special processing by the SGML application. `NDATA` is typically used, for example, when representing graphics. *See also* CDATA, #PCDATA, RCDATA, and SDATA.

parameter entity In SGML, an entity that can be referenced only within declarations.

parent element An element that contains a given element and is one level above it in the hierarchy. For example, if a `Section` element contains a `Head` element followed by a `Paragraph` element, the `Section` element is the parent element of the `Head` and `Paragraph` elements, but not of the `Variable` element. *See also* child element, ancestor, descendant, and sibling.

parser *See* validating parser.

#PCDATA In SGML, parsed character data. This is normal text that can include markup to be parsed. Occurs in an SGML element's model group and corresponds to <TEXT> in a FrameMaker+SGML element's general rule. *See also* CDATA, NDATA, RCDATA, and SDATA.

prefix Text that is automatically placed before the content of an element. In FrameMaker+SGML, defined as part of the formatting of an element. For example, a `Quote` text range element might have an open quotation mark as its prefix and a close quotation mark as its suffix. *See also* suffix.

processing instruction In an SGML document, a way of indicating that the SGML application needs to perform some special processing. For example, you can use a processing instruction to indicate a location in an SGML document that should have a page break.

public identifier In SGML, a way of identifying an external entity. Formal public identifiers have a specified syntax that includes an identifier of the owner of the entity and an indication of the SGML construct it provides. Formal public identifiers are typically available to any user of SGML, not just the users at a particular company. Informal public identifiers may be available more widely than a single

	document or system, but perhaps no more widely than within a single company. *See also* system identifier
RCDATA	In SGML, replaceable character data. In replaceable character data, no markup is recognized, other than character and entity references. *See also* CDATA, NDATA, #PCDATA, and SDATA.
read/write rule	*See* SGML read/write rule.
reference concrete syntax	In SGML, a particular concrete syntax defined by the SGML standard. *See also* concrete syntax.
reference page	An underlying page that stores repeatedly-used graphics and formatting information.
Rubi text	Small characters that appear above Japanese-language characters to indicate pronunciation.
rule	*See* SGML read/write rule.
SDATA	In SGML, specific character data. One common use is for specific characters that might not be in the standard character set. *See also* CDATA, NDATA, #PCDATA, and RCDATA.
SGML	An acronym for Standard Generalized Markup Language.
SGML API client	In FrameMaker+SGML, an FDK client created to change the translation between FrameMaker+SGML and SGML documents. *See also* FDK client.
SGML application	Rules that apply SGML to a text processing application. Includes a formal specification of the markup constructs used in the application, expressed in SGML. Can also include non-SGML definitions of semantics, application, conventions, and processing.
SGML declaration	In SGML, the part of a document that tells a parser how to interpret markup in the document.
SGML read/write rule	In FrameMaker+SGML, information you supply to modify how the software translates between FrameMaker+SGML and SGML documents.
SGML text entity	An entity whose replacement text can contain both data and markup.
sibling	Elements at the same level in the structure and with the same parent element. For example, if a `Section` element contains a `Head` element followed by a `Paragraph` element, the `Head` and `Paragraph` elements are siblings. *See also* ancestor, descendant, child element, and parent element.
source	*See* cross-reference source.
start-tag	In SGML, the markup that indicates the beginning of an element.
subrule	In FrameMaker+SGML, an SGML read/write rule that is part of another rule.

suffix Text that is automatically placed after the content of an element. In FrameMaker+SGML, a prefix is defined as part of the formatting of an element. *See also* prefix.

system identifier In SGML, a way of identifying an external entity that's specific to the particular document or system. *See also* public identifier.

template In FrameMaker+SGML, a document used to create new documents. A template can include all the formats, structure descriptions, and other information you need to create a document.

<TEXT> In a FrameMaker+SGML element's general rule, indicates that the element can directly contain text characters and elements included by itself or its ancestors. `<TEXT>` corresponds to `#PCDATA` in an SGML element's model group.

text flow The text in a series of connected text frames. A text flow can also be contained in a single text frame, not connected to any other frame. A text flow with elements is a structured text flow.

text inset Text imported by reference.

<TEXTONLY> In a FrameMaker+SGML element's general rule, indicates that the element can directly contain text characters and cannot contain elements included by an ancestor. By default, on export `<TEXTONLY>` corresponds to a declared content of `RCDATA` in an SGML element's definition and on import to either a declared content of `RCDATA` or of `CDATA`.

valid document A structured document that conforms to all its content rules. Every element in the document must be valid. In FrameMaker+SGML, every structured flow must have a highest-level element that is allowed at the highest level.

valid element An element with contents that conform to its own content rules and to the inclusion and exclusion rules of all of its ancestors.

validating parser In SGML, a software module that takes an SGML document and interprets the data and markup to recognize the parts of the document, such as entity declarations and element start-tags.

variable In FrameMaker+SGML, text that is defined once but can be used several times. Similar to some varieties of SGML entity.

Index

A

absolute values, in format rules 116
Advanced properties (text formatting) 141-142
{after} sibling indicator 123, 177
alignment settings
  for paragraphs 136
  for table cells 143
all-contexts rules (for formatting) 121, 175
ampersand (&)
  in conversion tables 452
  in format rules 124
  in general rules 100
ancestors
  describing relationships to siblings 122, 177
  inheriting formats from 115-119
  tags in context rules 121, 176
  tags in level rules 126
anchored frame (rule) 333
anchored frames. *See* graphics
angle brackets (< >)
  in prefix or suffix strings 146
  with attributes in prefix or suffix rules 149
{any} sibling indicator 123, 177
ANY keyword, in general rules 101
application definition files 32, 42-57
  contents of 43
  default information 46
  defining applications in 44
  document elements 47
  DTDs for import and export 48
  editing 42
  entity catalogs 49-50
  external entities 51-53
  files for rules documents 55
  individual entities 51
  length of log files 57
  public identifiers 53
  read/write rules documents 47
  search path for external entities 53-55
  SGML API clients 56
  SGML declarations 48
  templates for import 48
Asian text spacing properties (text formatting) 143
asterisk (*)
  in conversion tables 452
  in format rules 122, 176
  in general rules 99
attribute definitions 157-171
  attribute name 160
  attribute type 160
  correspondence to SGML 207-208
  default values 164
  hidden and read-only specification 162
  list of choice values 163
  optional-value specification 161
  range of numeric values 163
  required-value specification 161
attribute names
  correspondence to SGML 209
  renaming for SGML 212
  restrictions on 160

attribute (rule) 335
attribute types
  changing for SGML 220
  correspondence to SGML 208
  list of possible 160
attribute values
  default 164
  default values for SGML 219
  for choice attributes 163
  for IDReference attributes 168
  for numeric attributes 163
  for UniqueID attributes 167
  renaming for SGML 213
  required or optional 161
attributes
  comparison with SGML 12
  for a prefix or suffix 149, 170
  for cross-referencing 164-168
  for formatting objects 177
  for formatting text 123-125
  for formatting text or objects 169-170
  for identifying overrides 457
  hidden 162
  how end users work with 158
  in conversion tables 453
  read-only 162
  uses for 157
  writing definitions for 159-164
attributes, default translation with SGML
  attribute definitions 207-208
  attribute names 209
  attribute types and declared values 208
attributes, modifying translation with SGML
  changing attribute types 220
  discarding attributes 218
  renaming attributes 212
  renaming values 213
  specifying default values 219
  specifying read-only 221
attributes, read/write rules for 323
  attribute 335
  drop 342
  fm attribute 366
  fm element 367
  implied value is 377
  is fm attribute 385
  is fm property 396
  is fm property value 398
  is fm value 413
  value 439
autoinsertion rules 106-108
autonumbers
  defining formats for 140
  when to use 150

B

Basic properties (text formatting) 135-137
batch import and export (UNIX) 493-496
{before} sibling indicator 123, 177
{between} sibling indicator 123, 177
books
  comparison with SGML 15
  inheritance of text formats in 118
  internal and external references 298
  vs. text entities in SGML 315
books, default translation with SGML
  on export to SGML 318-319
  on import from SGML 316-317
books, modifying translation with SGML
  identifying book components 319-321
  suppressing creation of PIs 321
books, read/write rules for 324

generate book 373
output book processing instructions 422
put element 373
use processing instructions 373
brackets ([ ])
in cross-reference formats 35
in format rules 123, 178

C

CALS table model 461-466
attribute structure 465
colspec elements 465, 466
element and attribute declarations 463
element structure 465
spanspec elements 465, 466
CALS tables
EDD object format rules and 173
read/write rules for 467-469
CALS tables, translating from SGML
CALS attributes for formatting 250, 259
colspec and spanspec elements 250, 260
formatting properties for 252-256
how CALS tables translate 249
caret (^), in error messages 112
CDATA entities, default translation of internal 229
character encoding, XML 503
character formats
applying to particular contexts 132
finding errors in 154
wrapping text formatted without 457
character map (rule) 337
character set mapping 475-481
characters allowed
in attribute names 160
in choice attribute values 163
in conversion tables 449
in element tags 80
child elements
exclusions for 105
in general rules 99
inclusions for 104
inserted automatically 106-108
choice attributes 160
formatting elements with 124
specifying values for 163
comma (,)
in conversion tables 452
in general rules 100
commenting an EDD
with comment elements 83, 88
with paragraph elements 78
commenting read/write rules 202
containers
automatic descendants for 106-108
defining elements for 81-85
general rules for 99-103
validity at highest level 104
content rules 99-106
comparison with SGML 11
debugging 112
overview of 98
translation to SGML 106
context labels, in format rules 130
context rules (for formatting) 121-125, 175-178
ancestor tags in 121, 176
attributes in 123-125, 177
order of clauses in 125, 178
wildcard characters in 122, 176
conversion tables 445-459
adding rules to 448-455
attributes in 453
building tables from format tags with 458

columns and rows in 445, 448
documents for holding 446
flagging format overrides with 456
format and element tags in 447, 449, 450
generating initial 447
nesting graphics or tables with 457
object type identifiers in 450
order of rules in 446
promoting graphics or tables with 455
qualifiers for element tags in 449, 454
setting up from scratch 448
testing and correcting 459
updating 448
wrapping elements with 451
wrapping objects with 450
wrapping sequences with 452
wrapping untagged text with 457
cross-reference formats
for FrameMaker+SGML elements 34-35
cross-references
comparison with SGML 17
defining elements for 86-89
finding errors in formats of 185
formats for 34-35
IDReference attributes for 168
object format rules for 182
UniqueID attributes for 166-167
cross-references, default translation with SGML
internal and external references 298
on export to SGML 298
on import from SGML 299
cross-references, modifying translation with SGML
maintaining attribute values 301
renaming format attributes 300
translating elements as references 300
translating elements to SGML text 301
cross-references, read/write rules for 324
fm element unwrap 367
fm property 370
is fm cross-reference element 389
is fm property 396
is fm property value 398
is fm value 413
value is 370

D

debugging. *See* errors
default
attribute values 164
general rules 103
initial structure for tables 110
SGML declaration 471-473
descendants
exclusions for 105
inclusions for 104
inserted automatically 106-108
DOCTYPE elements 47
document type declarations (DTDs)
comparison with EDDs 9
creating from an EDD 93
EDD content rules and 106
EDD object format rules and 173
EDD text format rules and 113
errors when translating to an EDD 67
external DTD subsets 10, 32, 65
saving an EDD as 66
SGML declarations and 94
specifying location of 48
updating an EDD from 67
See also SGML, translation to and from
documentation, for applications 33

documents, comparison with SGML 14
drop content (rule) 344
drop (rule) 342
DTD. *See* document type declarations

E

EDD. *See* element definition documents
Element Catalogs
  creating in a template 90-92
  elements in an EDD catalog 69-76
  exporting to an EDD 68
element definition documents (EDDs) 63-95
  adding comments to 78
  comparison with DTDs 9
  creating from a DTD 66
  creating new 68
  errors when translating to a DTD 94
  exporting an Element Catalog to 68
  high-level elements in 69
  list of elements in 70-76
  overview of developing 22-25, 64
  samples to review 95
  saving as a DTD 93
  setting an SGML application in 77
  shortcuts for working in 89
  updating from a DTD 67
element definitions
  attribute definitions in 157-171
  basic steps for writing 79-89
  comments in 83, 88
  comparison with SGML 11
  creating formats when importing 77
  debugging 92
  element tags in 80
  element types in 83, 88
  errors when importing 91
  for containers, tables, and footnotes 81-85
  for object elements 86-89
  for Rubi groups 85-86
  guidelines for writing 80
  importing into a template 91
  object format rules in 173-186
  organizing in sections 78
  structure rules in 97-112
  text format rules in 113-155
element (rule) 345
element tags
  correspondence to SGML 209
  in conversion tables 447, 449, 450
  in element definitions 80
  in read/write rules 203
  renaming for SGML 211
element types
  comparison with SGML 10
  list of possible 83, 88
elements
  comparison with SGML 10-12
  discarding to or from SGML 218
elements, default translation with SGML
  element tags 209
  exclusions 210
  general rules and model groups 206
  inclusions 210
  line breaks and record ends 211
elements, modifying translation with SGML
  converting SGML elements to footnotes 213
  converting SGML elements to Rubi groups 214
  renaming elements 211
  retaining content but not structure 216
  retaining structure but not content 216
  suppressing display of content 217

elements, read/write rules for all 323
attribute 335
drop 342
drop content 344
element 345
fm element 367
is fm element 391
preserve fm element definition 422, 424
unwrap 436
else clauses, in format rules 121, 176
else/if clauses, in format rules 121, 176
EMPTY keyword, in general rules 101
end vertical straddle (rule) 348
entities 13
external files for 51-53
ISO public 483-491
searching for external files 54
searching for filename patterns 52
specifying location of 51
specifying search path for 53-55
entities, default translation with SGML
external data entities 231
external text entities 232
for marking documents in book 318
how used for variables 304
internal CDATA entities 229
internal SDATA entities 230
internal text entities 229
on export to SGML 226
on import from SGML 228-233
parameter entities 232
PI entities 232
SUBDOC entities 232
entities, modifying translation with SGML
changing structure and format of insets 243
discarding external data references 244
renaming entities for variables 235
translating characters as entities 244
translating external system entities as insets 241
translating SDATA as characters 237-238
translating SDATA as elements 240
translating SDATA as insets 238
translating SDATA as variables 236
translating SDATA references 235
translating text entities as insets 241
entities, read/write rules for 325
drop 342
entity 349
entity name is 352
external data entity reference 362
is fm char 387
is fm reference element 402
is fm text inset 411
is fm variable 414
reformat as plain text 429
reformat using target document catalogs 429
retain source document formatting 430
entity catalogs 33
format of entries in 50
searching for 50
specifying location of 49-50
uses for 50
entity name is (rule) 352
entity (rule) 349
equation (rule) 355
equations
comparison with SGML 16
defining elements for 86-89
finding errors in sizes of 185
in conversion tables 451
object format rules for 182

equations, read/write rules for 326
entity name is 352
equation 355
export dpi 357
export to file 359
fm property 370
is fm equation element 392
is fm property 396
is fm property value 398
is fm value 413
notation is 420
specify size in 431
value 439
value is 370
equations, translation with SGML. *See* graphics and equations
errors
in imported element definitions 91
in object format rules 185-186
in structure rules 112
in text format rules 154
when creating an EDD from a DTD 67
when saving an EDD as a DTD 94
exclusions
correspondence to SGML 210
in element definitions 105
when used with inclusions 104
export dpi (rule) 357
export to file (rule) 359
exporting to SGML. *See* SGML, translation to and from
exporting to XML 497
external data entity reference (rule) 362
external dtd (rule) 363
external DTD subsets 10, 32, 65

F

facet (rule) 364
files, for applications 41
{first} sibling indicator 123, 177
first format rules 143-145
how applied 144
when to use 150
with autonumbers 145
fm attribute (rule) 366
fm element (rule) 367
fm element unwrap (rule) 367
fm marker (rule) 368
fm property (rule) 370
fm variable (rule) 372
fmsgml version (rule) 373
Font properties (text formatting) 137-139
footnotes
converting SGML elements to 213
defining elements for 81-85
general rules for 99-103
in conversion tables 451
inheritance of text formats in 117
footnotes, read/write rules for 327
is fm footnote element 393
format change lists
changes that apply from 152
defining 151
limits on values in 152
referring to 131, 132
format overrides, flagging in conversion tables 456
format rule overrides
object format rules and 174
text format rules and 115
format rules
comparison with SGML 15

See also object format rules *and* text format rules
format tags
in conversion tables 447, 449, 450
in text format rules 119, 131
formats
creating automatically in templates 77
storing in templates 91

G

general rules 99-103
avoiding ambiguous 100, 102
child elements in 99
correspondence to SGML 206
default 103
for empty elements 101
grouping elements in 101
restrictions on tables 102
specifying text in 101
syntax of 99-102
table formats and 179
generate book (rule) 373
graphics
comparison with SGML 15
defining elements for 86-89
nesting in conversion tables 457
object format rules for 179
promoting in conversion tables 455
graphics and equations, default translation with SGML
anchored frame properties 278-280
creating graphic files on export 282
element and attribute structure 277
entity and file attributes 278
exporting entity declarations 281
graphic properties 280
on export to SGML 276-281
on import from SGML 282
text of default declarations 276
graphics and equations, modifying translation with SGML
changing format of files 291
changing name of files 290
changing size of graphics 295
exporting elements 285
omitting elements and attributes 288
omitting graphic properties 288
renaming attributes for properties 286
renaming elements 284
representing structure of equations 286
specifying data content notation 289
specifying entity names 294
graphics, read/write rules for 327
anchored frame 333
entity name is 352
export dpi 357
export to file 359
facet 364
fm property 370
is fm graphic element 394
is fm property 396
is fm property value 398
is fm value 413
notation is 420
specify size in 431
value 439
value is 370

H

hidden and read-only attributes
specifying in definitions 162
HTML

conversion macros 39
elements mapped from FrameMaker+SGML elements 38
export 34
mapping for export 37
hyphenation settings 141

I

IDReference attributes
comparison with SGML 17
defining 168
using for cross-references 164
if clauses, in format rules 121, 176
implied value is (rule) 377
imported graphic files. *See* graphics
importing element definitions 91
importing from SGML. *See* SGML, translation to and from
include dtd (rule) 379
INCLUDE keyword, in rules documents 202
include sgml declaration (rule) 380
inclusions
correspondence to SGML 210
in element definitions 104
when used with exclusions 104
indentation settings 135
inheritance of formatting information 115-119
in tables or footnotes 117
within books 118
initial conversion tables 447
initial structure pattern, for Rubi groups 111
initial structure pattern, for tables 108-110
insert table part element (rule) 381
is fm attribute (rule) 385
is fm char (rule) 387
is fm colspec (rule) 389
is fm cross-reference element (rule) 389
is fm element (rule) 391
is fm equation element (rule) 392
is fm footnote element (rule) 393
is fm graphic element (rule) 394
is fm marker element (rule) 395
is fm property (rule) 396
is fm property value (rule) 398
is fm reference element (rule) 402
is fm rubi element (rule) 404
is fm rubi group element (rule) 405
is fm spanspec (rule) 406
is fm system variable element (rule) 407
is fm table element (rule) 408
is fm table part element (rule) 409
is fm text inset (rule) 411
is fm value (rule) 413
is fm variable (rule) 414
is processing instruction (rule) 415
ISO Latin-1 character set 475-481
ISO public entities 483-491
declarations and rules 488-491
default character formats 487
default variable definitions 487
entity declaration files 485
entity read/write rules files 485
format of entity rules 486

K

keyboard shortcuts for an EDD 89

L

{last} sibling indicator 123, 177
last format rules 143-145
how applied 144
when to use 150

with autonumbers 145
level rules (for formatting) 125-127
ancestor tags in 126
counts using current element 126
order of clauses in 126
limits on formatting values 152
line break (rule) 417
line breaks and SGML record ends 211
log files
generating 58
hypertext links in 59
limiting length of 57
messages in 58
See also errors

M

marker text is (rule) 418
markers
comparison with SGML 17
defining elements for 86-89
list of predefined types 181
object format rules for 180
markers, default translation with SGML
on export to SGML 310
on import from SGML 310
markers, modifying translation with SGML
discarding nonelement markers 313
identifying markers with attributes 312
translating elements as markers 311
writing marker text as content 311
markers, read/write rules for 328
drop 342
external data entity reference 362
fm marker 368
fm property 370
is fm marker element 395
is fm property 396
is fm property value 398
is fm value 413
is processing instruction 415
marker text is 418
processing instruction 425
value 439
value is 370
{middle} sibling indicator 123, 177

N

nested format rules 128
notation is (rule) 420
{notfirst} sibling indicator 123, 177
{notlast} sibling indicator 123, 177
Numbering properties (text formatting) 140
numeric attributes 160
specifying a range for 163
values allowed in 161

O

object format rules 173-186
all-contexts rules in 175
context rules in 175-178
debugging 185-186
for cross-references 182
for equations 182
for graphics 179
for markers 180
for system variables 183
for tables 178
format rule overrides and 174
overview of 174
seeing which rules apply 185
sibling indicators in 177
translation to SGML 173

object type identifiers, in conversion tables 450
{only} sibling indicator 123, 177
operators with attributes, for formatting 124
output book processing instructions (rule) 422

P

Pagination properties (text formatting) 139
paragraph formats
  applying to particular contexts 131
  building table structure from 458
  finding errors in 154
  how inherited from ancestors 115-119
  setting a base element format 119
paragraphs
  as a prefix or suffix 147, 148
  formatting elements as 131
parentheses
  in conversion tables 452
  in general rules 101
PIs. *See* processing instructions
plus sign (+)
  in conversion tables 452
  in general rules 99
prefix rules 145-150
  attributes in 149
  defining for paragraphs only 147
  defining for ranges and paragraphs 148
  defining for text ranges only 146
  how applied 146
  when to use 150
preserve fm element definition (rule) 422, 424
processing instruction (rule) 425
processing instructions (PIs), default translation with SGML
  for marking book or document 316
  on export to SGML 226
  on import from SGML 233
  PI entities 232
processing instructions (PIs), modifying translation with SGML
  discarding unknown instructions 245
  suppressing creation of in books 321
processing instructions (PIs), read/write rules for 329
  drop 342
  fm marker 368
  is processing instruction 415
  output book processing instructions 422
  processing instruction 425
  use processing instructions 373
PROMOTE keyword 455, 456
proportional width resolution is (rule) 426
public identifiers 53
put element (rule) 373

Q

qualifiers, in conversion tables 445, 449, 454
question mark (?)
  in conversion tables 452
  in general rules 99
quotation marks ("), in attribute values 454

R

read/write rules
  case conventions for 200
  checking correctness of 203
  commands for working with 203
  comments in 202
  constants in 201
  documents for 33, 47, 197
  for CALS tables 467-469
  for creating an EDD from a DTD 65

including files with 55, 202
order of rules 199
overview of developing 26-29
reserved element names in 203
strings in 200
summary of 323-331
syntax for 199
uses for 189
variables in 201
XML 504
See also specific rules and categories of rules
reader (rule) 427
read-only attributes
creating with read/write rules 221
reformat as plain text (rule) 429
reformat using target document catalogs (rule) 429
relative values, in format rules 116
retain source document formatting (rule) 430
Rubi groups
converting SGML elements to 214
defining elements for 85-86
initial structure pattern for 111
Rubi groups, read/write rules for
is fm rubi element 404
is fm rubi group element 405

S

sample documents
for testing an application 25-26
that come with FrameMaker+SGML 95
SDATA entities
default translation of internal 230
translating as characters 237-238
translating as reference elements 240
translating as text insets 238
translating as variables 236
translating entity references 235
SGML
comparison with FrameMaker+SGML 9-19
features not in FrameMaker+SGML 18
optional unsupported features 19, 474
SGML API clients
specifying location of 56
uses for 190
SGML applications
defining 44
location of files in 41
overview of 4-7
pieces of 32-34
process of developing 22-31
scenarios for 3
setting in an EDD 77
SGML declarations 32
default for FrameMaker+SGML 471-473
for a DTD created from an EDD 94
specifying location of 48
SGML documents, read/write rules for 329
external dtd 363
include dtd 379
include sgml declaration 380
write sgml document 441
write sgml document instance only 441
SGML parser
applying to a DTD 94
concrete syntax variants 473
SGML PI markers
for storing entities and PIs 227
SGML, translation to and from
batch export (UNIX) 495
batch import (UNIX) 493-495
books 315-321

cross-references 297-302
EDD content rules 106
elements and attributes 205-223
entities and PIs 225-245
example of 191-195
graphics and equations 273-296
markers 309-313
overview of 189-195
Rubi groups 214-215
tables 247-272
UniqueID values 167
variables 303-308
See also read/write rules *and specific element types*
sibling indicators
in format rules 122, 177
used with TEXT keyword 123, 177
spacing settings
for lines 136
for paragraphs 136
for words 142
line spacing and font sizes 136
specify size in (rule) 431
start new row (rule) 433
start vertical straddle (rule) 434
string attributes 160
structure, adding to documents. *See* conversion tables
structure rules 97-112
debugging 112
overview of 98
Structure View
attributes in 158
cross-references in 165
invalid contents in 98
style sheets for XML 499
subrules, for format rules 128
suffix rules 145-150
attributes in 149
defining for paragraphs only 147
defining for ranges and paragraphs 148
defining for text ranges only 146
how applied 146
when to use 150
syntax, XML 499
system variables
defining elements for 86-89
finding errors in definitions of 185
in conversion tables 451
list of predefined variables 184
object format rules for 183
system variables, default translation with SGML
entities for nonelement variables 304
on export to SGML 304
on import from SGML 305
system variables, modifying translation with SGML
discarding variables 307
renaming or changing entity types 305
translating as variable elements 306
translating SDATA entities 236
translating to SGML text 307

T

tab stop settings 137
Table Cell properties (text formatting) 142
table ruling style is (rule) 435
tables
building structure from format tags 458
comparison with SGML 16
default general rules for 103
default initial structure for 110

defining elements for 81-85
finding errors in formats of 185
formatting cells in 142
general rules for 99-103, 179
inheritance of text formats in 117
initial structure pattern for 108-110
nesting in conversion tables 457
object format rules for 178
paragraph formats for 119
promoting in conversion tables 455
restrictions on general rules 102
tables, default translation with SGML
on export to SGML 251
on import from SGML 248-251
tables, modifying translation with SGML
creating parts without content 264-266
creating tables inside tables 272
creating vertical straddles 267-270
exporting widths proportionally 267
formatting as boxed paragraphs 270
formatting with CALS attributes 259
identifying colspecs and spanspecs 260
omitting representation of parts 262-264
renaming table parts 256
representing properties as attributes 257
representing properties implicitly 258
rotating tables on a page 272
specifying columns for cells 261
specifying location of rows or cells 260
specifying ruling style for tables 266
table cell paragraph properties 256
table format properties 253-254
table straddle properties 254
tables, read/write rules for 329
end vertical straddle 348
fm property 370
insert table part element 381
is fm colspec 389
is fm property 396
is fm property value 398
is fm spanspec 406
is fm table element 408
is fm table part element 409
is fm value 413
proportional width resolution is 426
start new row 433
start vertical straddle 434
table ruling style is 435
use proportional widths 438
value 439
value is 370
templates 33
creating an Element Catalog in 90-92
creating formats automatically in 77
specifying location of 48
storing formats in 91
text format rules 113-155
all-contexts rules in 121
context labels in 130
context rules in 121-125
debugging 154
element paragraph format in 119
first and last format rules 143-145
formatting specifications in 133-143
how inherited from ancestors 115-119
level rules in 125-127
limits on values in 152
multiple format rules 128
nested format rules 128
no additional formatting 132
overview of 114
paragraph formatting with 131

seeing which rules apply 154
sibling indicators in 122
text range formatting with 131
translation to SGML 113
text insets, read/write rules for 331
entity 349
is fm text inset 411
reformat as plain text 429
reformat using target document catalogs 429
retain source document formatting 430
TEXT keyword
in general rules 101
used with sibling indicators 123, 177
text ranges
as a prefix or suffix 146, 148
formatting elements as 131
text, read/write rules for 331
character map 337
entity 349
is fm char 387
line break 417
TEXTONLY keyword
changing SGML declared content 215
in general rules 101
Type 11 markers
discarding constructs using 245

U

UniqueID attributes
comparison with SGML 17
defining 166
translating IDs to SGML 167
using for cross-references 164
values provided by end users 167
values provided by system 167
untagged formatted text, wrapping 457
unwrap (rule) 436
use processing instructions (rule) 373
use proportional widths (rule) 438
user variables
in container elements 184
in conversion tables 451
user variables, default translation with SGML
entities for variables 304
on export to SGML 304
on import from SGML 305
user variables, modifying translation with SGML
discarding variables 307
renaming or changing entity types 305
translating SDATA entities 236
Using attibutes in a CALS table model 461

V

validity at highest level, specifying 104
value is (rule) 370
value (rule) 439
variables, read/write rules for 331
drop 342
entity 349
fm element unwrap 367
fm variable 372
is fm system variable element 407
is fm variable 414
variables. *See* system variables *or* user variables
version numbers in EDDs 77
vertical bar (|)
in conversion tables 452
in format rules 122, 176
in general rules 100

W

wildcard characters

in conversion tables 449
in format rules 122, 176
wrapping with conversion tables
document objects 450
elements 451
sequences of elements or paragraphs 452
untagged formatted text 457
write sgml document instance only (rule) 441
write sgml document (rule) 441
writer (rule) 442

X

XML
character encoding 503
CSS style sheets 499
export 497
read/write rules 504
syntax 499